*Special issue: Advanced Programming Techniques for Construction of Robust, General and Evolutionary Programs*

Research Paper

# GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations*1

Soichiro HIDAKA[1], Zhenjiang HU[2], Kazuhiro INABA[3], Hiroyuki KATO[4] and Keisuke NAKANO[5]
[1,2,4] *National Institute of Informatics*
[3] *National Institute of Informatics*[*2]
[5] *University of Electro-Communications*

## ABSTRACT

**Bidirectional model transformation is useful for maintaining consistency between two models, and has many potential applications in software development including model synchronization, round-trip engineering, and software evolution. Despite these attractive uses, the lack of a practical tool supporting for systematic development prevents it from being widely used. In this paper, we solve this problem by proposing an integrated framework called GRoundTram (Graph Roundtrip Transformation for Models), which is carefully designed and implemented for compositional development of well-behaved and efficient bidirectional model transformations. GRoundTram is built upon a well-founded bidirectional framework and is equipped with a user-friendly language for coding bidirectional model transformations, a novel tool for validating both models and transformations, an optimization mechanism for improving efficiency, and a powerful debugging environment for testing bidirectional behavior. GRoundTram has been used by other reseach groups besides ourselves and their results show its usefulness in practice.**

## KEYWORDS

Model-driven development, bidirectional transformation, model transformation, graph transformation

## 1 Introduction

Bidirectional model transformation [6], [12], [14], [44], [45] is an enhancement of model transformation with bidirectional capability, and is an important requirement in Object Management Group (OMG)'s Queries/Views/Transformations (QVT) standard for defining model transformation languages. It describes not only a forward transformation from a source model to a target model, but also a backward transformation showing how to reflect the changes in the target model in the source model so that consistency between them is maintained. Bidirectional model transformation has many potential applications in software development, including model synchronization [6], [21], [50], round-trip engineering [5], software evolution [38], and multiple-view software development [19], [22].

Unlike (unidirectional) model transformation where lots of tools have been developed for supporting design, validation, and test of model transformation, bidirectional model transformation lacks such useful tools, and that prevents it from being widely used. In fact, we have new requirements and challenges in the context of bidirectional model transformation.

Most importantly, we should be sure that a bidirectional model transformation behaves exactly as we want. Such a transformation has more complicated be-

Received April 13, 2012; Revised September 30, 2012; Accepted December 10, 2012.
*1 This paper is an extended version of the conference short paper [30] presented in ASE'11.
*2 Current affiliation is Google.
1) *hidaka@nii.ac.jp*, 2) *hu@nii.ac.jp*, 3) *kinaba@nii.ac.jp*, 4) *kato@nii.ac.jp*, 5) *ksk@cs.uec.ac.jp*
DOI: 10.2201/NiiPi.2013.10.7

havior than a unidirectional one. It should be *well-behaved* in the sense that both the forward and backward transformations are consistent with each other and have the roundtrip property [12]. As argued in [43] though, there are semantic issues with many of the existing tools.

Next, bidirectional model transformations should be *compositional* so that they can reuse existing transformations and bigger ones can be constructed from smaller ones. As indicated in the conclusion of [15], most model transformation languages based on graph transformations are rule-based, describing direct relationships between the source and target models. They are not compositional in the sense that we cannot introduce *intermediate models* for gluing together model transformations. Therefore, rule-based techniques cannot easily support systematic development of model transformations in the large [36]. However, composition comes at the cost of efficiency; many unnecessary intermediate models might be produced. Therefore, an optimization method is required to automatically eliminate unnecessary intermediate models during execution.

Furthermore, a bidirectional model transformation should be general enough because it is to be used at various stages of the software development life cycle. It should be able to be applied to different models such as UML diagrams, sequence diagrams, Petri-nets, and even lower level control/data flow graphs. While visual frameworks are useful in high-level design, *general text-based languages* play an important role in developing large-scale transformations, say, to deal with lower level mappings or complex code refactoring. Moreover, we would expect to have a set of language-based tools for type checking (validating) both models and bidirectional model transformations to remove errors before execution, an efficient execution model, and a tool for testing/debugging bidirectional behavior. Apart from a recent attempt in [49] that embed lens [18] as a DSL in Scala, which benefits from the type system in the host language, as far as we are aware, no such *language-based modeling environments* have been proposed for bidirectional model transformation.

In this paper, we remedy this situation by proposing a language-based modeling framework called GRoundTram (Graph Roundtrip Transformation for Models), which is carefully designed and implemented for *compositional* development of *well-behaved* and *efficient* bidirectional model transformation at the various stages of software development. Our work is inspired by recent research on bidirectional languages (with well-behavedness) and automatic bidirectionalization in the programming language community [9], [18], [32], [39]. In particular, it has been recently shown [27] that a graph query algebra UnCAL (Unstructured CALculus) [10], which consists of carefully designed graph constructors, conditional and structural recursion operators, can be fully bidirectionalized. Each graph transformation in UnCAL has clear bidirectional semantics and is guaranteed to be well-behaved.

This paper is about a successful application of a bidirectional graph query algebra in the programming community to the construction of a framework for developing bidirectional model transformations in the software engineering community. Our main technical contributions are summarized as follows.

**Well-behavedness.** We propose a novel *bidirectional graph contraction algorithm* so that we can build well-behaved bidirectional model transformations upon the well-founded bidirectional UnCAL algebra. In fact, there is a gap between UnCAL graphs and the models used in model transformation: graphs in UnCAL are edge-labeled and their equivalence is defined by bisimilarity, while models in model transformation may have labels on both edges and nodes and their equivalence is defined by unique identifiers. We close this gap so that every UnCAL graph has a bidirectional correspondence with a model.

**Compositional.** We design a user-friendly language UnQL$^+$, which is the first *purely functional language* for developing large bidirectional model transformations in a *compositional* way. UnQL$^+$ is an extension of the graph query language UnQL [10] with new additional language constructs for graph transformation. We show that any UnQL$^+$ program can be correctly translated into an UnCAL construct and inefficiency due to intermediate models in the composition can be automatically eliminated.

**Language-based IDE.** We implement an integrated development environment GRoundTram, which has a novel tool for validating both models and bidirectional model transformations, an automatic optimization mechanism for improving efficiency, and a powerful debugging environment for testing bidirectional behavior. The system (including sources, documents, and many application examples) is available online [2], and being used by research groups besides ourselves for developing some nontrivial applications. Their successes indicate the usefulness of GRoundTram in practice.

This paper is an enhancement of the emerging idea of our previous work that was presented in a short paper [26]. Moreover, in relation to the bidirectionalization presented in [27], a bidirectional contraction stage has been added after stage 2 (epsilon edge and un-

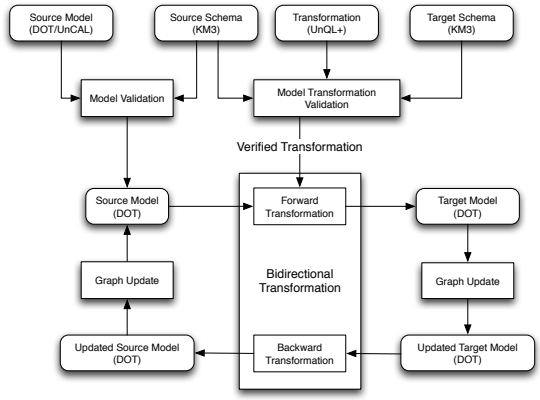



Fig. 1 Overview of GRoundTram.

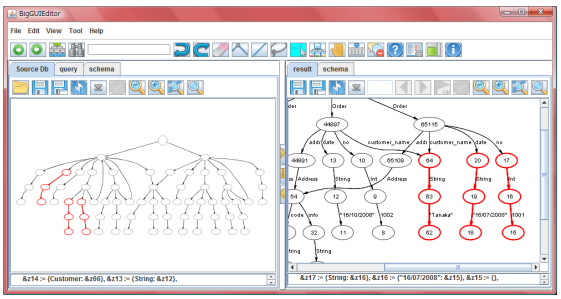


Fig. 2 Snapshot of GRoundTram.

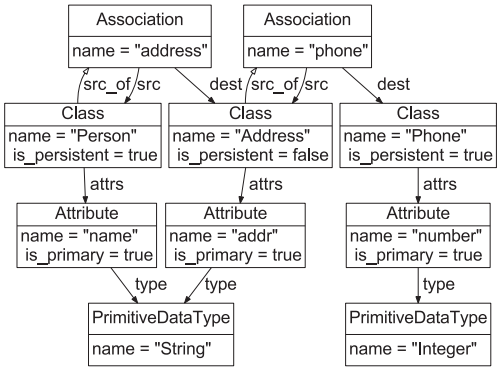



Fig. 3 A class diagram.

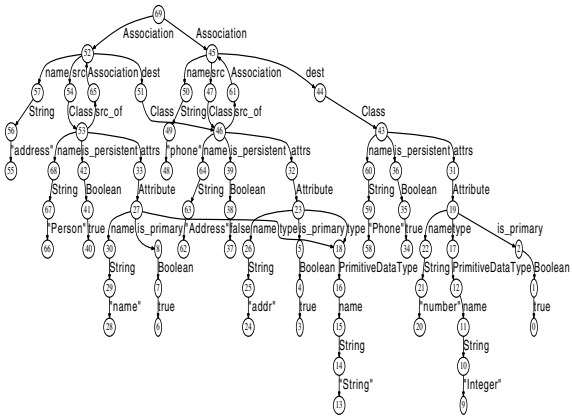



Fig. 4 A class model represented by an edge-labeled graph.

reachable part elimination stage) of the previous work. With respect to the surface syntax extension presented in [25], the present paper can deal with regular path expressions and treat multiple graph databases. We have also added a proof of correctness of a translation to an internal graph algebra and have integrated the verification framework presented in [33] into ours.

The rest of the paper is organized as follows. We begin by demonstrating how the GRoundTram system works in Section 2. We then briefly review the UnCAL bidirectional framework on which GRoundTram is based in Section 3. After explaining the design architecture of the GRoundTram system, we show in detail the definition of UnQL$^+$, the bidirectional graph contraction algorithm, and the translation from UnQL$^+$ to UnCAL in Section 4. We evaluate the system in Section 5, discuss the related work in Section 6, and conclude the paper in Section 7.

## 2 Overview of GRoundTram

Before proceeding with the technical details, let us overview the GRoundTram system to give the reader an idea of what it can do. Figure 1 shows the basic functions the GRoundTram system provides.

### 2.1 Input

The input to the system is a source model together with its schema, a transformation described in UnQL$^+$, and a target model schema. The target model is produced by the forward transformation.

**Model.** Models are represented by general edge-labeled graphs, which form a general representation of various models. As a running example, consider the class model diagram in Figure 3, which is taken from [15]. It consists of three classes and two directed associations, and each class has a primary attribute[*3]. This model can be represented by the graph in Figure 4, where the information has been moved to the edges. The graph is in the standard DOT format which can be

[*3] For this particular example, the diagram could be further simplified by using simple labeled references instead of model elements for phone number, for example, and nothing prevents us from doing so. However, we believe this example helps us demonstrate the expressiveness of transformations in GRoundTram.

visualized and edited by the popular Graphviz tool [16].

**Model schema (metamodel).** Each model has a structure. For instance, a class diagram has the following structure. A class diagram consists of classes and directed associations between classes. A class is indicated as persistent or non-persistent. It consists of one or more attributes, at least one of which must be marked as constituting the classes' primary key. An attribute type is of a primitive data type (e.g. String, Integer). An association associates classes and is represented here using a model element. KM3 [35] is used to describe such a model structure [25], and its definition can be found in [2]. We currently do not support complex OCL constraints in the schema.

**Model transformation.** (Forward) Model transformation is described compositionally in UnQL$^+$ (Section 4.1), a SQL-like graph query/transformation language. As an example, consider extracting all persistent classes from the class model $*db*, and transforming them into tables by replacing *Attributes* with *Columns*. This can be described compositionally as follows, using the intermediate model $*persistentClass*.

```
select {tables : $table} where
  $persistentClass in
    (* select classes *)
    (select $class where
      {Association.(src|dest).Class : $class} in $db,
      {is_persistent : {Boolean : true}} in $class),
  $table in
    (* replace Attribute *)
    (replace attrs → $g
      by (select {Column : $a} where
        {attrs.Attribute:$a} in $persistentClass)
      in $persistentClass)
```

## 2.2 Validation

In order to detect errors during development as early as possible and help users to develop correct models and transformations, GRoundTram provides two validation mechanisms.

**Model validation.** The system can verify the conformance of the source and the target model to their associated schemata. In particular, after editing the models, it is important to check that they are in valid states.

**Model transformation validation.** Correct model transformations should always generate a target model conforming to the target schema from any source model satisfying the source schema.

While the model validation is standard, a general model transformation validation is more challenging but more useful for ensuring correct model transformations. As an instance of simple erroneous transformation, suppose the user made an error writing **select** $*a* instead of **select** {*Column* : $*a*} in the previous example. Its outputs do not conform to the schema and hence an error is reported by the system. The check is *automatic* and *static*. Users neither have to provide any test cases by hand, nor execute the transformation for testing; the system automatically finds and displays an example of a source model that reveals the problem (in this case, a class model containing at least one persistent class).

## 2.3 Bidirectional transformation

The GRoundTram system is unique in its execution of well-behaved bidirectional transformations, as seen in the lower part of Figure 1.

**Forward transformation.** After the user specified the source model and the UnQL$^+$ model transformation, the target model is computed by running the transformation with the source model set to the variable $*db*. Like the source model, the target model can also be exported and be edited in the standard DOT format.

**Backward transformation.** The most distinct feature of GRoundTram is the automatic derivation of backward transformations that appropriately propagate modifications on the target models to the source models. There is no need to maintain two separate transformations or to worry about their consistency. Users just write a forward transformation from one model to another in a compositional way, and a corresponding backward transformation is automatically derived.

## 2.4 Graphical user interface

The GRoundTram system combines all its functions as an integrated framework with a user-friendly GUI (Figure 2). The user loads a source graph (displayed in the left pane) and a bidirectional transformation written in UnQL$^+$. Once they are loaded, the forward transformation can be conducted by pushing the "forward" button (right arrow icon). The target graph appears on the right pane. The user can graphically edit the target graph and apply a backward transformation by pushing the "backward" button (left arrow icon). The source graph can be edited as well, of course. The user can optionally specify the source schema and the target schema and can run the validation by pushing the check button on both panes. The transformation itself can also be checked.

For ease of debugging/understanding the behavior of the bidirectional computation between two models, *trace information* is instantly displayed between the source and target (red part in Figure 2). If subgraphs

on either pane are selected, corresponding subgraphs on the other pane are also highlighted. This helps users to understand how a modification on the target affects that on the source, and vice versa.

## 3 Background: Bidirectional UnCAL

The GRoundTram system is built upon our recent work [27] on bidirectionalization of UnCAL, a graph algebra known in the database community for graph querying [10]. It has been shown that any unidirectional graph transformation written in UnCAL can be fully bidirectionalized with a backward transformation such that both forward and backward transformations are consistent and well-behaved. We briefly explain the basic results that will be used in this paper.

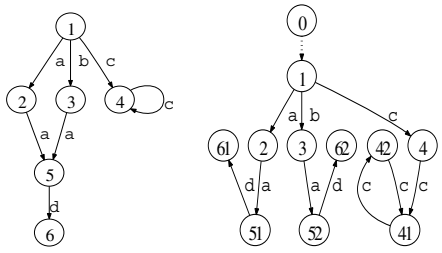
### 3.1 Graph data model

Graphs in UnCAL are rooted and directed cyclic graphs with no order between outgoing edges. They are edge-labeled in the sense that all information is stored as labels on edges and labels on nodes serve as unique identifiers and have no particular meaning. Figure 5(a) gives a small example of a directed cyclic graph with six nodes and seven edges. In text, it is represented by

$$\begin{array}{lcl} g & = & \{\mathsf{a} : \{\mathsf{a} : g_1\}, \mathsf{b} : \{\mathsf{a} : g_1\}, \mathsf{c} : g_2\} \\ g_1 & = & \{\mathsf{d} : \{\}\} \\ g_2 & = & \{\mathsf{c} : g_2\} \end{array}$$

where the notation $\{l_1 : g_1, \ldots, l_n : g_n\}$ denotes a set representing a graph which contains $n$ edges with labels $l_1, \ldots, l_n$, each edge pointing to a graph $g_i$, and the empty set $\{\}$ denotes a graph with a single node. Two graphs $g_1$ and $g_2$ can be merged using the set union operation $g_1 \cup g_2$. In addition, the $\varepsilon$-edge is allowed to represent a shortcut between two nodes, and works like the $\varepsilon$-transition used in automata.

Two graphs in UnCAL are considered to be equal if they are *bisimilar*. An intuitive understanding of bisimilarity is that unfolding of cycles and duplication of equivalent subgraphs do not affect the equivalence of graphs, and unreachable parts from the root are ignored. For instance, the graph in Figure 5(b) is equivalent to the graph in Figure 5(a); the new graph has an additional $\varepsilon$-edge (denoted by the dotted line), duplicates the graph rooted at node 5, and unfolds and splits the cycle at node 4.




(a) A simple graph (b) An equivalent graph

Fig. 5 Graph equivalence based on bisimulation.

It is worth noting that bisimilarity plays an important role in bidirectionalization [27], query optimization [10], and verification of graph transformations [33]. However, bisimilarity is different from the usual equivalence of models whose elements have unique identifiers. We will show how to bridge this gap in Section 4.2.

### 3.2 UnCAL

The most important feature of UnCAL is that any graph transformation in UnCAL is described by structural recursions or their composition.

A structural recursive function $f$ in UnCAL is a recursive computation scheme on graphs defined by

$$\begin{array}{lcl} f\ (\{\}) & = & \{\} \\ f\ (\{l : g\}) & = & (l, g) \odot f(g) \\ f\ (g_1 \cup g_2) & = & f(g_1) \cup f(g_2) \end{array}$$

where $\odot$ is a given binary operator. Different choices of $\odot$ define different recursive functions. For simplicity, the definition above is abbreviated to

$$\textbf{sfun}\ f\ (\{l : g\}) = (l, g) \odot f(g).$$

Note that even for a graph $g$ having cycles, the computation of $f(g)$ always terminates under the usual *recursive semantics*, where all recursive calls are memoized and their results are reused to avoid entering infinite loops.

As a simple example, we may use the following recursive function *a2d_xc* to replace all edges labeled a by d and skip all edges labeled c for the graph in Figure 5(a).

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

We can naturally extend the structural recursion above so that it allows mutual recursion. Any number of mutually recursive functions can be merged into one by using the standard tupling method [31].

### 3.3 Bidirectional semantics of UnCAL

A query in UnCAL is usually run in the forward direction: under an environment (a mapping from variables to graphs) $\rho$, a query $Q$ generates a result graph denoted by $\mathcal{F}[\![Q]\!]\rho$.

Let $g = \mathcal{F}[\![Q]\!]\rho$ be a result graph. Assume that a user has edited it into $g'$. For example, one may add

a new subgraph, modify some labels, or delete several edges. In our previous study [27], we gave *backward semantics* that properly reflect back the editing to the original inputs. Formally speaking, given the modified result graph $g'$ and the original input environment $\rho$, we presented a method that computes the modified environment $\rho' = \mathcal{B}[\![Q]\!](\rho, g')$.

By "properly reflecting back" (or *well-behaved*), we mean the following two properties hold:

$$\frac{\mathcal{F}[\![Q]\!]\rho = g}{\mathcal{B}[\![Q]\!](\rho, g) = \rho}(\textsc{GetPut})$$

$$\frac{\mathcal{B}[\![Q]\!](\rho, g') = \rho'}{\mathcal{B}[\![Q]\!](\rho, \mathcal{F}[\![Q]\!]\rho') = \rho'}(\textsc{WPutGet})$$

The (GetPut) property says that if no change is made to the output $g$, there should be no change in the input environment. The (WPutGet) property is an unrestricted version of the (PutGet) property that appeared in [17], which requires $g' \in \text{Range}(\mathcal{F}[\![Q]\!])$ and $\mathcal{B}[\![Q]\!](\rho, g') = \rho'$ to imply $\mathcal{F}[\![Q]\!]\rho' = g'$. The (PutGet) property states that if a result graph is modified to $g'$, which is in the range of the forward evaluation, this modification can be reflected in the source such that a forward evaluation will produce the same result $g'$. In contrast, the (WPutGet) property allows the modified result to be different (this difference is sometimes called the *view side-effect*) from the result obtained by a backward evaluation followed by a forward evaluation, but requires both results to have the same effect on the original source if the backward evaluation is applied again.

Although the (WPutGet) property is weaker than the (PutGet) property that forbids view side-effect, this property enables us to make flexible modifications on the result graphs. For example, if the transformation includes a duplication, the target will include two copies of the same data. If a user, being unaware of the duplicates, edits only one of them, then this modification would be forbidden in the (PutGet) setting, because updating only one of the copy will make the updated target out of the range of the transformation. Instead, we reflect the updates and the user has a chance to do another forward transformation to see the update reflected in another copy in the view.

Even with the flexibility of (WPutGet) explained above, we reject updating of values that come from transformation, like label `d` in *a2d_xc* described earlier. Because whatever reflection was made, another forward transformation will always produce `d` again, so the user's modification in the target would not have been preserved.

Note that these properties are true under bisimulation equivalence. We additionally need encoding in order to represent models whose equivalence is based on isomorphism. In our earlier study in [42], we fill this gap by encoding identifiers of the model elements with dedicated edges. In this way, no distinct value-equivalent subgraph encoding different model elements will be contracted.

We use three different editing reflection mechanisms [27] for different editing operations: edge renaming, edge deletion, and subgraph insertion. Trace information is used to reflect edge deletion and determine the insertion point in the source. Insertion is handled by using a general inversion technique [3] to enumerate the pairs of the updated target and the corresponding source. Unspecified edge label (corresponding to default values) in the source should be filled in by users.

Although we have an SQL-like **select/replace/delete-where** surface syntax (described in section 4.1), bidirectional interpretation of the transformation takes place at the UnCAL level. Therefore, although we can encode join operations by using consecutive **where** clauses, this join operation is not our unit of bidirectionalization (this join operation is translated to nested structural recursions which are units of bidirectionalization), so we can not exploit these high level semantics to reflect changes like relational lenses [8]. Diskin et al. [13] pointed out a similar composability issue in the state-based setting under the context of keys. We do not use the concept of a key, so we do not have this problem. We do not have a control over update policy using the notion of key.

Hermann et al. [24] discuss the correctness of the synchronization algorithm for TGGs. Compared with this related work, our well-behavedness (correctness) reasoning is only at the graph level of representation. We could have better notion of correctness for users at model level as in [24]. These related works are important towards our reasoning about correctness in the model level.

## 4 Design and implementation of GRoundTram

Figure 6 depicts the architecture of GRoundTram. We provide a new user-friendly model transformation language UnQL$^+$ that is *functional* (rather than rule-based as in many existing tools) and *compositional* with high modularity for reuse and maintenance, and the architecture handles models that are described by edge-labeled graphs that are general enough to express various models. GRoundTram system runs on the powerful engine of bidirectional UnCAL, which has a set of language-based tools: a bidirectional interpreter [27], a graph and graph transformation verifier [33], an optimizer to improve efficiency [28], and a checker of valid updates in the backward transformation [40]. The key

contributions in this implementation are (1) a translation of UnQL$^+$ into UnCAL to enable the engine of bidirectional UnCAL to execute UnQL$^+$ bidirectionally, and (2) a bidirectional graph contraction algorithm for contracting bisimilar UnCAL graphs so that an ordinary model will have a bidirectional correspondence with an UnCAL graph.

In the rest of this section, we will focus on the exploration of UnQL$^+$, the bidirectional graph contraction algorithm, and the translation from UnQL$^+$ to UnCAL.

### 4.1 Model transformation in UnQL$^+$

UnQL$^+$ is the language the GRoundTram system provides for users to describe (bidirectional) model transformations. It is an extension of the well-known UnQL [10], a graph querying language, which is compositional and can be implemented by FO (TC) (first order with transitive closure) with PTIME time complexity for graph querying.

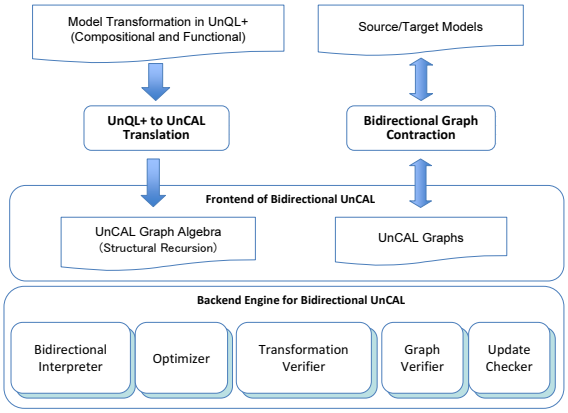



Fig. 6 GRoundTram implementation on bidirectional UnCAL engine.

Figure 7 gives the core syntax of UnQL$^+$. A graph transformation is described by a template expression to construct a graph from graphs that are bound by graph variables. The expression $\{l_1 : t_1, \ldots, l_n : t_n\}$ creates a new node having $n$ outgoing edges labeled $l_i$ and pointing to the root of the graph computed from $t_i$. The union $g_1 \cup g_2$ constructs a graph with a root sharing the roots of $g_1$ and $g_2$. The variable expression $\$g$ returns the graph that is bound by $\$g$ in the environment (i.e., a mapping from variables to graphs). The conditional expression has the usual meaning, i.e., choosing different branch according to the (binding) condition $B$.

Like other query languages, UnQL$^+$ has a convenient template expression **select** $t$ **where** $bs$, which is used to select the subgraphs satisfying the a sequence of conditions $bs$, binds them to variables, and construct a result according to the template expression $t$. For instance, the following query extracts all persistent classes from the class model in Figure 4, which is assumed to be bound by $\$db$.

**select** $\$class$ **where**
$\{Association.(src|dest).Class : \$class\}$ **in** $\$db$,
$\{is\_persistent : \{Boolean : true\}\}$ **in** $\$class$

This query returns all bindings of the variable $\$class$ satisfying the two conditions in the *where* clause. The first condition is to find bindings of $\$class$ by matching the *regular path pattern* $Association.(src|dest).Class$ with the graph bound by $\$db$, while the second condition is to ensure that the class is persistent.

In model transformations, one often wants to replace a subgraph satisfying a certain condition by another graph, and it is onerous to describe such transformations using *select-where* because some context structure must be copied and propagated. To this end, we introduce three new template expressions, namely, *replace-*

| | | | |
|---|---|---|---|
| (template) | $T$ | ::= | $\{L : T, \ldots, L : T\} \mid T \cup T \mid \$g$ |
| | | | \| **if** $BC$ **then** $T$ **else** $T$ |
| | | | \| **select** $T$ **where** $B, \ldots, B$ |
| | | | \| **replace** $Rp \to \$G$ **by** $T$ **in** $T$ **where** $B, \ldots, B$ |
| | | | \| **extend** $Rp \to \$G$ **with** $T$ **in** $T$ **where** $B, \ldots, B$ |
| | | | \| **delete** $Rp \to \$G$ **in** $T$ **where** $B, \ldots, B$ |
| | | | \| **let sfun** $fname(L : \$G) = \ldots$ **in** $fname(T)$ |
| (binding) | $B$ | ::= | $Gp$ **in** $\$G \mid BC$ |
| (condition) | $BC$ | ::= | not $BC \mid BC$ and $BC \mid BC$ or $BC$ |
| | | | \| isEmpty($\$G$) $\mid L = L \mid L \neq L \mid L < L \mid L \leq L$ |
| (label) | $L$ | ::= | $\$l \mid$ a |
| (label pattern) | $Lp$ | ::= | $\$l \mid Rp$ |
| (graph pattern) | $Gp$ | ::= | $\$G \mid \{Lp : Gp, \ldots, Lp : Gp\}$ |
| (regular path pattern) | $Rp$ | ::= | a $\mid$ _ $\mid Rp.Rp \mid (Rp\|Rp) \mid Rp? \mid Rp* \mid Rp+$ |

Fig. 7 Syntax of UnQL$^+$.

*where*, *extend-where*, and *delete-where*.

- The *replace-where* expression replaces a subgraph with a new graph. For the following *replace-where* expression,

  **replace** $r \to \$v$ **by** $e_1$ **in** $e_2$ **where** $b_1, \ldots, b_n$

  the semantics of this expression are that, starting from the root node of $e_2$, it traverses every path and replaces the node $\$v$, which is on each path that matches $r$ and satisfies $b_1 ... b_n$, by $e_1$. Consider the class model again, prefixing every name of the class by "class_" can be done as follows. Note that "^" is a built-in function for string concatenation.

  **replace** `_*.Class.name.string` $\to \$u$
  **by** {("`class_`"^$\$name$) : {}} **in** $\$db$
  **where** {$\$name$ : {}} **in** $\$u$

- The *delete-where* expression is used to describe the deletion of part of the graph.

  For the following *delete-where* expression,

  **delete** $r \to \$v$ **in** $e$ **where** $bs$

  the semantics of this expression are that, starting from the root node of $e$, it traverses every path and deletes the node $\$v$, which is on each path that matches $r$ and satisfies $bs$. For instance, we may eliminate all persistent classes by

  **delete** `Association.(src|dest).Class` $\to \$c$
  **in** $\$db$
  **where** {`is_persistent.Boolean` : *true*}
  **in** $\$c$

  where the subgraph matching $\$c$ will be deleted from its original graph $\$db$.

- The *extend-where* expression describes the extension of a graph with another graph.

  For the following *extend-where* expression,

  **extend** $r \to \$v$ **with** $e_1$ **in** $e_2$ **where** $bs$

  the semantics of this expression are that, starting from the root node of $e_2$, it traverses every path and extends the node $\$v$, which is on each path that matches $r$ and satisfies $bs$, with $e_1$. For example, we write the following transformation to add `date` information to each `class`.

  **extend** `_*.class` $\to \$c$
  **with** {`date` : "`2008/8/4`"}
  **in** $\$db$

These three new template expressions can be automatically translated to structural recursions in UnCAL (see Section 4.3).

Unlike most rule-based model transformation languages, where model transformation composition is not straightforwardly supported [15], UnQL$^+$ is functional and compositional; smaller model transformations can be composed to form bigger ones (see Section 2 and Section 5).

### 4.2 Bidirectional graph contraction

As explained in Section 3, our graph model is based on bisimulation equivalence, which means bisimilar graphs cannot be distinguished. Moreover, since UnCAL is based on bisimulation, a transformation may introduce redundant nodes that are bisimilar to each other. Therefore, after a transformation such redundancy has to be eliminated in a normalization phase.

Fortunately, for any set of graphs that are bisimilar to each other, there exists a unique normal form up to isomorphism and we can obtain the normal form after a transformation by using the partition refinement algorithm of Paige and Tarjan [41]. The algorithm's complexity is $O(|E| \log |V|)$, where $|E|$ and $|V|$ are the numbers of edges and nodes, respectively, and we consider that this level of complexity would be acceptable in practice. Although this algorithm works on node-labeled graphs, we lift the algorithm to our edge-labeled graph model by converting edges labeled by $l$ into two unlabeled edges and a node between them labeled by $l$, as described in [10]. Any bisimilar subgraphs are then contracted to one subgraph, in which no pairs of nodes are bisimilar to each other. In particular, leaf nodes (nodes that have no outgoing edges) are bisimilar to each other, so they all shrink to one node.

We carefully design our contraction algorithm so that it forms a well-behaved bidirectional transformation that has the (GETPUT) and (WPUTGET) properties explained in Section 3.3, in the sense that no modification on the contracted graph results in no modification on the uncontracted graph, while the modified uncontracted graph can be obtained again after re-uncontracting the graph obtained by contracting the modified uncontracted graph. For example, suppose the set of nodes $V_1 = \{v_{1_1}, \ldots, v_{1_M}\}$ are contracted to $v_1$, and the set of nodes $V_2 = \{v_{2_1}, \ldots, v_{2_N}\}$ are contracted to $v_2$. Further, suppose an edge $(v_1, l, v_2)$ is inserted in the contracted graph. As a result, edges labeled $l$ are inserted between $V_1$ and $V_2$ in the uncontracted graph. If $M > 1$ and $N > 1$, then edges labeled $l$ are inserted only between pairs of nodes that were originally connected in the uncontracted graph, although all-to-all connections from $V_1$ to $V_2$ are also well-behaved. If no pair of nodes were originally connected, then the above all-to-

all connection is used.

It is worth noting that (WPUTGET) law (instead of PURGET) here is not caused by the duplication of forward transformation. Violation of (PUTGET) law occurrs when the modification of the target makes non-bisimilar nodes bisimilar. For example, suppose source graph {a:{b:{}}, a:{c:{}}}. Contraction will produce {a:{b:{}}, a:{c:{}}}. And suppose the label c is modified to b in the contracted graph. Then backward transformation will produce {a:{b:{}}, {b:{}}}. Next foward transformation (contraction) will produce {a:{b:{}}}, which is not isomorphic to previous target {a:{b:{}}, {b:{}}}, although they are bisimilar. Since contraction transforms bisimilar graphs to its normal form up to isomorphism, the non-isomorphic targets should be considered different, so only (WPUTGET) law is satisfied.

### 4.3 Translating UnQL$^+$ to UnCAL

UnQL$^+$ is different from UnCAL in that it uses four important template expressions, namely *select, replace, extend*, and *delete*, to describe graph transformations rather than using structural recursion. In this section, we show that all these template expressions can be translated into structural recursions in UnCAL.

The *select* expression, which is inherited from UnQL, can be translated into structural recursion in [10] (the explanation is omitted). According to the semantics described in Section 4.1, the *delete* and *extend* expressions can be defined in terms of the *replace* expression as:

$$\textbf{delete } Rp \to \$v \textbf{ in } e_1 \textbf{ where } bs \\ \Rightarrow \textbf{ replace } Rp \to \$v \textbf{ by } \{\} \textbf{ in } e_1 \textbf{ where } bs$$

$$\textbf{extend } Rp \to \$v \textbf{ with } e_1 \textbf{ in } e_2 \textbf{ where } bs \\ \Rightarrow \textbf{ replace } Rp \to \$v \textbf{ by } \$v \cup e_1 \textbf{ in } e_2 \\ \textbf{where } bs$$

Therefore, what we need to show is how the *replace* expression is translated into a structural recursion.

Our idea for this translation is to use structural recursion to simulate the behavior of a deterministic finite automaton (DFA) for finding the nodes in the graph where the replace operation is to be applied. For the *select* expression in UnQL, NFA is used to define the structural recursion for finding the nodes in graph to be selected [10]. The reason we use a DFA instead of an NFA is to keep the context correct. The detailed explanation is provided at the end of this section.

Now, consider the following general form of the *replace* expression.

$$\textbf{replace } Rp \to \$v \textbf{ by } e_1 \textbf{ in } e_2 \textbf{ where } bs$$

First, we translate the regular path pattern $Rp$ into a DFA $(Q, \Sigma_{\$l}, \delta, q_0, F)$, where $Q = \{q_0, \ldots, q_N\}$ is a finite set of states, $\Sigma_{\$l} = \Sigma \cup \{\$l\}$ (where $\Sigma = \{l_0, \ldots, l_K\}$) is a finite set of labels used in $Rp$, $\delta : Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is the start state, and $F \subseteq Q$ is a set of accept states. We use the special label $\$l$ to denote a label other than those used in $Rp$.

Next, we introduce $N + 1$ functions $h_{q_0}, \ldots, h_{q_N}$, where $h_{q_i}$ corresponds to state $q_i$, and define each $h_{q_i}$ as a structural recursion in the following way. For each label $l \in \Sigma_{\$l}$, we define

$$h_{q_i}(\{l : \$v\}) \quad = \quad e_{ij}$$

and construct a graph with $e_{ij}$ by considering two cases. Note that $j$ ranges over transitions from $q_i$. If $\delta(q_i, l) \notin F$ (i.e., transition from $q_i$ through $l$ does not reach to an accept state), we keep the context by propagating $l$ and continuing the recursive computation by defining

$$e_{ij} = \{l : h_{\delta(q_i,l)}(\$v)\}.$$

Otherwise, we check whether $\$v$ satisfies the condition $bs$. If it does, we replace the graph with the query result of $e_1$ satisfying $bs$:

$$e_{ij} \quad = \quad \textbf{if } \mathsf{isEmpty}(\textbf{select } \{\text{“found”}\} \textbf{ where } bs) \\ \textbf{then } \{l : h_{\delta(q_i,l)}(\$v)\} \\ \textbf{else } \{l : (\textbf{select } e_1 \textbf{ where } bs)\}.$$

The condition isEmpty(...) in the **if**-expression checks whether the condition $bs$ holds. Note that since $e_1$ might be evaluated to {}, the checking expression should not be **select** $e_1$ **where** $bs$.

By applying the function associated with the initial state to $e_2$, we get a UnQL expression having both structural recursions and *select* expressions. Finally, since the *select* expressions can be translated into structural recursions by using the existing method, we can get structural recursions in UnCAL.

**Example 1.** Our algorithm maps the *replace* expression shown in Figure 8(a) to the structural recursion in Figure 8(c) via the DFA obtained from _*.(a|c) in Figure 8(b).

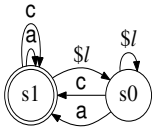
This example also demonstrates the user-friendliness of the **replace** syntax, since without this extension, we have to directly code the mutually recursive function in Figure 8(c), after manually constructing the corresponding DFA in Figure 8(b). This manual coding would be error-prone and much more verbose than the **replace** syntax.

Now, let us show the correctness of the above translation from UnQL$^+$ to UnCAL. As described in Section 4.1, we interpret the expression

$$\textbf{replace } r \to \$v \textbf{ by } e_1 \textbf{ in } e_2 \textbf{ where } bs$$

```
replace _ * .(a|c)− > $v
by      $v′
in      $db
where   {g : $u} in $v,
        {_ * .d : $v′} in $v
```

(a) A replace expression

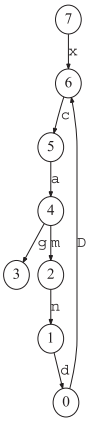



(b) DFA for _ * .(a|c)

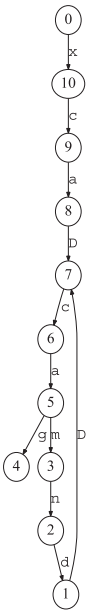
```
let sfun hs0({a : $v}) = if isEmpty(e1)
                         then {a : hs1($v)}
                         else {a : e2}
    |   hs0({c : $v}) = if isEmpty(e1)
                         then {c : hs1($v)}
                         else {c : e2}
    |   hs0({$l : $v}) = {$l : hs0($v)}
  sfun hs1({a : $v}) = if isEmpty(e1)
                         then {a : hs1($v)}
                         else {a : e2}
    |   hs1({c : $v}) = if isEmpty(e1)
                         then {c : hs1($v)}
                         else {c : e2}
    |   hs1({$l : $v}) = {$l : hs0($v)}
in hs0($db)
where e1 ≡ select {"found" : {}}
           where ({g : $u} in $v), ({_ * .d : $v′} in $v)
      e2 ≡ select $v′
           where ({g : $u} in $v), ({_ * .d : $v′} in $v)
```

(c) Translated structural recursion

Fig. 8 A translation example.




(a) A source graph




(b) A target graph

Fig. 9 Source graph and and its transformation by replace expression in Figure 8(a).

so that, starting from the root node of $e_2$, it traverses every path and replaces the node $\$v$, which is on each path that simultaneously matches $r$ and satisfies $bs$, by $e_1$ (actually **select** $e_1$ **where** $bs$ should be used instead of $e_1$ to obtain the bindings from $bs$). More formally, it is intended to work equivalently to the following pseudo code.

```
let sfun f({$l : $v}) =
        if MATCH[r, bs]($v)
        then {$l : select e1 where bs}
        else {$l : f($v)}
in f(e2)
```

where MATCH$[r, bs]$ is a pseudo predicate that is evaluated to be true if and only if $\$v$ is the first node on the path from the root that matches $r$ and satisfies $bs$. All branches in our desugaring have either the form {$\$l$ : **select** $e_1$ **where** $bs$} or {$\$l$ : $f(\$v)$}. Thus, what we have to prove is that the MATCH$[r, bs]$ pseudo predicate is correctly encoded in the mutual recursion of structural recursive functions. First, whether the node $\$v$ matches $r$ is encoded in the mutual recursion, which simulates the behavior of the deterministic finite automaton (DFA). This is a standard technique to represent regular patterns. Next, whether the node $\$v$ satisfies $bs$ is encoded in a **if**-expression having isEmpty(**select** {"found"} **where** $bs$) as its condition. This condition holds if and only if $bs$ does not hold, since isEmpty($e$) holds if and only if $e$ is evaluated into {}. Thus, this **if**-expression results in **select** $e_1$ **where** $bs$ when $bs$ holds. Otherwise, it results in keeping the input label as $\$l$, together with calling the function associated with the next state in the DFA. Therefore, the MATCH$[r, bs]$ pseudo predicate is correctly encoded in our desugaring code.

**Example 2.** Figure 9(b) shows the result of using the **replace** expression in Figure 8(a) to transform the cyclic graph in Figure 9(a). This example illustrates what happens if a pattern matches the middle of a cycle. Since the replace expression captures the first match along the path, node 4 in Figure 9(a) matches for replacement. Note that DFA in Figure 8(b) only corresponds to a path in the **replace** clause, but the **where** clause also works to specify the matching node. Therefore, even though node 5 also matches by virtue of the DFA, it does not actually match since it does not satisfy the **where** clause. The obtained binding of $\$v'$ is a subgraph under node 0, bound by the regular path pattern _*.$d$ The subgraph under node 8 in Figure 9(b) is a copy of the subgraph under node 0. The entire corre-

spondence of the source nodes to target nodes is

$7 \to \{0\}$
$6 \to \{10, 7\}$
$5 \to \{9, 6\}$
$4 \to \{5\}$
$3 \to \{4\}$
$2 \to \{3\}$
$1 \to \{2\}$
$0 \to \{8, 1\}$

The 'first match' semantics correspond to a translation algorithm which uses **select** to generate subgraphs in the **by** clause, since the **select** construct itself does not conduct recursive replacement as **replace** can.

#### Necessity of DFA in the translation of UnQL$^+$

We give a detailed explanation of why DFA is necessary for translation of **replace/delete/extend** constructs.

Automata created for **replace/delete/extend** play a role that is different from those created for **select**.

For **select**, a regular path pattern (RPP) represents the paths from a node to target nodes reached by the paths and all the subgraphs below the target nodes are unified by graph union ∪ and returned as a result. Multiple matches from a given node through identical labels are encoded in an automaton by transitions (nondeterministic branches) to different sub-automata that encodes different subsequent patterns. This is implemented in **sfun** that calls different **sfun**s associated with the target states and unify the graphs returned by those called **sfun**s by graph union ∪. If, during the traversal, no further match is possible for the RPP, then no further transition in the automata is possible (i.e., dead state is reached) and correspondingly the **sfun** returns empty ({}). Note that there is no semantic problem if the NFA for **select** is determinized. It is just unnecessary to determinize in order to preserve semantics. Also note that by dead sates, we mean states with no transitions, not useless states like those unreachable from the initial state.

Instead, **extend** uses RPP to specify target nodes and unify the subgraphs below each of these nodes with a given subgraph by ∪, while the rest of the input graph is kept intact. It is implemented by traversing from the top as **select** does, copying the traversed path, instead of discarding the traversed path in case of **select**. Before reaching the target node during the traversal, if no more match is possible, then the original subgraph should just be returned. This situation is also represented by a special dead state in the automata encoding RPP, and corresponding **sfun** just returns the subgraph. On the other hand, if transitions with an identical label have multiple matches, NFA based encoding would utilize different sub-automata for the targets of these transitions and corresponding **sfun** unifies the results by ∪. Suppose input graph $\$db = \{\mathtt{a} : \{\mathtt{b} : \{\mathtt{x} : \{\}\}\}\}$ and transformation

$$\textbf{extend}\ (\mathtt{a.b})|(\mathtt{a.c}) \to \$v\ \textbf{with}\ \{\mathtt{y} : \{\}\}\ \textbf{in}\ \$db$$

The NFA approach would generate an automaton in which target sub-automata from the initial state via transition by a would be different, each encoding the rest of the patterns b and c. So corresponding **sfun** would call two **sfun**s for input a and unify them. Then, the former sub-automaton would copy b and then produce matching result by unifying {x : {}} and given graph {y : {}}, thus, {b:{x:{}, y:{}}}. On the other and, the latter sub-automaton would produce non-matching result after copying b the original subgraph, thus, {b:{x:{}}}. So the unified result would be {a:{b:{x:{}, y:{}}}, a:{b:{x:{}}}}. However, semantics of **extend** should return {a:{b:{x:{}, y:{}}}} instead. Although they are trace equivalent, they are not bisimilar. On the contrary, DFA approach produces DFA with only one target state from the transition by label a from the initial state. The target sub-automaton is only one. Therefore, corresponding **sfun** produces linear preceding paths before reaching target node of the RPP, i.e., {a : {b : _}}. Similar arguments apply for **replace** and **delete**. Therefore, DFA is essential to implement **extend/replace/delete** correctly. It is true that exponential blowup might occur during determinization, however, if the RPP is not complex, i.e., it is not large and corresponding NFA does not include many nondeterministic branches, the number of states will remain moderate. Note that the need of DFA does not come directly from the dead states, but from the problem of unifying the result of subautomaton which might behave differently (some of which might reach dead state and the other reach matching state, as in the example given above).

## 5 Evaluation and applications

Here, we demonstrate the power (expressiveness and efficiency) of the GRoundTram system through the development of a known nontrivial (bidirectional) model transformation between UML class diagrams and relational databases models, and highlight its usefulness in practice by giving a list of important applications developed by other groups using it.

### 5.1 Developing bidirectional Class2RDB

Class2RDB is a model transformation proposed by Bézivin et al. [7] as a common benchmark example for all the participants of the Model Transformation in Practice workshop for comparing and contrasting various approaches to model transformation. Class2RDB maps Class models to RDB models. For instance, it can

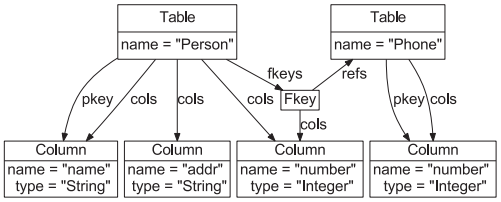



Fig. 10 An RDB model.

be used to transform the Class model in Figure 3 into the RDB model in Figure 10. Class2RDB maps each persistent class in a Class model to a table in the RDB model. All attributes of the class and its subclasses are mapped to columns in the corresponding table. If a primary attribute belongs to the class, a `pkey` reference from the table model element pointing the corresponding column model element is established. If an attribute belongs to its subclass which is persistent, a reference describing foreign key to the corresponding table model element is established.

We show that UnQL$^+$ is powerful enough to (compositionally) describe the forward transformation (from class diagrams to relational database models), while getting the backward transformation for free in our framework. Figure 11 shows the whole transformation in UnQL$^+$. Let us briefly explain how this UnQL$^+$ program was developed by splitting the transformation into two steps. In the first step (denoted by the binding of `$tables_step1`), every persistent class is mapped to a subgraph representing table, which is connected to subgraphs that correspond to columns originated from attributes of the class and its subclasses. All subclasses are collected by using regular path patterns as shown in Section 4.1. If necessary, references `pkey` and `fkeys` are added by an **extend** construct in UnQL$^+$, provided that the references `refs` of `Fkey` do not point to the referring table because the table may not have been constructed yet. They point to the name of the referring table instead. In the second step (denoted by the binding of `$tables_step2`), each name pointed to by `refs` is replaced by the corresponding table by using a **replace** construct.

## 5.2 Optimization and efficiency

Next, we show that the forward and backward transformations can be run efficiently in a scalable manner, while the inefficiency due to composition can be automatically removed through our fusion optimization. The fusion operation was originally designed to merge two successive applications of structural recursions into one [10]. It was later enhanced in [28], [29].

Table 1 summarizes the performance of bidirectional

Table 1 Summary of experiments (running time is in CPU seconds).

| | direction | no rewriting | rewriting | |
|---|---|---|---|---|
| *Class2RDB* | forward | 1.18 | 0.68 | |
| | backward | 14.5 | 7.90 | |
| *PIM2PSM* | forward | 0.08 | 0.07 | (13) |
| | backward | 1.62 | 0.75 | |
| *C2Osel* | forward | 0.04 | 0.05 | (11) |
| | backward | 0.26 | 0.27 | |
| *C2Osel'* | forward | 0.05 | 0.04 | (11) |
| | backward | 2.53 | 1.26 | |
| *Ex1* | forward | 0.036 | 0.007 | (1) |
| | backward | 0.83 | 0.69 | |

transformations on various compositional transformations. The test used a MacOSX on a 17 inch MacBookPro, with a 3.06 GHz Intel Core 2 Duo CPU. An edge-renaming algorithm was used in the measurement, and no modification was actually made, since it does not significantly affect the running time. The rightmost column shows the running time with rewriting optimization. The number of nodes and edges of the graph (Figure 4) that encoded the Class model (Figure 3) in *Class2RDB* were respectively 70 and 73. The sizes of the source models in the other transformations were similar. (See the analysis at the end of this subsection for the performance with respect to the size of the input graph.) In the table, *PIM2PSM* stands for Platform Independent Model to Platform Specific Model transformation, *C2Osel* for transformation of a customer oriented database into an order oriented database, followed by a simple selection, and *Ex1* for the example from our previous paper [25], which in tern was borrowed from [10]. *Ex1* is a composition of two structural recursions. The numbers in parentheses show how often the fusion transformation happened. Our rewrites led to performance improvements in both directions. As the run-time optimization, unreachable parts were removed after every application of the UnCAL structural recursion operator. This run-time optimization is effective when the composed transformation has high selectivity (generates small output from large input), whereas fusion is effective when the selectivity is low. Note that this optimization was turned off for *C2Osel'*. The slowdown in *C2Osel* after rewriting accounts for this trade-off. For the principles of this rewriting optimization, please refer to our papers [28], [29]. You can test other optimizations, like, e.g., subgraph computation optimization, at our project's website [2].

To account for the slowdown of the backward transformation compared with the forward transformation, we took a sample execution profile. The backward

```
select $tables_step2 where
  $tables_step1 in
    (select $tables where
       {Class:$class} in (select $assoc where {Association.(src|dest):$assoc} in $db),
       {is_persistent.Boolean:true} in $class,
       $dests in (select {Class:$dest} where {(src_of.Association.dest.Class)+:$dest} in $class),
       $related in ({Class:$class} U $dests),
       $cols in (select {cols:{Column:{name:$n,type:$t}}}
                 where {Class.attrs.Attribute:{name:$n,type:$t}} in $related),
       $tables in (select {Table:{name:$cname} U $cols} where {name:$cname} in $class),
       $tables in (extend Table -> $table with $pkeys U $fkeys in $tables where
                     {cols:$cols} in $table,
                     {Column.name.String:{$cname:{}}} in $cols,
                     $pkeys in (select {pkey:$cols} where
                                  {attrs.Attribute: {is_primary.Boolean:true, name.String:{$pname:{}}}} in $class,
                                  $cname = $pname),
                     $fkeys in (select {fkeys:{Fkey:{cols:$cols, ref:$ref}}} where
                                  {Class:{is_persistent.Boolean:true,
                                          attrs.Attribute.name.String:{$aname:{}}, name:$ref}} in $dests,
                                  $cname = $aname))),
  $tables_step2 in (replace Table.fkeys.Fkey.ref -> $ref by {Table:$table} in $tables where
                     {Table:$table} in $tables_step1,
                     {String:{$rname:{}}} in $ref,
                     {name.String:{$tname:{}}} in $table,
                     $tname = $rname)
```

Fig. 11 Class2RDB in UnQL+.

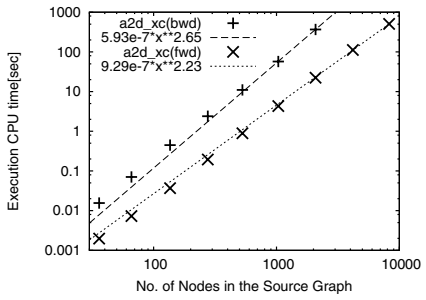



Fig. 12 Transformation time v.s. source graph size.

transformation decomposes a graph using reachable subgraph extraction computation. In addition, the structural recursion **let sfun** $\ldots e_b \ldots$ **in** $e_a$ involves restoring the input of the argument expression $e_a$, which in turn requires examining environment produced by the backward transformation of the structural recursion and superposing the resultant graphs in order to restore the entire graph as the updated target of the argument expression. The backward transformation of *PIM2PSM* (after rewriting) uses 15 times more node id comparison operations compared with the forward transformation. The node comparisons are a result of adding or looking up nodes or edges in sets or maps that are implemented by using balanced binary trees in the OCaml standard library.

Figure 12 shows how the size of the source model affects time to execute the *a2d_xc* transformation described in Section 3, in both directions. Lattice-like regularly shaped strongly connected graphs are used as the source. These execution times match the complexity of PTIME mentioned in Section 4.1 for relatively large (several thousand nodes) graphs.

### 5.3 Other applications

The GRoundTram website [2] has a bunch of examples, big and small, and all the examples presented in this paper can be tried through the demo website. In addition, we would like to give the reader a rough idea about the current status of GRoundTram uses by listing applications that have been or are being developed by other groups: they include bidirectional feature model transformation [48] (Peking University), bidirectional transformation between VDM (Vienna Development Method) specifications and Java implementation [34] (another group at National Institute of Informatics), bidirectional transformation between Simulink diagrams and UML diagrams [47] (Waseda University), bidirectionalizing ATL (ATLAS Transformation Language) with GRoundTram [42] (Shibaura Institute of Technology), and co-evolution of Java models and codes [51] (Open University & Shanghai Jiao Tong University). Moreover, Chen [11] at Shanghai Jiao Tong University identifies GRoundTram as a potential means to synchronize the behavior model of concurrent systems. All these activities indicate the promise of GRoundTram as a practical tool.

As an example of our own experience in [51], we extensively used the UnQL+ syntax as an internal rep-

resentation that was automatically generated by the blinkit[*4] tool. Although it was not directly used by users, it was a concise way of describing the user editing operations to Java code, and also for exploiting high-level optimization opportunities at this syntax level.

## 6 Related work

Besides the related work in the introduction, we highlight some others related to graph-based model transformation and linguistic approach to bidirectional transformation.

Our work is much related to research on model transformation based on graph transformation. AGG [46] is a rule-based visual tool that supports typed (attributed) graph transformations, including type inheritance and multiplicities. Triple Graph Grammars (TGG) [21], [37] is intended as a declarative specification of model-to-model integration rules. MOFLON [4] implements TGG and adopts the visual notation of QVT Relational, the OMG standard bidirectional model transformation language. Giese and Hildebrandt [20] proposed a model synchronization algorithm based on TGG that can synchronize large-scale models. Guerra et al. [23] proposed more general notion called "inter-modelling", where a specification can be compiled into different operational mechanisms not only for model-to-model transformation but also for model matching and model traceability. To perform forward and backward transformations, a pattern specification is compiled into Triple Graph Grammar (TGG) operational rules.

Different from these rule-based approaches, ours is a functional one that supports model transformation composition and its automatic optimization. As far as we are aware, this is the first nontrivial *functional and algebraic framework for model transformation*.

This study was inspired by the recent linguistic approach to bidirectionalization of the tree transformation [9], [17], [18], [32], [39] for tree data synchronization. One important feature of this approach is clear bidirectional semantics, something that is missing from most of the existing bidirectional model transformation systems [43]. Although some attempts at using the linguistic approach have been made [6], [50], it remains a challenge to provide a general bidirectional framework for graphs which are more complicated than trees, and our work is a big step in this direction.

GRoundTram has grown out of our two-year effort to realize the emergent idea presented in our short paper [26]. UnQL$^+$ is based on the graph query language UnQL [10] but it is significantly extended with a powerful language construct *replace* that can handle transformation context. It is also worth noting that a simple *replace* expression was studied in [25] but it can neither deal with regular path expressions nor treat multiple graph databases.

[*4] http://computing-research.open.ac.uk/linkit/

## 7 Conclusions

We proposed a novel algebraic framework called GRoundTram to support systematic development of bidirectional model transformation. Different from many rule-based frameworks, our framework is functional and algebraic, and based on a graph algebra and structural recursion. Our new framework supports systematic development of model transformations in a compositional manner, has a clear semantics for bidirectional model transformation, and can be efficiently implemented.

This is our first step towards *bidirectional model programming*, a linguistic framework to support systematic development of model transformation programs. In the future, we will look more into relation between the rule-based approach and the algebraic and functional approach, and see how to integrate them into a more powerful framework for bidirectional model transformation. One initial attempt has already been made by integrating GRoundTram with ATL [42], and we plan to continue this line of research by collaborating with the AtlanMod [1] team. In such an integration of model transformation frameworks, we need to automate the graph encoding of UML-like generic diagrams that are not yet fully automated. Several attempts towards automation had been made [52], and we plan to deal with Eclipse Modeling Framework (EMF) models as well. Another direction towards integration is the approach by Wider [49]. In this approach, asymmetric lens was implemented as an internal DSL in Scala. Although it is not graph-based, the Java-friendliness of the host language may make it easier to integrate with the model driven engineering framework.

Last but not least, we currently do not have any explicit control over update policy in backward transformations. The only property we have is well-behavedness described by the (GETPUT) and the (WPUTGET). Even if multiple source models may be possible as a result of backward transformation, programmer do not have a way to choose them. Controlling the choice is one of our important future work.

## Acknowledgments

We would like to thank the anonymous reviewers for their thorough comments and constructive suggestions to improve the paper. The anonymous ASE'11 reviewers for the prior version of this paper also made helpful suggestions. We also thank Kazutaka Matsuda, Kazuyuki Asada, and Isano Sasano for their valuable

discussions with us. The research was supported in part by the Grand-Challenging Project on the "Linguistic Foundation for Bidirectional Model Transformation" of the National Institute of Informatics, and a Grant-in-Aid for Scientific Research for Encouragement of Young Scientists (B) No. 20700035, and a Grant-in-Aid for Scientific Research (B) No. 22300012.

## References

[1] The AtlanMod team web site: http://www.emn.fr/x-info/atlanmod/

[2] The BiG project web site: http://www.biglab.org/

[3] S. M. Abramov and R. Glück, "Principles of inverse computation and the universal resolving algorithm," In *The Essence of Computation*, pp.269–295, 2002.

[4] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr, MOFLON, "A Standard-Compliant Metamodeling Framework with Graph Transformations," In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference*, vol.4066 of *Lecture Notes in Computer Science (LNCS)*, pp.361–375, Heidelberg, Springer Verlag, 2006.

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

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

[7] J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt, "Model transformation in practice workshop announcement," In *Satellite Events at the MoDELS 2005 Conference*, pp.120–127, Springer-Verlag, 2005.

[8] A. Bohannon, B. C. Pierce, and J. A. Vaughan, "Relational lenses: a language for updatable views," In S. Vansummeren, editor, *PODS*, pp.338–347, ACM, 2006.

[9] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt, "Boomerang: resourceful lenses for string data," In G. C. Necula and P. Wadler, editors, *POPL '08: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pp.407–419. ACM, 2008.

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

[11] Yuting Chen, "A bidirectional graph transformation approach to analysis of concurrent software models," In *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on*, pp.339–343, july 2010. doi: 10.1109/ICSESS.2010.5552447.

[12] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, "Bidirectional transformations: A cross-discipline perspective," In *International Conference on Model Transformation (ICMT 2009)*, pp.260–283, LNCS 5563, Springer, 2009.

[13] Z. Diskin, Y. Xiong, and K. Czarnecki, "From state- to delta-based bidirectional model transformations: the asymmetric case," *Journal of Object Technology*, vol.10, no.6, pp.1–25, 2011.

[14] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Taentzer, "Information preserving bidirectional model transformations," In *Proceedings of the 10th international conference on Fundamental approaches to software engineering*, FASE'07, pp.72–86, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71288-6.

[15] K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, G. Taentzer, D. Varró, and S. Varró-Gyapay, "Model transformation by graph transformation: A comparative study," Presented at *MTiP 2005*. http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2005/mtip05.pdf, 2005.

[16] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz and dynagraph - static and dynamic graph drawing tools," In *GRAPH DRAWING SOFTWARE*, pp.127–148, Springer-Verlag, 2003.

[17] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, "Combinators for bi-directional tree transformations: a linguistic approach to the view update problem," In *POPL '05: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pp.233–246, 2005.

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

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

[20] H. Giese and S. Hildebrandt, "Efficient model synchronization of large-scale models," *Technical Report 28, Hasso Plattner Institute at the University of Potsdam*, 2009.

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

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

[23] E. Guerra, J. Lara, and F. Orejas, "Inter-modelling with patterns," *Software & Systems Modeling*, pp.1–

30, 2011. ISSN 1619-1366. doi: 10.1007/s10270-011-0192-1.

[24] F. Hermann, H. Ehrig, F. Orejas, K. Czarnecki, Z. Diskin, and Y. Xiong, "Correctness of model synchronization based on triple graph grammars," In *Lecture Notes in Computer Science*, vol.6981, pp.668–682, Springer, 2011. ISBN 978-3-642-24484-1.

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

[26] S. Hidaka, Z. Hu, H. Kato, and K. Nakano, "A compositional approach to bidirectional model transformation," In *ICSE Companion*, pp.235–238, IEEE, 2009.

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

[28] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, K. Nakano, and I. Sasano, "Marker-directed Optimization of UnCAL Graph Transformations," In *Proceediings of 21st International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2011)*, vol.7225 of LNCS, pp.123–138, Odense, Denmark, 2011.

[29] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, K. Nakano, and I. Sasano, "Marker-directed Optimization of UnCAL Graph Transformations (revised version)," *Technical Report GRACE-TR-2011-06*, GRACE Center, National Institute of Informatics, Nov. 2011.

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

[31] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano, "Tupling calculation eliminates multiple data traversals," In *ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pp.164–175, Amsterdam, The Netherlands, ACM Press, June 1997.

[32] Z. Hu, S.-C. Mu, and M. Takeichi, "A programmable editor for developing structured documents based on bidirectional transformations," *Higher-Order and Symbolic Computation*, vol.21, no.1-2, pp.89–118, 2008.

[33] K. Inaba, S. Hidaka, Z. Hu, H. Kato, and K. Nakano, "Graph-transformation verification using monadic second-order logic," In P. Schneider-Kamp and Michael Hanus, editors, *PPDP*, pp.17–28, ACM, 2011.

[34] F. Ishikawa, "Towards customizable and bidirectionally traceable transformation between vdm++ and java," In *The 9th Overture/VDM Workshop*, June 2011.

[35] F. Jouault and J. Bézivin, "KM3: A DSL for metamodel specification," In *Formal Methods for Open Object-Based Distributed Systems*, pp.171–185, LNCS 4037, Springer, 2006.

[36] F. Klar, A. Königs, and A. Schürr, "Model transformation in the large," In I. Crnkovic and A. Bertolino, editors, *ESEC/SIGSOFT FSE*, pp.285–294, ACM, 2007.

[37] A. Königs and A. Schürr, "Tool integration with triple graph grammars - a survey," *Electronic Notes in Theoretical Computer Science*, vol.148, no.1, pp.113–150, Feb. 2006.

[38] R. Lämmel, "Coupled Software Transformations (Extended Abstract)," In *First International Workshop on Software Evolution Transformations*, Nov. 2004.

[39] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi, "Bidirectionalization transformation based on automatic derivation of view complement functions," In *12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*, pp.47–58, ACM Press, Oct. 2007.

[40] K. Nakano, S. Hidaka, Z. Hu, K. Inaba, and H. Kato, "Simulation-based graph schema for view updatability checking of graph queries," *Technical Report GRACE-TR11-01*, GRACE Center, National Institute of Informatics, May 2011.

[41] R. Paige and R. Tarjan, "Three partition refinement algorithms," *SIAM Journal of Computing*, vol.16, no.6, pp.973–988, 1987. DOI: http://dx.doi.org/10.1137/0216062.

[42] I. Sasano, Z. Hu, S. Hidaka, K. Inaba, H. Kato, and K. Nakano, "Toward bidirectionalization of ATL with GRoundTram," In *Theory and Practice of Model Transformations, Fourth International Conference, ICMT 2011*, vol.6707 of *LNCS*, pp.138–151, Springer, June 2011.

[43] P. Stevens, "Bidirectional model transformations in QVT: Semantic issues and open questions," In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *Proc. 10th MoDELS*, vol.4735 of *Lecture Notes in Computer Science*, pp.1–15, Springer, 2007.

[44] P. Stevens, "A landscape of bidirectional model transformations," In R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II*, pp.408–424, Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-88642-6.

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

[46] G. Taentzer, "AGG: A graph transformation environment for modeling and validation of software," In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *AGTIVE*, vol.3062 of *LNCS*, pp.446–453, Springer, 2003.

[47] T. Kozawa, "Bidirectionaltransformation with UML model for Simulink model maintainability improvement (in Japanese)," http://www.washi.cs.waseda.ac.

jp/papers/2011/submission/1w070119.pdf, Feb. 2011. Summary of the bachelor's thesis at the Department of Conputer Science, Waseda University.

[48] B. Wang, Z. Hu, Q. Sun, H. Zhao, Y. Xiong, and H. Mei, "Supporting feature model refinement with updatable view," *Technical Report GRACE-TR-2010-05*, GRACE Center, National Institute of Informatics, May 2010.

[49] A. Wider, "Towards combinators for bidirectional model transformations in scala," In A. M. Sloane and U. Aßmann, editors, *SLE*, vol.6940 of *Lecture Notes in Computer Science*, pp.367–377, Springer, 2011. ISBN 978-3-642-28829-6.

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

[51] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux, "Blinkit: Maintaining Invariant Traceability through Bidirectional Transformations," In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, Zurich, Switzerland, pp.540–550, June 2012.

[52] Y. Zhu, T. Zan, S. Hidaka, and Z. Hu, "iGRT: A generic interface for GRoundTram," *Technical Report GRACE-TR-2012-06*, GRACE Center, National Institute of Informatics, 2012.

**Soichiro HIDAKA**

Soichiro Hidaka is an assistant professor at the National Institute of Informatics and The Graduate University for Advanced Studies in Japan. He received his bachelor's degree and Ph. D in Engineering from The University of Tokyo in 1994 and 1999 respectively. He had involved in research projects such as parallel programming language implementation, micro-kernel based operating system and document processing system. His research interests include infrastructure software systems, and database programming languages in particular. Recently he has been conducting research on bidirectional graph transformations that are intended to facilitate bidirectional model transformations. He is a member of ACM, IPSJ, IEICE and JSSST.

**Zhenjiang HU**

Zhenjiang Hu is Professor of National Institute of Informatics (NII) and The Graduate University for Advanced Studies in Japan. He received his BS and MS from Shanghai Jiao Tong University in 1988 and 1991 respectively, and PhD degree from University of Tokyo in 1996. He was a lecture (1997-1999) and an associate professor (2000-2007) in University of Tokyo, before joining NII as a full professor in 2008. His main interest is in programming languages and software engineering in general, and functional programming, parallel programming and bidirectional model-driven software development in particular. He is now serving on the steering committees of ACM ICFP, APLAS and FLOPS, and is the academic committee chair of the NII Shonan Meetings.

**Kazuhiro INABA**

Kazuhiro Inaba is a software engineer of Google Inc. He received PhD degree from University of Tokyo in 2009. He was a researcher in National Institute of Informatics from 2009 to 2011. His main interest is in formal languages, in particualr in the complexity of automata and transducers over strings and trees.

**Hiroyuki KATO**

Hiroyuki Kato is Assistant Professor of National Institute of Informatics (NII) and The Graduate University for Advanced Studies in Japan. He received BS from Tokyo University of Science in 1990, and MS and PhD degree from Nara Institute of Science and Technology in 1996 and 1999 respectively. He joined National Center for Science Information Systems (NACSIS) from 1999 as a research associate. His research interests include semistructured databases, database programming languages and query optimization by program transformation.

**Keisuke NAKANO**

Keisuke Nakano is an associate professor at the University of Electro-Communications, Japan. He received his Ph.D. degree from Graduate School of Science, Kyoto University. He was a postdoctoral researcher at the University of Tokyo from 2003 to 2008. He has assumed his current post since April 2012 after working there as an assistant professor from 2008. His research interests include formal language theory, programming language theory, functional programming, and theorem proving. He is a member of JSSST, IPSJ, and ACM.