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

Soichiro Hidaka
National Institute of Informatics, Japan
hidaka@nii.ac.jp

Martin Billes
Technical University of Darmstadt, Germany
martin.billes@cased.de

Quang Minh Tran
Daimler Center for IT Innovations, Technical University of Berlin, Germany
quang.tranminh@dcaiti.com

Kazutaka Matsuda
Tohoku University
kztk@ecei.tohoku.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. The connections view elements have among each other and with source elements, which lead to restrictions of view editability, and parts of the transformation which are responsible for these relations, are not apparent to the user of a bidirectional transformation program.

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 where the target of a graph transformation can be the source of another 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.

Keywords: Bidirectional Graph Transformation, Traceability, Editability

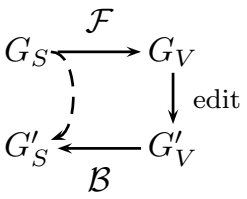
Copyright © by the paper's authors. Copying permitted for private and academic purposes.

In: A. Cunha, E. Kindler (eds.): Proceedings of the Fourth International Workshop on Bidirectional Transformations (Bx 2015), L'Aquila, Italy, July 24, 2015, published at http://ceur-ws.org

1 Introduction

Bidirectional transformations has been attracting interdisciplinary studies [5, 20]. 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.

A bidirectional transformation consists of forward ($\mathcal{F}$) and backward ($\mathcal{B}$) transformations [5, 20]. $\mathcal{F}$ takes a source model (here a source graph [15, 16]) G_S and transforms it into a view (target) graph G_V. $\mathcal{B}$ takes an updated view graph G'_V and returns an updated source graph G'_S, with possibly propagated updates.



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 (for example, operators or variables) in the transformation, and what that part is. Thus, it is not easy to tell where edits to the view edge are propagated back to. Moreover, in a setting in which usual strong well-behaved properties like PutGet [8] is relaxed to permit the backward transformation to be only partially defined, like WPutGet (WeakPutGet) [13, 16], it is not easy to predict whether a particular edit to the view succeeds. In particular, backward transformation rejects updates if (1) the label of the edited view edge appears as a constant of the transformation[1], (2) a group of view edges that are originated from the same source edge are edited inconsistently or (3) edits of view edges lead to changes in control flow (i.e., different branch is taken in the conditional expressions) 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 [7]. 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 (well-behavedness) can be recognized by the user early.

Our bidirectional transformation framework called GRoundTram (Graph Roundtrip Transformation for Models) [18, 15, 16] features compositionality (e.g., the target of a sub-transformation can be the source of another sub-transformation), a user-friendly surface syntax, a tool for validating both models and transformations with respect to KM3 metamodels, and an performance optimization mechanism via transformation rewriting. It suffers from the above issues. To fix this, 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 the information for correspondence and editability analysis. For the user, corresponding elements in the artifacts are highlighted with different colors according to editability and other properties.

There are some existing work with similar objectives. Traceability is studied enthusiastically in model-driven engineering [9, 25]. Van Amstel et al. [30] proposed a visualization of traces, but in the unidirectional model transformation setting. We focus on the backward transformation to give enough information on editability of the views for the programmer of the transformation and the users who edit the views. For classification of elements in the view, Matsuda and Wang's work [24], an extension of the semantic approach [31] to general bidirectionalization, is also capable of a similar classification, while we reserve opportunities to recommend a variety of consistent changes for more complex branching conditions. In addition, our approach can trace between nodes of the graph, not just edges. More details can be found in the related work section (Section 6).

The rest of the paper is organized as follows: Section 2 overviews our motivation and goal with a running example. More involved examples can be found on our project website at `http://www.prg.nii.ac.jp/projects/gtcontrib/cmpbx/`. Section 3 summarizes the semantics of our underlying graph data model, core graph language UnCAL [3], and its bidirectional interpretation [13]. 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 discusses related work, and Section 7 concludes the paper with future work. The long version of our paper [12] includes some non-essential technical details that are omitted in this paper.

[1] In this case, the constant in the transformation is pointed out so that user can consider changing it in the transformation directly.

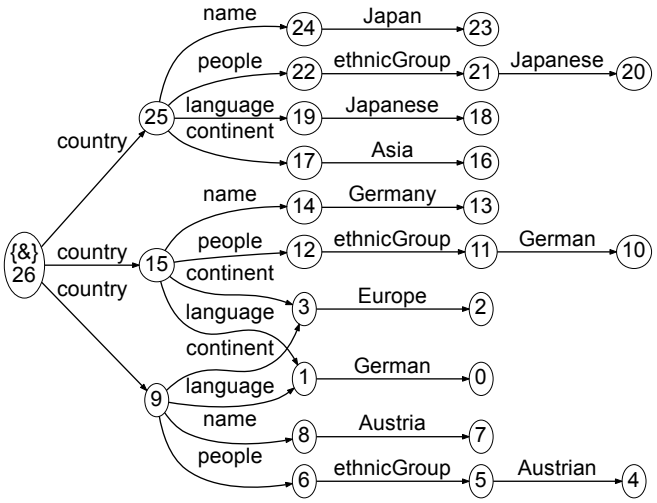


Figure 1: Example Source Graph

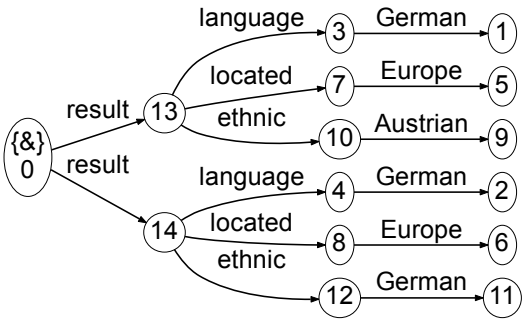


Figure 2: View Graph Generated by Transformation of the Graph in Fig. 1

2 Motivating Example

This section exemplifies the importance of the desirable 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 as a syntactic sugar of the 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: Transformation in UnQL

S1: In the view graph (Fig. 2), 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. For example, the edge $\zeta = (3, \text{German}, 1)$ is the language of Germany and is a 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.

S2: 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. 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.

S3: 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. Ideally, the system would highlight groups of edges that could cause the conflict, and would prohibit triggering the backward transformation in that case.

S4: Finally, suppose both of the edges labeled "Europe" in the view graph are edited to "Eurasia". Although this time these updates would be propagated to the same originating source edge $(3, \text{Europe}, 2)$ without conflict, 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 the control flow are very difficult or impossible to predict if the transformation is too complex. We can assist the prediction by highlighting the conditional branches involved.

Formal property As a remark, our editability checking is sound in the sense that the edits that passed the checking always succeed. We provide a proof sketch in Section 5.

Using this property and analysis, additional interesting observations can be made. In the Class2RDB example on our project website, there is no edge in the view for which the above last warning is caused. Thus, the transformation is "fully polymorphic" in the sense that all edges that come from source graph are free from label inspection in the transformation. Note that this editability property may not be fully determined by just looking at the transformation. Non-polymorphic transformations may still produce views that avoid the warning. For example,

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

is not polymorphic, but the edits on edges in the view never cause condition expressions to be changed, because they are not inspected (they are inspected by bulk semantics explained in the next section but left unreachable).

Tracing mechanism Though the editability analysis is powerful enough to, for example, cope with complex interactions between graph variables and label variables, the underlying idea is simple. For the analysis of a chain of transformations, if an edge ζ is related to ζ' in the first transformation and ζ' is related to ζ'' in the second transformation, then trace information allows to relate ζ and ζ'', possibly with the occurrence information of the language constructs in the transformation that created these edges. The variable binding environment is extended to cope with this analysis.

Support of editing operations other than edge renaming As another remark, this paper focuses on updates on edge labels, and trace information is designed as such. However, we can relatively easily extend the data structure of the trace to cope with edge deletion. The data structure already supports node tracing, so insertion operation can be supported by highlighting the node in the source on which a graph will be inserted.

Support of permissive bidirectional transformation The bidirectional transformation we are dealing with under relaxed notion of consistency, i.e., WeakPutGet as mentioned in the introduction, in fact is permissive in users edits. For example, in the above scenario **S3**, if the user edits only one of the view edge, then the backward transformation successfully propagates the changes to the source edge, though the updated view graph on the whole is not perfectly consistent if we define source s and view v to be consistent if and only if $gets = v$ where *get* is the forward transformation, because another forward transformation from the updated source would propagate the new edge label to the other copy edge in the view graph. The perfectly consistent view graph would have been the one which all the edges originating from a common source edge are updated to the same value. However, it might be hard for the users to identify all these copies. Therefore, our system accepts updates in which only one copy is updated in the view. Borrowing the notion of degree of consistency by Stevens [28], such updated views are (strictly) less consistent than the perfect one but still tolerated.

3 Preliminaries

We use the UnCAL (Unstructured CALculus) query language [3]. UnCAL has an SQL-like syntactic sugar called UnQL (Unstructured Query Language) [3]. Listing 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 [13]. 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 $Label \cup \{\varepsilon\}$ ($Label_\varepsilon$), 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 [3], and graphs can be normalized [16] up to isomorphism.

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 $Edge_\varepsilon$, where an edge is represented by a triple of source node, label and destination node. $I : Marker \to V$ 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. $O \subseteq V \times Marker$ 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

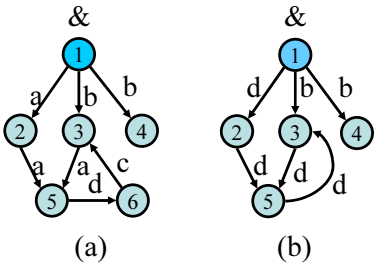


Figure 3: Cyclic graph examples

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

Figure 4: Core UnCAL Language

input nodes serve as "entry points". For example, the graph in Fig. 3 (a) is represented by (V, E, I, O), where $V = \{1, 2, 3, 4, 5, 6\}$, $E = \{(1, \mathtt{a}, 2), (1, \mathtt{b}, 3), (1, \mathtt{b}, 4), (2, \mathtt{a}, 5), (3, \mathtt{a}, 5), (5, \mathtt{d}, 6), (6, \mathtt{c}, 3)\}$, $I = \{\texttt{\&} \mapsto 1\}$, and $O = \{\}$. Each component of the quadruple is denoted by the "." syntax, such as $g.\mathrm{V}$ 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 $DB^{\{\texttt{\&}\}}_{\emptyset}$. The superscript may be omitted, if the set is $\{\texttt{\&}\}$, and the subscript likewise, if the set is empty. The type of this graph is simply denoted by DB.

3.2 UnCAL Query Language

Graphs can be created in the UnCAL query language [3], 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., $\mathcal{F}[\![\texttt{\&}y]\!] \in DB_{\{\texttt{\&}y\}}$.

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 $g.\mathrm{I}(\texttt{\&})$; thus $\{l : g\} \in DB_{\mathcal{Y}}$. The union $g_1 \cup g_2$ of

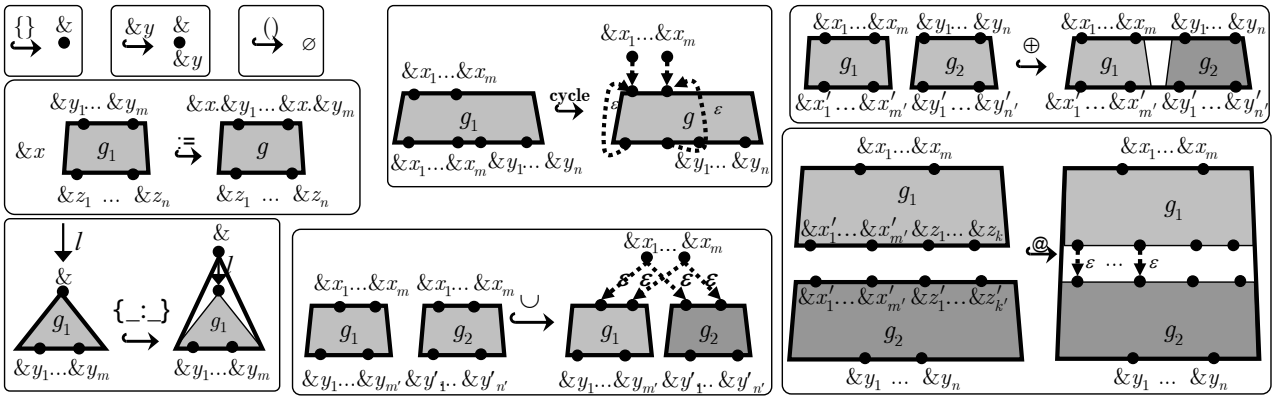


Figure 5: Graph Constructors of UnCAL

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., $(\&x := g) \in DB^{\{\&x\}}_{\mathcal{Y}}$ 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 $g_1 \mathbin{@} g_2 \in DB^{\mathcal{X}}_{\mathcal{Y}}$. An idiom $\&x' @ g_2$ projects (selects) one input marker $\&x'$ and renames it to default ($\&$), while discarding the rest of the input markers (making them unreachable). The cycle construction $\mathbf{cycle}(g)$ for $g \in DB^{\mathcal{X}}_{\mathcal{X} \cup \mathcal{Y}}$ 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), where the notion of bisimilarity is extended to ε-edges [3].

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 [14].

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

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

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 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 [13] *StrID*:

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

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. 6, the node RN 7 1, originating from node 1, is created by **rec** at position 7

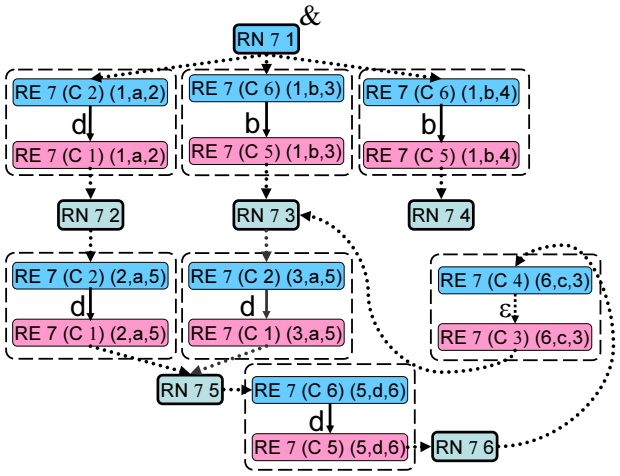


Figure 6: Bulk Semantics by Example

(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 $(\{(\mathsf{C}\ 2), (\mathsf{C}\ 1)\}, \{(\mathsf{C}\ 2, \mathtt{d}, \mathsf{C}\ 1)\}, \{\mathtt{\&} \mapsto \mathsf{C}\ 2\}, \{(\mathsf{C}\ 2, \mathtt{\&})\})$, 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 ζ 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 $\boxed{\mathsf{RE}\ 7\ (\mathsf{C}\ 2)\ (1, \mathtt{a}, 2)}$ 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 [5, 27], while ours satisfy the GetPut and WPutGet properties [13], 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* [6], a weaker notion of PutGet [8] or Correctness [27] or Consistency [2] 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 [3] 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 [3, 19, 17] for details. The expression (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.

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

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 in the trace 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 $\mathcal{F}[\![\_]\!] : \mathit{Expr} \to \mathit{Env} \to \mathit{Graph} \times \mathit{Trace}$ takes an UnCAL expression and an environment as arguments, and produces a target graph and trace. The shaded part represents the augmented part and we apply the same shading in the following. 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

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

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, the top level environment ρ_0 is initialized as follows.

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

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.

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

Now we describe the semantics $\mathcal{F}[\![\_]\!]$. The graph component returned by the semantics is the same as that of [13] 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. For the constructors, we only show the semantics of

some representative ones. See the long version [12] for the rest.

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

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

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

For the constructor $\{\}$, the trace maps the only node ($G.\mathrm{I}(\&)$) created, to the code position p of the constructor (T-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). 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.

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.

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

$$\mathcal{F}_\mathrm{L}[\![\$l^p]\!]_\rho = (l, p : \tau) \tag{LVAR}$$
$$\textbf{where } (l, \tau) = \rho(\$l)$$

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.

The 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 of the variable reference to it.

$$\mathcal{F}[\![\textbf{llet}^p\ \$l = e_\mathrm{L}\ \textbf{in}\ e]\!]_\rho = \mathcal{F}[\![e]\!]_{\rho \cup \{\$l \mapsto (l, p:\tau)\}} \tag{LLET}$$
$$\textbf{where } (l, \tau) = \mathcal{F}_\mathrm{L}[\![e_\mathrm{L}]\!]_\rho$$

$$\mathcal{F}[\![\textbf{let}^p\ \$g = e_1\ \textbf{in}\ e_2]\!]_\rho = \mathcal{F}[\![e_2]\!]_{\rho \cup \{\$g \mapsto (g, \mathsf{prep}_p\, t)\}} \tag{LET}$$
$$\textbf{where } (g, t) = \mathcal{F}[\![e_1]\!]_\rho$$

$$\mathcal{F}[\![\$g^p]\!]_\rho = (g, \mathsf{prep}_p\, t) \tag{VAR}$$
$$\textbf{where } (g, t) = \rho(\$g)$$

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

Structural recursion **rec** (REC) introduces a 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 $M : \mathit{Edge} \to (\mathit{Graph} \times \mathit{Trace})$ 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.

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

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

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. 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

This section elaborates utilization of the traces defined in Sect. 4 for the correspondence and editability analysis motivated in Sect. 2. Soundness of this analysis is discussed at the end of this section.

Given transformation e, environment ρ_0 (defined by InitEnv), and the corresponding trace t for $(g,t) = \mathcal{F}[\![e]\!]_{\rho_0}$ through semantics given in Sect. 4, the trace (represented as a list) 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 the label of ζ is a copy of the 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 $\{(x', x) \mid (x \mapsto \tau) \in t, x' = \mathsf{last}\, \tau, x' \in (\mathit{Node} \cup \mathit{Edge})\}$. 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 is obtained. Given the view edge $\zeta = (3, \texttt{German}, 1)$, the code positions of the graph variables $\$\mathit{lang}$ of the **select** part and $\$\mathit{db}$ in Listing 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.

In the following, although the trace t can be used for both nodes and edges, we only focus on edges instead of nodes, and edge renamings as the update operations. However, the node trace can be used to support insertion operation by highlighting the node in the source on which a graph corresponding to the graph to be inserted in the view will be attached. For the deletion operation, we can relatively easily extend the data structure of the trace defined in the previous section to be tree-structured, being able to trace multiple sources, and extend rule (Rec) to associate each edge created by the body expression with the trace of the edge bound to the label variable.

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 $[\zeta]_\sim$. We define our updates as $\mathsf{upd} : \mathit{Upd}$ where $\mathit{Upd} = \mathit{Edge} \to \mathit{Label}$, an expression and its environment at position p as $\mathsf{expr} : \mathit{Pos} \to \mathit{Expr} \times \mathit{Env}$, and update propagation along trace t as $\mathsf{prop} : \mathit{Env} \to \mathit{Upd} \to \mathit{Env}$. Then, our editability analysis is defined as follows.

Definition 1 (Editability checking)**.** *Given $(g,t) = \mathcal{F}[\![e]\!]_{\rho_0}$ and update upd on g, editability checking succeeds if all the following three conditions are satisfied.*

1. *For all updated edges ζ, other view edges in $[\zeta]_\sim$ are unchanged or consistently updated, i.e., $\forall\zeta \in \mathrm{dom}(\mathsf{upd}).\forall\zeta' \in [\zeta]_\sim.\zeta' \notin \mathrm{dom}(\mathsf{upd}) \vee \mathsf{upd}(\zeta) = \mathsf{upd}(\zeta')$.*
2. *For every $\mathbf{if}\ e_\mathrm{B} \ldots$ expression in the backward evaluation path, applying the edit to the binding does not change the condition, i.e., $\forall p \in \{\cup[p \mid p \in t(\zeta), p \in \mathit{Pos}] \mid \zeta \in \mathrm{dom}(\mathsf{upd})\}, \mathsf{expr}(p) = (\mathbf{if}\ e_\mathrm{B} \ldots, \rho), \mathcal{F}[\![e_\mathrm{B}]\!]_\rho = \mathcal{F}[\![e_\mathrm{B}]\!]_{\mathsf{prop}(\rho,\mathsf{upd})}$. For the case of label variables, this interference can be checked by $\$l \in \mathcal{FV}(e_\mathrm{B}), \rho'(\$l) = (\_,\tau), \mathsf{last}(\tau) = \zeta'$ for $\zeta' = \mathsf{origin}_\mathrm{E}\,\zeta$. Weakened version for graph variables is: $\forall\$g \in \mathcal{FV}(e_\mathrm{B}).(g',t') = \rho'(\$g), \forall\zeta'' \in g'.\mathrm{E}.\,\mathsf{last}(t'((\zeta'')) \neq \mathsf{origin}_\mathrm{E}(\zeta)$ for all $\zeta \in \mathrm{dom}(\mathsf{upd})$.*
3. *No edited edge trace to code position, i.e., $\forall\zeta \in \mathrm{dom}(\mathsf{upd}).\,\mathsf{origin}(\zeta) \notin \mathit{Pos}$.*

Further, we recap all the three run-time errors to reject updates [13] as (1) failure at environment merging $\uplus$ ("inconsistent change detected"), (2) failure at $\mathcal{B}[\![\mathbf{if}\ldots]\!]$ ("branch behavior changed") and (3) failure at $\mathcal{B}_\mathrm{L}[\![\mathtt{a}]\!]$("no modifications allowed for label literals."). Then the following lemmas hold.

Lemma 1. *Condition 1 in Def. 1 is false iff error (1) occurs.*

Lemma 2. *Error (2) implies condition 2 in Def. 1 is false.*

Lemma 3. *Condition 3 in Def. 1 is false iff error (3) occurs.*

Based on these lemmas, we have the following theorem.

Theorem 1 (Soundness of editability analysis)**.** *Given forward transformation $(g,t) = \mathcal{F}[\![e]\!]_{\rho_0}$ and update upd on g to g', success of editability checking in Def. 1 implies $\mathcal{B}[\![e]\!](\rho_0, g')$ defined and results in $\mathsf{prop}(\rho_0, \mathsf{upd})$.*

Thus 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 the checking in Def. 1 succeeds. An edit on the view edge ζ with $\mathsf{origin}(\zeta) = p \in \mathit{Pos}$ is not propagable to the source by Lemma 3. 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 with the source graph of Fig. 1 and view graph of Fig. 2. We get four equivalence classes, one each for the source edges $(1, \mathtt{German}, 0)$, $(3, \mathtt{Europe}, 2)$, $(5, \mathtt{Austrian}, 4)$ and $(11, \mathtt{German}, 10)$, as well as the class that violate condition 3. For view edge $\zeta = (3, \mathtt{German}, 1)$, we have $(4, \mathtt{German}, 2) \in [\zeta]_\sim$ via $\mathsf{origin}_\mathrm{E}\,\zeta = (1, \mathtt{German}, 0)$, so these equivalent edges can be selected simultaneously, inconsistent edits on which can be prevented (Lem. 1). Direct edits of the view edge $\zeta = (0, \mathtt{result}, 14)$ are suppressed since $\mathsf{origin}\,\zeta \in \mathit{Pos}$ (Lem. 3). On condition 2, when the view edge $\zeta = (7, \mathtt{Europe}, 5)$ is given, all the code positions in t ζ'' 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 $(\$l = \mathtt{Europe})$ for warning.

Soundness Proof of the Editability Analysis

Cases in which completeness is lost: Note that system may choose the weakened version of condition 2, and just issue warning (i.e., whenever the edge being updated has common source edge with any variable reference occurring free in any of the conditional expressions along the execution path), because when condition includes not only the simple label comparison expressions but also graph emptiness checking by $\mathsf{isEmpty}$, the cost of fully checking the change of the condition may amount to re-executing the most part of the forward transformation. If this warning-only strategy is chosen, backward transformation may succeed despite this warning, because the warning does not necessarily mean condition value changes. Therefore, the checking is not complete (may warn some safe updates). We argue that this "false alarm" is not so problematic in practice, with the following reasons. The argument of $\mathsf{isEmpty}$ usually includes **select** to test an existence of certain graph pattern. So we further analyze conditions in **select** down to simple label equivalence check. If the argument is a bare graph variable reference, then we analyze its binder. If the graph variable is $\$db$ (the entire input graph), then any changes

cause false alarms. This predicate may be used only for debugging purpose to exclude empty source graphs, but otherwise we don't think we frequently encounter this situation.

Now we provide a proof sketch. First, we prove Lemma 3. We do this by parallel case analysis on augmented forward semantics and backward semantics [13].

Base case No edit is possible for $\{\}, ()$ and $\&y$ that produce no edge. For a transformation $\{\mathtt{a} : e'\}^p$ and the view graph as $v \xrightarrow{\mathtt{a}} G$, since $\mathtt{a}$ is a constant, augmented forward semantics generates code position p as trace, which violates condition 3 when $\mathtt{a}$ is being updated. Backward semantics rejects with error (3).

Inductive case

Transformation $\{\$l : e'\}^p$ and the view graph as $v \xrightarrow{l} G$: Update on l to l' will be handled by the backward semantics of $\$l$, which updates its reference. Suppose it is introduced by an expression $\mathbf{llet}\ \$l = \mathtt{a}\ \mathbf{in}\ \{\$l : e'\} \ldots$. Then binding $\$l \mapsto l'$ is generated and it is passed to the backward evaluation of label expression $\mathtt{a}$, which is constant. So no value other than $\mathtt{a}$ is allowed. So backward transformation fails with error (3). The trace generates code position of label constant $\mathtt{a}$, which also signals violation of condition 3. If $\$l$ is bound by $\mathbf{rec}(\lambda(\$l, \_).\{\$l : e_1\})(e_2)$, then the update is also translated to the argument expression e_2, so if corresponding part in e_2 originates from a label constant, then the condition is violated and backward transformation fails similarly.

$e_1 \cup e_2, e_1 \oplus e_2, e_1 \mathbin{@} e_2$: We reduce the backward transformation to that of subexpressions, assuming decomposition operation for each operator. So assuming Lemma 3 for these subexpressions, Lemma 3 holds for entire expression. Similar argument applies for $\mathbf{cycle}(e)$, $\&\_ := e$ and $\mathbf{if}\ \_\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2$.

$\mathbf{rec}(e_1)(e_2)$: What is produced as a trace depends on e_1 and e_2. Assuming the decomposition of target graph, the backward evaluation is reduced to those of e_1 and e_2. The only interesting case is a graph variable reference as (sub) expression. The update on the target graph that was produced by the variable reference is reduced to updated graph variable bindings, and the bound graph is aggregated for each edge in the input graph that was produced by e_2, and passed to the backward evaluation of e_2. The trace, on the other hand, is produced by e_2 and combined with the trace by e_1. If e_1 is a graph variable reference, then the (sub) trace induced from the trace from e_2 is used, so the edge that trace back to constant is inherited in the trace of the result of **rec**, so attempt to update an edge that traces back to the constant produced by e_2 and carried by the graph variable reference is signaled with condition 3. So, assuming Lemma 3 on e_2 by induction hypothesis, Lemma 3 holds for the entire expression.

Thus conclude the proof for Lemma 3.

Next we prove Lemma 1. When violation of 1 is signaled, multiple edge labels with the same equivalence class is updated to different values. The proof can be conducted similarly to the case for Lemma 3, except that we focus on multiple edge labels generated by different subexpressions of the transformation.

$\{\$l : \$g\}$: Suppose bindings $\$l \mapsto \mathtt{a}$ and $\$g \mapsto \{\mathtt{b} : G\}$ are generated by updates on the target graph. Further, suppose these bindings are generated by $\mathbf{rec}(\lambda(\$l', \_).\, \mathbf{rec}(\lambda(\$l, \$g).\{\$l : \$g\})(\{\$l' : \{\$l' : \{\}\}\}))(\$db)$ as the inner body expression of **rec** and $\$db$ is bound to graph $\{\mathtt{x} : \{\}\}$. Then, backward evaluation of the argument expression in the inner **rec**, i.e., $\{\$l' : \{\$l' : \{\}\}\}$ will produce the bindings $\$l' \mapsto \mathtt{a}$ and $\$l' \mapsto \mathtt{b}$ for the first label expression $\$l'$ and the label construction expression $\{\$l' : \{\}\}$, respectively, and merge these bindings by $\uplus$ operator relative to original binding $\$l' \mapsto \mathtt{x}$. Then backward transformation fails because of the conflicting bindings. In this case, the augmented forward semantics also allocate the top and the following edge the same origin edge, so the update signals violation of condition 1. So violation of condition 1 coincides failed backward transformation. Similar situation can be observed for graph constructors $\cup, \oplus$ and $@$ that unifies the graphs, when conflicting updates are attempted to edges originated from identical edges. The merge phase for the subexpressions in the backward transformation causes the failure. Note that the duplicate is propagated through variable bindings, so conflicting updates can be detected within the target graph created by a single graph variable reference, like $\mathbf{let}\ \$g' = \mathbf{rec}(\lambda(\$l', \_).\, \mathbf{rec}(\lambda(\$l, \$g).\{\$l : \$g\})(\{\$l' : \{\$l' : \{\}\}\}))(\$db)\ \mathbf{in}\ \$g'$. The failure takes place in the **let** expression in this case. We omit the detailed case analysis here.

For the Lemma 2, its proof is straightforward by the warning algorithm in the long version. Note that if we exclude the update on target that causes violation of condition 2, we have soundness and completeness. In general, we only have soundness. This concludes the proof sketch. □

6 Related Work

Our novelty is to analyze editability using traces in compositional bidirectional graph transformations.

Tracing mechanism. Traceability is studied enthusiastically in model-driven engineering (MDE) [9, 25]. Van

Amstel et al. proposed a visualization framework TraceVis for chains of ATL model transformations [30]. Systematic augmentation of the trace-generating capability with model transformations [22] is achieved by higher-order model transformations [29]. Although they also trace between source, transformation and target, we also use the trace for editability analysis. We can help improve this kind of unidirectional tool. Grain size of transformation trace in TraceVis is at ATL rule level, while our trace is more fine grained. If TraceVis refines the transformation trace to support inspection of guards (conditions), they can also support control flow change analysis. To do so, ATL engine may be extended to maintain the variable binding environment similar to ours at run-time, and use it to trace bindings within the guards.

Our own previous work [13] 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 [4] for the nested relational calculus.

Triple Graph Grammars (TGG) [26] and frameworks based on them are studied extensively and are applied to MDE [1] including traceability [11]. 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 [10]. The transformation language UnQL is compositional in that the target of a (sub)transformation can be the source of another (sub)transformation, while TGG is not. Our tracing over compositions are achieved by keeping track of variable bindings.

Xiong et al. [32]'s traces are editing operations (including deletion, insertion and replacement) embedded in model elements. They are translated along with transformation (in ATL bytecode). They check control flow changes in these embedded operations *after* finishing backward transformations and reject updates when these changes are detected. ATL level trace could have been computed using our approach for correspondence analysis.

Target Element Classification and Editability Analysis. Model element classification has been studied in the context of Precedence Triple Graph Grammars [23]. Though the motivation is not directly related to ours, their classification is indeed similar in the sense that both group elements in graphs. The difference is that the equivalence class in [23] is defined such that all the elements in the class are affected in a similar manner (e.g., created simultaneously) while in our equivalence class, members are defined only on the target side, and they are originated from the same source edge, so edits on any member result in edits on the source edge. The equivalence class in [23] may correspond to the subgraph created by the same "bulk" by the body expression of **rec** with respect to common edge produced by the argument expression, so that when the common edge is deleted, then all the edges in the bulk are deleted simultaneously.

Another well-studied bidirectional transformation framework called semantic bidirectionalization [31] 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 [24] 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.

Hu et al. [21] treat duplications explicitly by a primitive Dup. They do not highlight duplicates in the view, rather they rely on another forward transformation to complete the propagation among duplicates.

7 Conclusion

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. In this paper we only focused on edge-reaming as edit operation but the proposed

framework can also support edge-deletion by slight extension, while insertion handling can be supported by the present node tracing.

As a future work, we would like to utilize the proposed framework to optimize the performance of edge deletion and subgraph insertion using the backward transformation semantics for in-place update semantics, because we currently handle these update operations by different algorithms. In particular, insertions use a general inversion strategy which is costly. We already have limited support in this direction, however we are still to establish formal bidirectional properties for complex expressions. 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". The proposed mechanism can be used for any other trace-based approaches, unidirectional or bidirectional, as we discussed in Section 6, and we would like to pursue this direction as well.

Acknowledgments The authors would like to thank Zhenjiang Hu, Hiroyuki Kato and other IPL members, and AtlanMod team for their valuable comments. We also thank the reviewers for their constructive feedbacks. The project was supported by the International Internship Program of the National Institute of Informatics.

References

[1] Carsten Amelunxen, Felix Klar, Alexander Königs, Tobias Rötschke, and Andy Schürr. Metamodel-based tool integration with MOFLON. In *ICSE '08*, pages 807–810. ACM, 2008.

[2] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[3] Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal*, 9(1):76–110, 2000.

[4] James Cheney, Umut A. Acar, and Amal Ahmed. Provenance traces. *CoRR*, abs/0812.0564, 2008.

[5] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT'09*, pages 260–283, 2009.

[6] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. From state- to delta-based bidirectional model transformations: The symmetric case. In *MODELS'11*, volume 6981 of *LNCS*, pages 304–318. 2011.

[7] Romina Eramo, Alfonso Pierantonio, and Gianni Rosa. Uncertainty in bidirectional transformations. In *Proc. 6th International Workshop on Modeling in Software Engineering (MiSE)*, pages 37–42. ACM, 2014.

[8] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.

[9] Ismenia Galvao and Arda Goknil. Survey of traceability approaches in model-driven engineering. In *EDOC '07*, pages 313–324. IEEE Computer Society, 2007.

[10] Holger Giese and Robert Wagner. From model transformation to incremental bidirectional model synchronization. *Software & Systems Modeling*, 8(1):21–43, 2008.

[11] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, and Richard F. Paige. Inter-modelling: From theory to practice. In *MODELS'10*, pages 376–391. Springer-Verlag, 2010.

[12] Soichiro Hidaka, Martin Billes, Quang Minh Tran, and Kazutaka Matsuda. Trace-based approach to editability and correspondence analysis for bidirectional graph transformations. Technical report, May 2015. `http://www.prg.nii.ac.jp/projects/gtcontrib/cmpbx/tesem.pdf`.

[13] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. Bidirectionalizing graph transformations. In *ICFP'10*, pages 205–216. ACM, 2010.

[14] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, Keisuke Nakano, and Isao Sasano. Marker-directed optimization of UnCAL graph transformations. In *LOPSTR'11, Revised Selected Papers*, volume 7225 of *LNCS*, pages 123–138, July 2012.

[15] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations (short paper). In *ASE'11*, pages 480–483. IEEE, 2011.

[16] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations. *Progress in Informatics*, (10):131–148, March 2013. Journal version of [15].

[17] Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. Towards compostional approach to model transformations for software development. Technical Report GRACE-TR08-01, GRACE Center, National Institute of Informatics, August 2008.

[18] Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. A compositional approach to bidirectional model transformation. In *ICSE New Ideas and Emerging Results track, ICSE Companion*, pages 235–238. IEEE, 2009.

[19] Soichiro Hidaka, Zhenjiang Hu, Hiroyuki Kato, and Keisuke Nakano. Towards a compositional approach to model transformation for software development. In *Proc. of the 2009 ACM symposium on Applied Computing (SAC)*, pages 468–475. ACM, 2009.

[20] Soichiro Hidaka and James F. Terwilliger. Preface to the third international workshop on bidirectional transformations. In *Workshops of the EDBT/ICDT 2014*, pages 61–62, 2014.

[21] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *PEPM '04*, pages 178–189, 2004.

[22] Frédéric Jouault. Loosely Coupled Traceability for ATL. In *ECMDA Traceability Workshop (ECMDA-TW)*, pages 29–37, 2005.

[23] Marius Lauder, Anthony Anjorin, Gergely Varró, and Andy Schürr. Efficient model synchronization with precedence triple graph grammars. In *ICGT'12*, pages 401–415. Springer-Verlag, 2012.

[24] Kazutaka Matsuda and Meng Wang. "bidirectionalization for free" for monomorphic transformations. *Science of Computer Programming*, 2014. DOI:10.1016/j.scico.2014.07.008.

[25] Richard F. Paige, Nikolaos Drivalos, Dimitrios S. Kolovos, Kiran J. Fernandes, Christopher Power, Goran K. Olsen, and Steffen Zschaler. Rigorous identification and encoding of trace-links in model-driven engineering. *Softw. Syst. Model.*, 10(4):469–487, October 2011.

[26] Andy Schürr. Specification of graph translators with triple graph grammars. In *WG '94*, volume 903 of *LNCS*, pages 151–163, June 1995.

[27] Perdita Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software and System Modeling*, 9(1):7–20, 2010.

[28] Perdita Stevens. Bidirectionally tolerating inconsistency: Partial transformations. In *FASE'14*, volume 8411 of *LNCS*, pages 32–46, 2014.

[29] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the use of higher-order model transformations. In *ECMDA-FA'09*, volume 5562 of *LNCS*, pages 18–33, 2009.

[30] Marcel F. van Amstel, Mark G. J. van den Brand, and Alexander Serebrenik. Traceability visualization in model transformations with TraceVis. In *ICMT'12*, pages 152–159, 2012.

[31] Janis Voigtländer. Bidirectionalization for free! (pearl). In *POPL '09*, pages 165–176. ACM, 2009.

[32] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In *ASE'07*, pages 164–173, November 2007.