Graph-Transformation Verification using Monadic Second-Order Logic

Kazuhiro Inaba
kinaba@nii.ac.jp

Soichiro Hidaka
hidaka@nii.ac.jp

Zhenjiang Hu
hu@nii.ac.jp

National Institute of Informatics, Japan

Hiroyuki Kato
kato@nii.ac.jp
National Institute of Informatics, Japan

Keisuke Nakano
ksk@cs.uec.ac.jp
The University of Electro-Communications

ABSTRACT

This paper presents a new approach to solving the problem of verification of graph transformation, by proposing a new static verification algorithm for the Core UnCAL, the query algebra for graph-structured databases proposed by Bunemann et al. Given a graph transformation annotated with schema information, our algorithm statically verifies that any graph satisfying the input schema is converted by the transformation to a graph satisfying the output schema. We tackle the problem by first reformulating the semantics of UnCAL into monadic second-order logic (MSO). The logic-based foundation allows to express the schema satisfaction of transformations as the validity of MSO formulas over graph structures. Then by exploiting the two established properties of UnCAL called bisimulation-genericity and compactness, we reduce the problem to the validity of MSO over trees, which has a sound and complete decision procedure. The algorithm has been efficiently implemented; all the graph transformations in this paper and the system web page can be verified within several seconds.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

Keywords

Validation, Graph Transformation, Monadic Second-Order Logic

1. INTRODUCTION

Graphs are very useful means to describe complex structures and systems and to model concepts and ideas in a direct and intuitive way [2], and a number of languages, such as UnQL [7], Lorel [1], Graphlog [9], have been proposed for graph transformations [23].

UnCAL (Unstructured Calculus), being the underlying algebra of the graph query language UnQL, is one of the useful graph transformation languages for efficient graph transformations [6]. It is recently adopted for bidirectional model-driven software development [16, 15], where software components in different levels of abstraction are modeled as graphs, and their relation is described as graph transformations.

In these applications, it is often assumed, for each graph transformation, that its input and output graphs have some structure (*schema*) in them. However, due to the complicated structure like cyclic reference of graphs, it is not straightforward for programmers to write a transformation that produces schema-conforming outputs for every valid input. It is thus very important to provide a static verification algorithm to check if the transformation is correct with respect to the input and output schemas, which describe structural constraints of graph databases [4].

The objective of this paper is to provide a static verification algorithm for transformations in UnCAL. More specifically, what we want to solve is the following problem:

> *Verification Problem*: Given an UnCAL transformation f, an input schema φ_{IN}, and an output schema φ_{OUT}, determine whether "for any graph g satisfying φ_{IN}, the output graph $f(g)$ satisfies φ_{OUT}".

Although many efforts have been devoted to verification of tree transformations [26, 21, 20, 12], there is little work on verification of graph transformation. One challenge here is that many verification problems turn to be undecidable when going from trees to graphs. Therefore, to deal with verification of graph transformation, we should carefully impose reasonable constraints on graphs and graph transformations.

One attempt made on verification of UnCAL transformation was to use simulation-based schema [5] (with constraints on the schema). There, a schema itself is again a graph, and data graphs simulated by the schema graph (i.e., any traversal on the data graph can be replicated on the schema graph) are defined to conform to the schema. The advantage of such a schema is the simplicity of verification of transformations. Since the input schema itself is a graph, it can be passed as an argument to the transformation; the transformation is valid if the outcome is subsumed by the output schema. However, it has very limited expressiveness on structures of graphs. Basically, simulation can state only conjunctions of optional conditions, like "there can be an outgoing edge labeled `foo` and there can be another edge of `bar`". It fails to describe a condition such as, “under the `contact` edge, we must have either `phone` edge or `mail` edge, but not both”. Such “either one of” feature is, however, crucial for writing structural constraints; it can be seen in all the standard XML schema languages [11, 29, 8] or in metamodeling language [3].

In this paper we propose a new approach to the verification problem based on the two important characteristics of UnCAL, *bisimulation-equivalence* of graphs and *structured recursion*, where a graph transformation in the Core UnCAL can be automatically checked against a schema in the powerful monadic second order logic (MSO). Our verification system enjoys the following features.

- Our verification system is *powerful*. First, it allows graph schemas to be described in terms of MSO. MSO has exactly the power of expressing regular languages [24], being widely used as a schema language for XMLs and graphs. The structural constraints expressible by commonly used graph schema language KM3 [19] is just in this category. Second, it accepts any graph transformation defined in terms of type-annotated Core UnCAL so that all the types can be fully checked.
- Our verification system is *fully automatic and decidable*. We propose an automatic algorithm that can map the type-annotated Core UnCAL to an MSO-definable graph transduction [10], and show that verification of such the MSO property on graphs can be reduced to that on infinite trees, which is decidable. In particular, if the graph transformation is *compact* [7], the problem can be reduced to verification on finite trees.

  In addition, thanks to the property that the inverse image of an MSO-definable set of graphs under an MSO-definable transduction is MSO-definable, validity of the transformation can be checked by the input-side subsumption. This makes it possible to generate a more understandable *counterexample with respect to the input* rather than on the output, which is in sharp contrast to the simulation-based approach [5].
- Our verification system is *efficient and practical* especially for *compact* [7] transformations. As not only schemas but also transformations can be described by MSO formula, and verification of graph transformation in UnCAL can be efficiently implemented[1] with MONA [14] MSO solver. In fact, all the examples in this paper can be verified by our system within several seconds.

The paper is structured as follows. In Section 2, we give an overview of our approach with an example for showing the taste how our verification works. In Section 3, we explain the graph data-model and transformation of Core UnCAL. In Section 4, we introduce MSO, and their usage as schema language. Section 5 is the main technical part, which shows how to translate Core UnCAL programs to MSO formula. Then in Section 6 we discuss two theorems that ensures the decidability of the generated MSO formulas. Section 7 compares the present paper by related work, and Section 8 concludes.

2. OVERVIEW

Before proceeding with the technical details, let us demonstrate through several examples to show a very informal overview how our verification works.

[1]The implementation is available at `http://www.biglab.org`.

2.1 A Simple Example

Consider the friend graph $\$db$ in Figure 1(a), which consists of a set of members, each member having a name, a contact information (either mail or phone), and a set of friends. The structure of this graph can be described by the following schema definition:

```
type Members = { mem  :  Person }
type Person  = {
  name    :  Data,
  contact :  MailOrPhone,
  friend  :  Person   }
type MailOrPhone = Mail | Phone
type Mail   = { mail   :  Data }
type Phone  = { phone  :  Data }
```

Now suppose that we want to transform this graph by renaming `mem` to `member`, `friend` to `knows`, and flattening the contact information. This transformation can be described as $\mathit{flatten}\ (\mathit{rename}(\$db))$ where *flatten* and *rename* can be defined by structured recursions as follows.

$$
\begin{aligned}
\mathit{rename} = \ & \mathbf{rec}(\lambda(\$L_1,\$G_1).\\
& \quad \&_1 := \mathbf{if}\ \$L_1 = \texttt{mem}\ \mathbf{then}\ \{\texttt{member} : \&_1\}\\
& \qquad\quad \mathbf{else\ if}\ \$L_1 = \texttt{friend}\ \mathbf{then}\ \{\texttt{knows} : \&_1\}\\
& \qquad\quad \mathbf{else}\ \{\$L_1 : \&_1\})\\
\mathit{flatten} = \ & \mathbf{rec}(\lambda(\$L_1,\$G_1).\\
& \quad \&_1 := \mathbf{if}\ \$L_1 = \texttt{contact}\ \mathbf{then}\ \$G_1\\
& \qquad\quad \mathbf{else}\ \{\$L_1 : \&_1\})
\end{aligned}
$$

Now our verifier can check that the above transformation is correct in the sense that if the input is `Member`, the output will always produce the graph meeting the following structure:

```
type Members2  = {member:Person2}
type Person2   = PM | PP
type PM={name:Data, mail:Data, knows:Person2}
type PP={name:Data, phone:Data, knows:Person2}
```

2.2 An Example of Verification Procedure

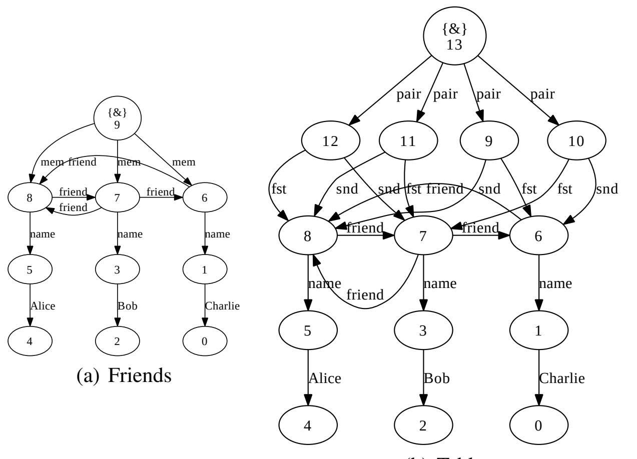
Our second example is to transform the friend graph to a friend-pair graph with the following structure:

```
type Pair   = { fst: Person, snd: Person }
type Pairs  = { pair: Pair }
```

For instance, the graph structured data in Figure 1(a) is transformed to the table-like structure in Figure 1(b).

To make sure this transformation does generate a structure that we intuitively expect, we annotate schema information to the UnCAL code. By using this schema, we describe the expected type of each graph-variable and a return-expression of the **rec** recursion as follows, where input schema φ_{IN} corresponds to Members, and output schema φ_{OUT} corresponds to Pairs.

$$
\begin{aligned}
& \mathbf{rec}(\lambda(\$L_1,\$G_1).\\
& \quad \&_1 :: \mathsf{Pairs} := \mathbf{if}\ \$L_1 = \texttt{mem}\ \mathbf{then}\\
& \qquad \mathbf{rec}(\lambda(\$L_2,\$G_2).\\
& \qquad\quad \&_1 :: \mathsf{Pairs} := \mathbf{if}\ \$L_2 = \texttt{friend}\ \mathbf{then}\\
& \qquad\qquad \{\texttt{pair} : \{\texttt{fst} : \$G_1 :: \mathsf{Person}, \texttt{snd} : \$G_2 :: \mathsf{Person}\}\}\\
& \qquad\qquad\qquad\qquad \mathbf{else}\ \{\}\\
& \qquad)(\$G_1)\\
& \qquad\qquad\qquad \mathbf{else}\ \{\}\\
&)(\$db :: \mathsf{Members}) :: \mathsf{Pairs}
\end{aligned}
$$



(a) Friends (b) Table

Figure 1: Example Graph Data

Then what the verifier confirms are: (1) under the assumption $\$db$ conforms to the type Members, the node bound to $\$G_1$ during recursion always conforms to the type Person, (2) under the assumption $\$G_1$ conforms to the type Person, the node bound to $\$G_2$ during recursion always conforms to the type Person, (3) under the assumption $\$G_1$ and $\$G_2$ conforms to the type Person, the inner most recursion returns a graph conforming to Pairs for each edge, (4) under the assumption that the inner recursion returns Pairs graphs, the outer recursion returns Pairs, and (5) the whole expression evaluates to a Pairs graph. Our verifier is sound, that is, if the verifier answers that all the above conditions hold, then it does hold. Also it is complete in the sense that if it says the conditions may be broken, then there indeed is a concrete assignment of graphs to variables that breaks the conditions. In such a case, our verifier emits an instance of a counter-example variable assignment that does break the conditions imposed by output schemas. For instance, if we forgot to write the generation of an edge $\{\texttt{pair} : \cdots\}$, the verifier reports an error with a counter-example. In this case, any input graph can be a counter-example. But the following example more appreciates the power of our contribution: the transformation extracts `contact` information, assuming it only has `Mail` information, the verifier report the counter-example of the input having `Phone`.

$$
\begin{aligned}
&\mathbf{rec}(\lambda(\$L_1, \$G_1).\\
&\quad \&_1 :: \mathsf{Pairs} := \mathbf{if}\ \$L_1 = \texttt{mem}\ \mathbf{then}\\
&\qquad \mathbf{rec}(\lambda(\$L_2, \$G_2).\\
&\qquad\quad \&_1 :: \mathsf{Pairs} := \mathbf{if}\ \$L_2 = \texttt{contact}\ \mathbf{then}\ G_2 :: \mathsf{Mail}\\
&\qquad\qquad\qquad \mathbf{else}\ \{\})(\$G_1)\\
&\qquad\qquad \mathbf{else}\ \{\})(\$db :: \mathsf{Members}) :: \mathsf{Mail}
\end{aligned}
$$

The check is carried out in the following three steps. Firstly, the schema is converted to a logic formula (more specifically, a formula of MSO logic) that exactly stats the conditions that are imposed by the schema.

Secondly, the annotated UnCAL transformation is converted into a set of MSO formulas describing the transformation. For instance, from the root node of the formula, the following is the excerpt of the set of formulas generated.

$$
\begin{aligned}
&\mathbf{edge}_{\texttt{pair},3,4,5}(x,y,z) :=\\
&\qquad \exists^f v, e, u.(x = y = z = e \wedge \mathbf{edge}_{\texttt{friend},1,1,1}(v,e,u))\\
&\mathbf{edge}_{\texttt{fst},5,6,1}(x,y,z) :=\\
&\qquad \exists^f v, e, u.(x = y = e \wedge z = \mathsf{root} \wedge \mathbf{edge}_{\texttt{friend},1,1,1}(v,e,u))\\
&\mathbf{edge}_{\texttt{snd},5,7,2}(x,y,z) :=\\
&\qquad \exists^f v, e, u.(x = y = e \wedge z = \mathsf{root} \wedge \mathbf{edge}_{\texttt{friend},1,1,1}(v,e,u))
\end{aligned}
$$

We assign a number (we call *copy-id*) 1 to the graph bound to the variable $\$G_1$ and the number 2 to $\$G_2$ (and 0 to $\$db$). The subformula $\mathbf{edge}_{\texttt{friend},1,1,1}(v,e,u)$ asserts that v and u are nodes of copy-id 1, and e is an edge with label `friend` connecting them. The nodes and edges created by the transformation is also numbered (in this case, we use 3 to 7).

$$\begin{tikzcd}
 & & (u,1)\\
(v,3) \arrow[r, "\overset{\texttt{pair}}{(e,4)}"] & (e,5) \arrow[ur, "\overset{\texttt{fst}}{(e,6)}"] \arrow[dr, "\underset{\texttt{snd}}{(e,7)}"'] & \\
 & & (u,2).
\end{tikzcd}$$

The definition of the predicate $\mathbf{edge}_{\texttt{pair},3,4,5}(x,y,z)$, for example, can be read as follows: "if 1st copy of e is an edge of label `friend`, then (and only then) an edge of label `pair` is drawn from the 3rd copy of e and 5th copy of e." This is essentially a complete description of the transformation represented by MSO.

Thirdly, the MSO formulas representing schema conformance is then expanded to a formula that only uses the predicates $\mathbf{edge}_{\texttt{pair},k,k,k}(x,y,z)$ arose from the variables, (i.e., k is a copy-id assigned to a variable, not a generated output). For instance, the type annotation $\&_1 :: \mathsf{Pairs}$ asserts that the return-value of the body of the recursion must satisfy the schema formula:

$$
\begin{aligned}
\mathit{isPairs}(x) := \exists^s \mathit{Pairs}.\ \exists^s \mathit{Pair}.\ \exists^s \mathit{Person}.\ & x \in \mathit{Pairs}\\
&\wedge\ \cdots\\
&\wedge\ \forall^f y \in \mathit{Pair}.\ \forall^f z\ w.\ \mathbf{edge}_{\texttt{fst}}(y,z,w) \rightarrow w \in \mathit{Person}\\
&\wedge \cdots).
\end{aligned}
$$

Since the body expression generates nodes and edges having the 1st to the 7th copy-id, the formula is instantiated to use $\mathbf{edge}_{\texttt{fst},3,4,5}$ etc. instead of the bare $\mathbf{edge}_{\texttt{fst}}$. The conversion is an inductive expansions of $\forall$ and $\exists$ into a finite number of $\wedge$s and $\vee$s, e.g., $\forall^f x.\psi(x)$ is converted to $\forall^f x.\psi_1(x) \wedge \ldots \wedge \psi_7(x)$ where ψ_i is a result for inductive transformation of the subformula ψ assuming that the variable x points to the i-th copy entity. After this process, the conditions that need to be verified can be written as a single MSO formula, which is valid on any interpretation of $\mathbf{edge}_{\_,1,1,1}$ if and only if the conditions are always satisfied.

Finally, the validity of the generated MSO formula is checked. Technical problem here is that validity of MSO on graphs is undecidable in general [27]. Fortunately, we can manage the problem by utilizing the property called bisimulation-genericity, which is shared in common for all UnCAL transformations; for bisimulation generic transformations, the validity on graphs can be reduced to the decidable validity on infinite trees [22]. Furthermore, the property called compactness that holds among a certain subset of UnCAL allows to reduce the validity problem to that on finite trees. On finite tree domain, good existing MSO solvers can be exploited for our implementation.

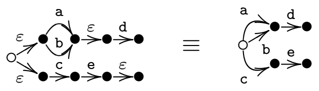
3. CORE UNCAL: A GRAPH TRANSFORMATION LANGUAGE

We present the target language of our verification technique: a core fragment of UnCAL graph algebra, and recall important aspects of the language (for the detail, see [7]).

3.1 Graph Data Model

UnCAL deals with rooted, directed, finite-branching and edge-labeled graphs whose nodes conveying no particular information. We fix the finite set *Label* of labels and the set *Data* of data values throughout the paper. We assume a special label $\varepsilon \notin \mathit{Label}$, and denote by $\mathit{Label}_\varepsilon$ the set $\mathit{Label} \cup \{\varepsilon\}$. We usually write the elements of *Label* by typewriter font like `a`, `foo`, or `name`, and write the elements of *Data* as double-quoted strings like `"John"` or `"3.14"`. A graph $g = (V, r, E)$ consists of a set V of *nodes*, a designated root node $r \in V$, and a set E of *edges* equipped with three mappings: $\mathrm{src} : E \to V$, $\mathrm{lab} : E \to \mathit{Label}_\varepsilon \cup \mathit{Data}$, and $\mathrm{dst} : E \to V$. The mappings src and dst denote the source and the destination node of the edge respectively, and lab denotes the label of the edge. We often write (v, l, u) to indicate the edge e with $\mathrm{dst}(e) = u$, $\mathrm{lab}(e) = l$, and $\mathrm{src}(e) = v$.

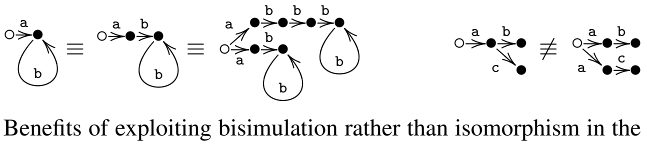
UnCAL's graph model has ε-edges resembling ε-transitions of automata, which work as shortcuts between nodes. Schemas and transformations will be defined to respect this intention of ε-edges. For example, the following two graphs are considered to be semantically equivalent.



(*)

Here, the white circle ◦ denotes the root node of each graph. The reason for using ε-edges is to make the transformation language as simple as possible. For instance, we do not need a union operator $\tau_1 \cup \tau_2$ of two edge-sets explicitly, because it can be simulated by a construction of a new node having two outgoing ε-edges, as exemplified by the root node of the figure above. We define the set $E^{\to}(v)$ of *outgoing edges* of a node v as the set of non-ε edges reachable from v by traversing only ε-edges. That is, $e = (v', l, u) \in E^{\to}(v)$ if and only if $l \neq \varepsilon$ and there exists a sequence $v = v_0, v_1, \ldots, v_k = v'$ of nodes with $(v_i, \varepsilon, v_{i+1}) \in E$ for $i \geq 0$.

In addition, two graphs in UnCAL are considered to be equal if they are bisimilar. Graphs $g_1 = (V_1, r_1, E_1)$ and $g_2 = (V_2, r_2, E_2)$ are defined to be *bisimilar* and written $g_1 \equiv g_2$ if there exists a relation (called *(extended-)bisimulation*) $S \subseteq V_1 \times V_2$ satisfying the following conditions: (1) $(r_1, r_2) \in S$, (2) for all $(v_1, v_2) \in S$ and $(\_, l, u_1) \in E_1^{\to}(v_1)$, there exists u_2 such that $(\_, l, u_2) \in E_2^{\to}(v_2)$ and $(u_1, u_2) \in S$, and (3) for all $(v_1, v_2) \in S$ and $(\_, l, u_2) \in E_2^{\to}(v_2)$, there exists u_1 such that $(\_, l, u_1) \in E_1^{\to}(v_1)$ and $(u_1, u_2) \in S$. Here _ is wild-card pattern indicating existence of some element whose particular value is not cared. Intuitive understanding of bisimulation is that unfolding of cycles and duplication of equivalent subgraphs are not distinguished, and unreachable part from the root is ignored. In particular, a rooted graph always has a (possibly infinite) tree bisimilar to it; it is obtained by infinitely unfolding all the cycles and sharings. Note that bisimulation is different from a weaker notion "set of all paths from root is equal".



Benefits of exploiting bisimulation rather than isomorphism in the semantics are throughly discussed in [7] and not repeated here.

3.2 Core UnCAL

We define Core UnCAL, a subset of UnCAL graph algebra. The syntax is shown in Fig. 2. In addition, we syntactically restrict the uses of markers $\&_i$. Markers do not occur globally nor directly in the argument expression τ in an expression $\mathbf{rec}(\cdots)(\tau)$; they can only appear in the body expressions of $\mathbf{rec}$s.

The relationship between the Core UnCAL and the full UnCAL resembles that of the Core XPath [13] and XPath XML Query Language. That is, manipulation of the data values (comparison with data-values $\$l =$ `"John"` or $\$l_1 = \$l_2$ in the if-expressions, and operations on labels such as $\{$`"foo"`$+\$l : \{\}\}$) are prohibited in Core UnCAL. Also, we have simplified the use of markers (they can only be used for connecting $\mathbf{rec}$ bodies), but this is just a syntactic difference. All the UnCAL expressions compiled from its front-end language UnQL satisfies the syntactic condition. Except the restrictions, the full computational power of UnCAL is also available in Core UnCAL.

We hope the intuition of the most of the constructs is clear. Node construction expression $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ creates a fresh node v and edges $\{(v, l_1, r_1), \ldots, (v, l_n, r_n)\}$ where r_i is the root node of the graph obtained by evaluating the expression τ_i. Variable reference and conditional branch is defined as usual. The isEmpty Boolean expression returns true if and only if the passed node has no outgoing edge. The output marker expression $\&_i$ is used only in the body of $\mathbf{rec}$ expressions as explained below. The distinct feature of UnCAL is that basically all graph manipulations are expressed in terms of one unified and powerful construct called *structural recursion*, expressed by the $\mathbf{rec}(\ldots)$ expression.

3.2.1 Structural Recursion

Let us first explain the structural recursion in intuitive fashion by using a union operator $\cup$ for two graphs temporally for the sake of explanation. A function f on graphs is called a structural recursion if it is defined by the following equations [2]

$$
\begin{array}{lcl}
f(\{\}) & = & \{\} \\
f(\{\$l : \$g\}) & = & \omega(\$l, \$g) \odot f(\$g) \\
f(\{\$l_1 : \$g_1\} \cup \ldots \cup \{\$l_n : \$g_n\}) & & \\
\qquad = f(\{\$l_1 : \$g_1\}) \cup \ldots \cup f(\{\$l_n : \$g_n\}) \ , & &
\end{array}
$$

where $\odot$ is a given binary operator and the term $\omega(\$l, \$g)$ does not contain recursive calls to f. Different choices of $\odot$ define different functions. Since the first and the third equations are common in all structural recursions, we may omit them and simplify the above definition as:

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

As a simple example, we may use the following structural recursion to replace all edges labeled `a` by `d` and delete the edges labeled `c` for an input graph.

$$
\begin{array}{lcl}
\mathbf{sfun}\ \mathit{a2d\_xc}(\{\$l : \$g\}) & = & \mathbf{if}\ \$l = \mathtt{a}\ \mathbf{then}\ \{\mathtt{d} : \mathit{a2d\_xc}(\$g)\} \\
 & & \mathbf{else\ if}\ \$l = \mathtt{c}\ \mathbf{then}\ \mathit{a2d\_xc}(\$g) \\
 & & \mathbf{else}\ \{\$l : \mathit{a2d\_xc}(\$g)\}
\end{array}
$$

The recursion $\mathbf{sfun}\ f\ \{\$l : \$g\} = \omega(\$l, \$g) \odot f(\$g)$ is represented in Core UnCAL by

$$\mathbf{rec}(\lambda(\$l, \$g).(\&_1 := \omega(\$l, \$g) \odot \&_1)).$$

For example, the structural recursive function $\mathit{a2d\_xc}$ shown in the

[2]Informally, the meaning of this definition can be considered to be a fixed point (though may not necessarily unique) over the graph, which is again defined by a set of equations using the three constructors $\{\}$, :, and $\cup$. For instance, the graph marked with $(*)$ in Section 3.1 can be considered to be the fixed point of the following equations: $G_{\mathrm{root}} = \{\mathtt{a} : G_1, \mathtt{b} : G_1\}, \mathtt{c} : \{\mathtt{e} : \{\}\}$ and $G_1 = \{\mathtt{d} : \{\}\}$

$$
\begin{array}{lcll}
\tau & ::= & \{l:\tau,\ldots,l:\tau\} & \text{node with edges}\\
 & \mid & \$g & \text{variable reference}\\
 & \mid & \textbf{if}\ b\ \textbf{then}\ \tau\ \textbf{else}\ \tau & \text{conditional}\\
 & \mid & \&_i & \text{output marker}\\
 & \mid & \textbf{rec}(\lambda(\$l,\$g).\ \&_1:=\tau,\ldots,\&_n:=\tau)(\tau) & \text{structural recursion}\\
l & ::= & \$l & \text{label variable reference}\\
 & \mid & \mathsf{a} & \text{label } (\mathsf{a}\in \mathit{Label}_\varepsilon \cup \mathit{Data})\\
b & ::= & \$l=\mathsf{a} & \text{label comparison } \mathsf{a}\in \mathit{Label}\\
 & \mid & \mathsf{isEmpty}(\$g) & \text{emptiness checking}\\
 & \mid & b\ \mathsf{and}\ b \mid b\ \mathsf{or}\ b \mid \mathsf{not}\, b & \text{logical connectives.}
\end{array}
$$

Figure 2: Core UnCAL Language

$$
\begin{array}{lcll}
v_f & = & \{x,y,\ldots\} & \text{1st order variables}\\
t_f & ::= & v_f \mid \mathsf{root} & \text{1st order terms}\\
v_s & = & \{X,Y,\ldots\} & \text{2nd order variables}\\
t_s & ::= & v_s \mid t_s\cup t_s \mid t_s\cap t_s \mid \emptyset & \text{2nd order terms}
\end{array}
$$

$$
\begin{array}{lcl}
\varphi & ::= & \mathsf{true} \mid \mathsf{false}\\
 & \mid & \neg\varphi \mid \varphi\vee\varphi \mid \varphi\wedge\varphi \mid \varphi\rightarrow\varphi \mid \varphi\leftrightarrow\varphi\\
 & \mid & t_f=t_f \mid t_s=t_s \mid t_f\in t_s \mid t_s\subseteq t_s\\
 & \mid & \exists^f v_f.\varphi \mid \forall^f v_f.\varphi \mid \exists^s v_s.\varphi \mid \forall^s v_s.\varphi\\
 & \mid & \mathbf{vert}(t_f) \mid \mathbf{edge}_l(t_f,t_f,t_f)
\end{array}
$$

Figure 3: Syntax of Monadic Second-Order Logic

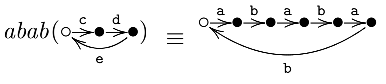
above is represented by

$$
\begin{array}{r}
\textbf{rec}(\lambda(\$l,\$g).\&_1 := \textbf{if}\ \$l=\mathsf{a}\ \textbf{then}\ \{\mathsf{d}:\&_1\}\\
\textbf{else if}\ \$l=\mathsf{c}\ \textbf{then}\ \{\epsilon:\&_1\}\\
\textbf{else}\ \{\$l:\&_1\}).
\end{array}
$$

Let us explain by an example. Up to bisimulation, the following UnCAL expression *abab*

$$\textbf{rec}(\lambda(\$l,\$g).\ \&_1:=\{\mathsf{a}:\&_2\},\ \&_2:=\{\mathsf{b}:\&_1\})(\$db)$$

changes all edges of even distance from the root node to a, and odd distance edges to b. Here, $\$db$ is a designated variable referring to the input graph and $\tau(g)$ for any UnCAL expression τ should be read as "evaluate τ under the environment $\{\$db\mapsto g\}$".



Note that in our Core UnCAL, $\&_1$ always corresponds to the defined function.

As we have mentioned in the explanation of the graph data model, the semantics of UnCAL is carefully designed to treat bisimilar graphs equally. Indeed, it is proved that all UnCAL transformations are bisimulation-generic (Proposition 4 of [7]), that is, for any $g\equiv g'$, we have $f(g)\equiv f(g')$.

4. GRAPH SCHEMA IN MSO

We employ powerful monadic second-order logic (MSO) to describe a graph schema which specifies structural constrains of graphs. MSO is first-order logic extended with set quantification. It has exactly the power of expressing regular tree languages [24], being widely used as a schema language for XMLs and graphs.

The syntax of the formula of MSO over edge-labeled graph structure is shown in Fig. 3. We adopt a variant of MSO which

$$
\begin{array}{lcl}
\mathit{Schema} & ::= & \mathit{Decl}\cdots\mathit{Decl}\\
\mathit{Decl} & ::= & \mathtt{type}\ \mathit{Name}=\{\mathit{Edge},\ldots,\mathit{Edge}\}\\
 & \mid & \mathtt{type}\ \mathit{Name}=\{\mathit{Edge},\ldots,\mathit{Edge},*\}\\
\mathit{Edge} & ::= & \mathit{Label}:\mathit{Type}\\
\mathit{Type} & ::= & \mathit{Name} \mid \mathsf{Data} \mid \mathit{Type}\mid\mathit{Type}
\end{array}
$$

Figure 4: Graph Schema Language $\mathcal{GS}$

is used to describe so called $(2,2)$-*definable MSO transductions* of Courcelle [10], with customizations to adjust for our purpose, namely adding the root constant and making edge predicates $\mathbf{edge}_l$ inspect labels. For a graph $g=(V,r,E)$ and an environment Γ that maps first-order variables to $V\cup E$ and second-order variables to subsets of $V\cup E$, the entailment relation $g,\Gamma\vDash\varphi$ is defined. We present the definition of the two graph-specific primitives:

$$
\begin{array}{ll}
g,\Gamma\vDash\mathbf{vert}(t) & \text{if } \Gamma(t)\in V\\
g,\Gamma\vDash\mathbf{edge}_l(t_1,t_2,t_3) & \text{if } \Gamma(t_2)=(\Gamma(t_1),l,\Gamma(t_3))\in E
\end{array}
$$

where Γ is extended as $\Gamma(\mathsf{root})=r$. The other entailment relations follow the standard definition. We write $g\vDash\varphi$ when $g,\Gamma\vDash\varphi$ holds for the empty environment Γ. Note that UnCAL's semantics is defined up to bisimulation as explained in Section 3. MSO formulas that distinguish bisimilar graphs are not suitable for describing properties of UnCAL graphs. We say that a closed MSO formula φ is *bisimulation-generic*, if $g\equiv g'$ implies $g\vDash\varphi$ iff $g'\vDash\varphi$.

An MSO formula φ with one free variable can be regarded as a graph schema. For a graph $g=(V,r,E)$ and a given formula φ with one free variable x, we can say that g conforms to φ when $g,x\mapsto r\vDash\varphi$ holds. We define the bisimulation genericity of schemas in a way similar to closed formulas. We say that an MSO formula φ with one free variable x is bisimulation-generic if $g\equiv g'$ implies $g,x\mapsto v\vDash\varphi$ iff $g',x\mapsto v'\vDash\varphi$ for any nodes v in g and v' in g' where v and v' are bisimilar. In the rest of the paper, by schema we mean a bisimulation-generic MSO formula with one free variable.

Adopting MSO formula as a front-end language of graph schemas may not be a good choice, however. In particular, it may be difficult to write correctly MSO formula while making sure its bisimulation genericity. It would be better to provide a graph schema language which is inherently bisimulation-generic and which can be automatically translated into MSO formula. As an example, the schema language $\mathcal{GS}$ in Fig. 4 fulfills the requirements. Its concrete semantics and its translation to MSO formula can be found in [17]. For instance, the graph schema `Members` presented in Section 2 is written in $\mathcal{GS}$, and can be systematically translated into the following bisimulation-generic MSO formula:

$$
\begin{aligned}
&\exists^s X_{\texttt{Members}}.\ \exists^s X_{\texttt{Person}}.\ \exists^s X_{\texttt{MailOrPhone}}.\ \exists^s X_{\texttt{Mail}}.\ \exists^s X_{\texttt{Phone}}.\\
&\quad \mathsf{root} \in X_{\texttt{Members}} \wedge\\
&\quad \forall^f v.\mathbf{vert}(v) \rightarrow\\
&\qquad v \in X_{\texttt{Members}} \rightarrow \varphi_{\texttt{Members}}(v) \wedge\\
&\qquad v \in X_{\texttt{Person}} \rightarrow \varphi_{\texttt{Person}}(v) \wedge\\
&\qquad v \in X_{\texttt{MailOrPhone}} \rightarrow \varphi_{\texttt{MailOrPhone}}(v) \wedge\\
&\qquad v \in X_{\texttt{Mail}} \rightarrow \varphi_{\texttt{Mail}}(v) \wedge\\
&\qquad v \in X_{\texttt{Phone}} \rightarrow \varphi_{\texttt{Phone}}(v)
\end{aligned}
$$

where each formula $\phi_S(v)$ with a schema name S is defined using its declaration. For example, the formula $\varphi_{\texttt{Members}}(v)$ is given by

$$
\begin{aligned}
&\exists^s O.\ e\_out(v, O) \wedge\\
&\quad \forall^f e.\ e \in O \wedge \neg\mathbf{vert}(e) \rightarrow\\
&\qquad \exists^f x.\ \exists^f y.\ \mathbf{edge}_{\texttt{mem}}(x, e, y) \wedge y \in X_{\texttt{Person}}
\end{aligned}
$$

Here, $e\_out(v, O)$ is a predicate for checking if O is a set of non-ε edges reachable from v by traversing only ε-edges, which is implemented in a standard technique for representing transitive closures in MSO.

Note that $\mathcal{GS}$ is just an example of a front-end schema language. The results in the following sections are not specific to $\mathcal{GS}$. It is applicable to any schemas representable in MSO.

5. CORE UNCAL IN MSO

In our verification method, not only schemas but also transformations are represented by MSO. Then, we combine the MSO formulas for transformations with those for schemas into a single MSO formula, whose validity is equivalent to the correctness of the transformation with respect to the schemas.

The difficulty here is how to map the structural recursion of UnCAL that iteratively walks through graphs to an MSO formula that declaratively represents a relationship between input and output graphs. This problem is addressed by exploiting an alternative semantics called *bulk semantics* of UnCAL [7], which more fits to logical formulation, and known to be equivalent to the usual recursive semantics.

Another challenge comes from the fact that MSO-definable transduction intentionally has been restricted its expressiveness to keep many important properties decidable. Not all Core UnCAL expressions can be translated into such a restricted class of MSO-definable transductions for the reason mentioned later. To avoid the problem and give a terminating decision procedure, we ask programmers to add several annotations on UnCAL, which provides schema information on intermediate result graphs. The annotations should be put on certain subexpressions.

This section first introduces the formalism to specify transformations in terms of MSO formula, and then shows how such formulas can be constructed from Core UnCAL.

5.1 MSO-Definable Graph Transduction

We basically adopt the formalism in [10] called *MSO-definable transduction* for specifying graph transformations in MSO. We, however, slightly generalize the formalism to what we call *MSO-definable transduction system* in order to give a simpler translation from UnCAL and an easier treatment of annotations.

Definition 1. MSO-definable transduction system is a tuple $\mathcal{M} = (I, S, D_{\mathrm{v}}, D_{\mathrm{e}})$ where I is a finite set called the set of *copy-ids*, S a nonempty subset of I called the *input set*, D_{v} a partial mapping that maps each $i \in I \setminus S$ to an *extended*-formula $\mathbf{vert}_i(y)$, and D_{e} a partial mapping that maps each $(l, i, j, k) \in \mathit{Label}_\varepsilon \times (I^3 \setminus S^3)$ to an extended-formula $\mathbf{edge}_{l,i,j,k}(x, y, z)$. Here, extended-formula is an MSO formula that has $\mathbf{vert}_i(x)$ and $\mathbf{edge}_{l,i,j,k}(x, y, z)$ for $i, j, k \in I$ and $l \in \mathit{Label}_\varepsilon$ as primitives, instead of $\mathbf{vert}(x)$ and $\mathbf{edge}_l(x, y, z)$.

In MSO-definable transductions, output graphs are considered to be constructed by first generating $|I \setminus S|$ copies of the input graph (hence the name *copy-id* is given for the set I), and then reorganizing the edge/vert relations among them according to the formulas in D_{v} and D_{e}. The essential difference of MSO-definable transduction systems as above from the original definition in [10] is that each $\mathbf{edge}_{l,i,j,k}$ and $\mathbf{vert}_i$ can be defined in terms of other $\mathbf{edge}_{l',i',j',k'}$ and $\mathbf{vert}_{i'}$. In the original version, they are only allowed to be defined in terms of the original input. This difference does not change their expressiveness of graph transductions.

We only consider *acyclic* systems. That is, there must be a total order on I such that in the definition of formulas $\mathbf{vert}_i$ and $\mathbf{edge}_{l,i,j,k}$, all the occurrences of elements of I must be strictly smaller than i and j. We often write $\mathbf{edge}_{l,i,j,k}(x, y, z) := \varphi$ to mean $D_{\mathrm{e}}(l, i, j, k) = \varphi$, and write similarly of $\mathbf{vert}_i$.

Let us explain the idea by the following example with $I = \{0, 1, 2\}$ and $S = \{0\}$:

$$
\begin{aligned}
\mathbf{edge}_{\texttt{buz},2,2,2}(x, y, z) &:= \mathbf{edge}_{\texttt{bar},1,1,1}(x, y, z)\\
\mathbf{edge}_{\texttt{bar},1,1,1}(x, y, z) &:= \mathbf{edge}_{\texttt{foo},0,0,0}(x, y, z)\\
\mathbf{vert}_2(y) &:= \mathbf{vert}_1(y)\\
\mathbf{vert}_1(y) &:= \mathbf{vert}_0(y)
\end{aligned}
$$

The input set S denotes the set of copy-ids for input graphs of the transformation defined by this system. Thus, the formula $\mathbf{edge}_{\texttt{foo},0,0,0}(x, y, z)$ is read as "in the input graph, x, y, and z form an edge labeled `foo`". Intuitively speaking, in an MSO-definable transduction system, output graphs are thought to be created by copy-and-edit from the input graphs. In the above example, $|I \setminus S| = 2$ copies of the input nodes and edges are created by the system, and are reorganized to form the output graph, guided by the supplied formulas. For instance, the 1st copies of x, y, and z form a `bar` edge if and only if they are a `foo` edge in the input. The 2nd copies of them form a `buz` edge if their 1st copies form a `bar` edge, which happens only when they form a `foo` edge in the original input. In other cases, no edge is drawn. After all, if we regard $\{2\} \subseteq I$ as the output graph of this system, the transformation defined by the system is what renames all the edges `foo` to `buz` and eliminates all the other edges. If we regard $\{1\}$ as the output, it defines the transformation renaming `foo` to `bar` and eliminating others.

In general, S may not be a singleton. In such a case, the system represents a transformation taking multiple inputs $g_1, g_2, \ldots, g_{|S|}$. Even in the case, we can regard them as a single-input transformation, by assuming a virtual input graph $g = \{\texttt{elem} : g_1, \texttt{next} : \{\texttt{elem} : g_2, \texttt{next} : \cdots\}\}$ and considering each g_i as one of the output graphs from the transduction system (each g_i can be extracted by a simple subgraph extraction, and it can easily be written in a set of MSO-formulas). Hence, in the following discussion in this subsection we assume a single input $S = \{s\}$.

Formally, for a nonempty set $J \subset I$, copy-id $\rho \in J$, and graph $g = (V, r, E)$, the transduction system defines an output graph $g_{J,\rho} = (V', r'E')$ by

- $V' = \{(v, i) \in (V \cup E) \times J \mid g, \{y \mapsto v\} \vDash \mathbf{vert}'_i(y)\}$,
- $E' = \{((v, i), (w, j), (u, m)) \in ((V \cup E) \times J)^3 \mid g, \{x \mapsto v, y \mapsto w, z \mapsto u\} \vDash \mathbf{edge}'_{l,i,j,m}(x, y, z)\}$, and

$$\begin{array}{rcl} q & ::= & \tau :: \varphi \\ \tau & ::= & \{l : \tau, \ldots, l : \tau\} \\ & | & \textbf{if } b \textbf{ then } \tau \textbf{ else } \tau \\ & | & \&_i \\ & | & \textbf{rec}(\lambda(\$l, \$g).\ \&_1 :: \varphi := \tau, \ldots, \&_n :: \varphi := \tau)(\tau) \\ & | & \$g :: \varphi \end{array}$$

Figure 5: Type Annotated Core UnCAL

- $r' = (r, \rho)$

where $\textbf{vert}'_i(y)$ is the formula obtained by recursively replacing $\textbf{vert}_i(y)$ with $D_v(i)$ (if $D_v(i)$ is not defined, it is replaced with $\textbf{vert}(y)$ when $i = s$ and otherwise with false) and $\textbf{edge}_{l,i,j,k}(x, y, z)$ with $D_e(l, i, j, k)$ (if $D_e(l, i, j, k)$ is not defined, it is replaced with $\textbf{edge}_l(x, y, z)$ when $i = j = k = s$ and otherwise with false).

The following lemma is important in MSO-definable transduction systems. The inverse image of an MSO-definable set of graphs under an MSO-definable transduction system is MSO-definable.

LEMMA 1 ([10], PROP. 3.2). *Let $\mathcal{M} = (I, \{s\}, D_v, D_e)$ be an MSO-definable transduction system, $J \subset I$, $\rho \in J$, and a closed MSO formula φ. Then there exists an MSO formula $\text{inv}(\mathcal{M}, J, \rho, \varphi)$ such that, for any graph g, we have $g \vDash \text{inv}(\mathcal{M}, J, \rho, \varphi)$ if and only if $g_{J,\rho} \vDash \varphi$.*

The lemma enables us to convert MSO formulas on output graphs into that on input graphs. Using this conversion, the verification problem that tests the assertion "for any input graph g, if it conforms to the input schema (i.e, $g \vDash \varphi_{\text{IN}}$), then $g_{J,\rho} \vDash \varphi_{\text{OUT}}$" can be restated as the validity of a single formula "$\varphi_{\text{IN}} \rightarrow \text{inv}(\mathcal{M}, J, \rho, \varphi_{\text{OUT}})$" on input graphs.

One limitation of MSO-definable transduction systems is that by definition it can represent only *linear-size increase* transformations; the size $|g_{J,\rho}|$ of the nodes in the output graphs is linearly bounded by the input size $|J||g|$. In UnCAL, superlinear growth is caused only by using nested-recursions. This is exactly the reason why our verifier, as explained later, requires annotation for such a case.

5.2 Adding Annotations to Core UnCAL

Annotations are supposed to be supplied by programmers in the syntax shown in Figure 5, which we call the type annotated Core UnCAL. The nonterminal q represents the whole program. Here the programmer can specify the schema for the output database (i.e., the result of the evaluation of the whole UnCAL expression τ). In the **rec** expression, the occurrence of variables $\$g$ and the body expressions of the recursion accept the schema annotation. In conventional programming languages, this means that every function is having type annotation on its parameters and return values.

Intuitively, the annotation $\$g :: \varphi$ on parameters works for the verifier in two ways. (1) The graph pointed by the node bound to $\$g$ must conform to the schema φ: the verifier is obliged to verify the conformance. (2) In the body of the rec expression, the use of graph $\$g$ can be assumed to be bound to a node pointing to an arbitrary graph satisfying φ: the verifier can use this assumption. The annotations $\&_i :: \varphi := \tau$ on the markers also have two roles. One is to tell that the verifier must make sure that the result of evaluating this expression must conform to the schema φ. Another is to tell the verifier that the result of evaluating the whole $\textbf{rec}(\ldots)$ expression can be approximated as an arbitrary graph that is constructed as the union of the graphs conforming to φ_1, where φ_1 is the supplied schema annotation to the first body expression $\&_1$.

5.3 Type Annotated Core UnCAL to MSO

From now on, we consider a fixed annotated Core UnCAL program q and explain how to translate it to MSO. For the finite copy-id set I in the definition of MSO-definable transduction system, we use the set *Cid* of elements generated by the following BNF

$$\textit{Cid} ::= \textit{CodePos} \mid \langle \textit{Cid}, \textit{CodePos}, \mathbb{N} \rangle$$

where *CodePos* is a set of unique identifiers assigned to each subexpression of q, and $\mathbb{N}$ is the set of natural numbers. The angle brackets $\langle\rangle$ just denote tupling. Although the set *Cid* is infinite, in the following construction we only use finite portion of them. More specifically, the nesting depth of $\langle\rangle$s are at most the nesting depth of recursions in the given UnCAL transformation, and the natural numbers $\mathbb{N}$ used is at most $\max(2, 2n, 2m)$ where n is the number of markers and m the maximum number of outgoing edges of the node-construction expression in the transformation.

We inductively define a procedure ft2mso that converts a type annotated Core UnCAL expression to a set of MSO formulas. It has the following form:

$$\text{ft2mso}(c, \Gamma, \varphi)(\tau^p) = (\mathcal{M}, J, \rho, O, A).$$

It takes four parameters (three of them are to hold contextual information used during the conversion, and the last one is the UnCAL expression) and returns a tuple consisting of five components. The fourth parameter τ^p, which is separately parenthesized for emphasizing its special position, denotes the UnCAL expression to be converted. The superscript ^{p} denotes the code-position of the subexpression. The first parameter c is a triple (c_v, c_e, c_u) of copy-ids denoting the ids of the current edge. The meaning of this parameter should become clear when we reach to the formal definition of ft2mso that deals with **rec** expressions. The second parameter Γ is the mapping from variable names to the copy-id of the graph denoted by the variable. The third parameter φ is an MSO formula representing the condition for the current subexpression to be executed; in other words, it is a conjunction of the condition of **if** expressions enclosing the current expression.

Then it computes five components simultaneously. The first component $\mathcal{M}$ is an MSO-definable transduction system that represents the UnCAL transformation τ. The second J and the third ρ components are to denote the copy-ids of the output graph obtained by evaluating τ. The fourth O and the fifth A components are sets of MSO formulas, which represent the conditions that are *O*bligations to satisfy and that can be *A*ssumed, respectively. They correspond to the two roles of annotations as explained before. They are stored in the form of triple (J, ρ, ψ) meaning that the output graph $g_{J,\rho}$ must (or can be assumed to) satisfy ψ.

Let us show a very simple example of the translation. Consider the type-annotated UnCAL expression $\{\texttt{foo} : \$db :: \varphi_1\} :: \varphi_0$ that simply prepends an edge labeled a to the input graph $\$db$. Let the code positions of each subexpression p, q, and r, i.e.,

$$(\{\texttt{foo} : (\$db :: \varphi_1)^r\}^q :: \varphi_0)^p.$$

Translation of the expression will yield the following MSO-definable transduction system

$$\begin{aligned} \mathcal{M} = (I = \{&\langle c, q, 0\rangle, \langle c, q, 1\rangle, \langle c, r, 0\rangle, \langle c, r, 1\rangle, r\}, \\ S = \{&r\}, \\ D_v = \{ & \textbf{vert}_{\langle c,q,0\rangle}(y) := (y = \mathsf{root}) \\ & \textbf{vert}_{\langle c,r,0\rangle}(y) := (y = \mathsf{root})\ \}, \\ D_e = \{ & \textbf{edge}_{\texttt{foo},\langle c,q,0\rangle,\langle c,q,1\rangle,\langle c,r,0\rangle}(x, y, z) := \psi \\ & \textbf{edge}_{\varepsilon,\langle c,r,0\rangle,\langle c,r,1\rangle,r}(x, y, z) := \psi\ \}\ ) \end{aligned}$$

where $\psi \equiv \exists^f v,e,u.(x=e \wedge y=e \wedge z=e \wedge e=\mathsf{root})$ (which is equivalent to $x = y = z = \mathsf{root}$) and $c = \langle p,p,1\rangle$. The system involves five copy-ids, and one of them, r, represents its input graph. In addition to the original input graphs, it adds to nodes $\langle c,q,0\rangle$-th and $\langle c,r,0\rangle$-th copies of the root node, and two edges labeled `foo` and ε (addition of ε-edge is a technical subtlety which is not important).

In addition to the system, the translation gathers the obligation and assumption formulas as follows:

$$O = \{\ (\{\langle c,q,0\rangle,\langle c,q,1\rangle,\langle c,r,0\rangle,\langle c,r,1\rangle,r\},\langle c,q,0\rangle,\varphi_0[\mathsf{root}])\ \}.$$
$$A = \{\ (\{r\},r,\varphi_1[\mathsf{root}])\ \}.$$

That is, the verifier must make sure that the output graph conforms to the schema φ_0, under the assumption that the input graph satisfies φ_1. Hence, the correctness of the transformation with respect to annotations are equivalent to the validity of the following MSO formula.

$$\mathrm{inv}(\mathcal{M},\{r\},r,\varphi_1[\mathsf{root}]) \rightarrow \mathrm{inv}(\mathcal{M},I,\langle c,q,0\rangle,\varphi_0[\mathsf{root}])$$

Testing procedure of this kind of MSO formula is discussed in Section 6.

Whole Program.

The whole program of type annotated UnCAL consists of an expression τ and a schema annotation $::\ \varphi$. It is translated as follows; it first translates the body expression into the corresponding transduction system, and adds an obligation formula stating that the output graph must conform to φ.

$$\begin{aligned}
&\mathrm{ft2mso}(\_,\_,\_)((\tau :: \varphi)^p) = (\mathcal{M},J,\rho,O_0\cup O,A)\\
&\quad\textbf{where}(\mathcal{M},J,\rho,O,A) = \mathrm{ft2mso}(c,\{\$db \mapsto p\},e{=}\mathsf{root})(\tau)\\
&\qquad\qquad O_0 = \{(J,\rho,\varphi[\mathsf{root}])\}\\
&\qquad\qquad c = (\langle p,p,0\rangle,\langle p,p,1\rangle,\langle p,p,2\rangle)
\end{aligned}$$

The first argument c to the recursive call of ft2mso is meant to be a three unique copy-ids that will not conflict with copy-ids used in the other place during translation (conflict avoidance is the reason why we include the code-position of the current expression in copy-ids). The second argument assigns a copy-id to the designated variable $\$db$ denoting the input graph. The third argument is a formula containing possibly three free variables v, e, and u that encodes the condition that the UnCAL expression is executed. In this case, we specify $e{=}$ root to mean we start evaluation from the root node.

THEOREM 1. *Let $q = \tau :: \varphi$ be a type annotated UnCAL program and $(\mathcal{M},\_,\_,O,A) = \mathrm{ft2mso}(q)$, then $\bigwedge_{\bar{a}\in A}\mathrm{inv}(\mathcal{M},\bar{a}) \rightarrow \bigwedge_{\bar{o}\in O}\mathrm{inv}(\mathcal{M},\bar{o})$ is valid if and only if q never violates the schema annotation. In particular, if the formula is valid, then for any input graph, the output graph conforms to φ.*

In the remaining subsections, we give the inductive construction of the translation ft2mso in detail for each kind of UnCAL expression. Although the proof is omitted for brevity, the correctness of the construction can be shown by straightforward induction on the structure of expression, showing that it exactly represents the *bulk semantics* of UnCAL [7].

Node Construction.

Let us examine the rules for subexpressions one by one. The first case is the node-construction. As an exercise, let us first explain the case of node creation $\{l_1 : \tau_1\}$ with only one outgoing edge.

$$\begin{aligned}
&\mathrm{ft2mso}(c,\Gamma,\varphi)(\{l_1:\tau_1\}^p) =\\
&\quad(\ \mathcal{M}_1[(l_1,\langle c_e,p,0\rangle\ e,\langle c_e,p,1\rangle\ e,\rho_1\ e)\mapsto\varphi],\\
&\qquad J_1\cup\{\langle c_e,p,0\rangle,\langle c_e,p,1\rangle\},\quad \langle c_e,p,0\rangle,\\
&\qquad O_1,\quad A_1\ \ )\\
&\quad\textbf{where } (\mathcal{M}_1,J_1,\rho_1,O_1,A_1) = \mathrm{ft2mso}(c,\Gamma,\varphi)(\tau_1)
\end{aligned}$$

Since this node construction expression itself does not have any schema annotation, it does not add any obligation or assumption. Hence, the O_1 and A_1 components are the same as those of the subexpression τ_1.

The first three components describe edges and nodes generated by the current expression. The notation $\mathcal{M}[(l,i\ \alpha,j\ \beta,k\ \gamma)\mapsto\varphi]$ for $\alpha,\beta,\gamma\in\{v,e,u,\mathsf{root}\}$ is a short hand for defining a new MSO-definable transduction system $(I',J,D'_{\mathrm{v}},D'_{\mathrm{e}})$ from $\mathcal{M} = (I,J,D_{\mathrm{v}},D_{\mathrm{e}})$ by $I' = I\cup\{i,j\}$, $D'_{\mathrm{v}} = D_{\mathrm{v}}\cup\{i\mapsto\exists^f xz.\psi, k\mapsto\exists^f xz.\psi\}$, and $D'_{\mathrm{e}} = D_{\mathrm{e}}\cup\{l,i,j,k\mapsto\psi\}$ where ψ is $\exists^f v,e,u.(x{=}\alpha\wedge y{=}\beta\wedge z{=}\gamma\wedge\varphi)$. It should be read as "$i$-th copy of α, j-th copy of β, and k-th copy of γ forms an edge in the output graph of this expression when φ holds" as the picture below:

$$(\langle c_e,p,0\rangle\text{-th copy of } e)\xrightarrow[\langle c_e,p,1\rangle\text{-th copy of } e]{l}\rho\text{-th copy of } e$$

For example, in the example in Section 2, an edge labeled `pair` will be drawn for each edge labeled friend in the input graph. The expression $\{$`pair` $:\ ...\}$ generating the `pair` edge is translated by the ft2mso procedure with the parameter $\varphi = \mathbf{edge}_{\mathtt{friend},c_v,c_e,c_u}(v,e,u)$. Then the transduction system has a definition of an edge as follows:

$$\begin{aligned}
&\mathbf{edge}_{\mathtt{pair},\langle c_e,p,0\rangle,\langle c_e,p,1\rangle,\rho_1}(x,y,z) :=\\
&\qquad\exists^1 v,e,u.(x{=}e\wedge y{=}e\wedge z{=}e\wedge\mathbf{edge}_{\mathtt{friend},c_v,c_e,c_u}(v,e,u)).
\end{aligned}$$

That is, "an edge (which is the $\langle c_e,p,1\rangle$-th copy of e) of label pair is drawn from the $\langle c_e,p,0\rangle$-th copy of e to the ρ_1-th copy of e, only when c-th copy of e is an edge labeled `friend`".

The actual definition of ft2mso is generalized for the case of n outgoing edges, by simply taking the union of the above construction:

$$\begin{aligned}
&\mathrm{ft2mso}(c,\Gamma,\varphi)(\{l_1:\tau_1,\ldots,l_n:\tau_n\}^p) =\\
&\quad(\ \bigcup_{1\le i\le n}\mathcal{M}_i[(l_i,\langle c_e,p,0\rangle\ e,\langle c_e,p,i\rangle\ e,\rho_i\ e)\mapsto\varphi],\\
&\qquad\bigcup_{1\le i\le n}(J_i\cup\{\langle c_e,p,0\rangle,\langle c_e,p,i\rangle\}),\quad\langle c,p,0\rangle,\\
&\qquad\bigcup_{1\le i\le n}O_i,\quad\bigcup_{1\le i\le n}A_i,\quad)\\
&\quad\textbf{where } (\mathcal{M}_i,J_i,\rho_i,O_i,A_i) = \mathrm{ft2mso}(c,\Gamma,\varphi)(\tau_i)\\
&\qquad\qquad\text{for each } 1\le i\le n.
\end{aligned}$$

Here, the union of transduction systems $(I,S,D_{\mathrm{v}},D_{\mathrm{e}})\cup(I',S',D'_{\mathrm{v}},D'_{\mathrm{e}})$ is defined as $(I\cup I',S\cup S',i\mapsto D_{\mathrm{v}}(i)\vee D'_{\mathrm{v}}(i),(l,i,j,k)\mapsto D_{\mathrm{e}}(l,i,j,k)\vee D'_{\mathrm{e}}(l,i,j,k))$.

If Expression.

In fact, **if** expression is quite similar to usual node construction $\{l_1 : \tau_1\}$; it just draws an ϵ-edge pointing to the **then** branch or **else** branch, depending on whether the condition holds or not.

$$\begin{aligned}
&\mathrm{ft2mso}(c,\Gamma,\varphi)((\textbf{if } b \textbf{ then } \tau_1 \textbf{ else } \tau_2)^p) =\\
&\quad(\ \mathcal{M}_1[(\epsilon,\langle c_e,p,0\rangle\ e,\langle c_e,p,1\rangle\ e,\rho_1\ e)\mapsto\varphi\wedge\varphi_b]\\
&\qquad\cup\mathcal{M}_2[(\epsilon,\langle c_e,p,0\rangle\ e,\langle c_e,p,2\rangle\ e,\rho_2\ e)\mapsto\varphi\wedge\neg\varphi_b],\\
&\qquad J_1\cup J_2\cup\{\langle c_e,p,0\rangle,\langle c_e,p,1\rangle,\langle c_e,p,2\rangle\},\quad\langle c_e,p,0\rangle,
\end{aligned}$$

$$O_1 \cup O_2, \quad A_1 \cup A_2 \quad)$$

$$\begin{aligned}\textbf{where } (\mathcal{M}_1, J_1, \rho_1, O_1, A_1) &= \text{ft2mso}(c, \Gamma, \varphi \wedge \varphi_b)(\tau_1)\\ (\mathcal{M}_2, J_2, \rho_2, O_2, A_2) &= \text{ft2mso}(c, \Gamma, \varphi \wedge \neg\varphi_b)(\tau_2)\\ \varphi_b &= \text{b2mso}(b)\end{aligned}$$

The procedure b2mso is to convert boolean condition to MSO formula in a straightforward manner. E.g., the condition $\$l = \texttt{a}$ is converted to $\textbf{edge}_{\texttt{a},c_v,c_e,c_u}(v, e, u)$. Only one complexity is in the isEmpty predicate of Core UnCAL, but it can be dealt with by the standard technique to represent transitive closure in MSO.) One thing that must be noted here is that we assume all label variables $\$l$ are always the innermost-scope variable. This assumption is satisfied by a simple program transformation; since we are now considering the case where the set $Label_\epsilon$ of labels is finite, we can eliminate nested-occurrence of $\$l$'s by first inserting an exhaustive branching **if** $\$l = \texttt{a}\cdots$ **else if** $\$l = \texttt{b}$ **else** $\cdots$ to the scope where the variable $\$l$ is introduced and then instantiate $\$l$ to the concrete label constant in each body of the branching. In fact, this transformation eliminates expressions of the form $\{\$l : \tau\}$ (which we did not consider in the definition of ft2mso above), too.

Marker.

In type annotated UnCAL, markers are always annotated with schema in the top-level of **rec** expression. So, we assign copy-ids for markers during processing **rec** expression, and store it to the environment Γ. At the occurrence site of a marker as an expression our MSO-encoding simply generates an ε-edge and connect to the root node of the graph whose copy-id is stored in Γ. The reason we add ε-edge here is a technical and non-essential reason; we want to make every output nodes/edges copies of input edges e (not root), which make implementation and definition slightly simpler.

$$\begin{aligned}&\text{ft2mso}(c, \Gamma, \varphi)(\&_i{}^p) =\\ &\quad (\ \mathcal{M}_p[(\epsilon, \langle c_e, p, 0\rangle\ e, \langle c_e, p, 1\rangle\ e, \Gamma(\&_i)\ \mathsf{root}) \mapsto \varphi],\\ &\quad\ \{\langle c_e, p, 0\rangle, \langle c_e, p, 1\rangle, \Gamma(\&_i)\}, \langle c_e, p, 0\rangle, \{\}, \{\})\end{aligned}$$

The transduction system $\mathcal{M}_p = (\{p\}, \{p\}, \emptyset, \emptyset)$ is the empty system with the copy-id of input graphs being p.

Variable Reference (Outer Scope).

There are two types of occurrences of variables in expression. One is the innermost-scope variable, which is the variable that is bound in the innermost enclosing **rec** expressions, like $\$g$ in $\textbf{rec}(\lambda(\$l, \$g).\&_1 := \$g)$. Another case is the outer-scope variables, which are bound in the outer **rec** recursion, like $\$g_1$ in $\textbf{rec}(\lambda(\$l_1, \$g_1).\&_1 := \textbf{rec}(\lambda(\$l_2, \$g_2).\&_1 := \$g_1))$. The latter case (and the designated input variable $\$db$) is treated similarly as markers. That is, we simply draw an ε-edge to the root of the graph.

$$\begin{aligned}&\text{ft2mso}(c, \Gamma, \varphi)(\$g :: \psi^p) =\\ &\quad (\ \mathcal{M}_p[(\varepsilon, \langle c_e, p, 0\rangle\ e, \langle c_e, p, 1\rangle\ e, \Gamma(\$g)\ \mathsf{root}) \mapsto \varphi],\\ &\quad\ \{\langle c_e, p, 0\rangle, \langle c_e, p, 1\rangle, \Gamma(\$g)\}, \quad \langle c_e, p, 0\rangle,\\ &\quad\ \{\}, \quad \{\ (\{\Gamma(\$g)\}, \Gamma(\$g), \psi[\mathsf{root}])\ \} \quad)\end{aligned}$$

We also add assumption formula here. Obligation formulas are generated in outside of this expression.

Variable Reference (Innermost Scope).

Difference of variables and markers is that the type of variable can be context-dependent. Consider the expression **if** $\$l = \texttt{contact}$ **then** $\$g :: \psi_1$ **else** $\{\$l : \$g :: \psi_2\}$. To generate obligations for the annotation $:: \psi_1$, it must take into account that the expression is under the branching by **if**. In this case, $\$g$ must have conform to ψ_1 only when $\$l = \texttt{contact}$. To incorporate the information, we use the third parameter φ of ft2mso containing the conditions of translated **if** branches.

$$\begin{aligned}&\text{ft2mso}(c, \Gamma, \varphi)(\$g :: \psi^p) =\\ &\quad (\ \mathcal{M}_p[(\varepsilon, \langle c_e, p, 0\rangle\ e, \langle c_e, p, 1\rangle\ e, \Gamma(\$g)\ \mathsf{root}) \mapsto \varphi],\\ &\quad\ \{\langle c_e, p, 0\rangle, \langle c_e, p, 1\rangle, \Gamma(\$g)\}, \quad \langle c_e, p, 0\rangle,\\ &\quad\ \{\ (J_0, c_u, \forall^f v, e, u.\ (\varphi \rightarrow \psi[u]))\ \},\\ &\quad\ \{\ (\{\Gamma(\$g)\}, \Gamma(\$g), \psi[\mathsf{root}])\ \} \quad)\end{aligned}$$

where J_0 is the set of copy-ids of the argument graph of the **rec** expression introduced the variable $\$g$, which is computed while ft2mso processes the **rec** expression.

Structural Recursion.

The rule for recursion is the most complicated one. The difficulty here is how to map the structural recursion of UnCAL that iteratively walk through graphs to an MSO formula that declaratively represents a relationship between input and output graphs. This problem is addressed by exploiting an alternative semantics called *bulk semantics* [7] of UnCAL, which more fits to logical formulation, and known to be equivalent to the usual recursive semantics.

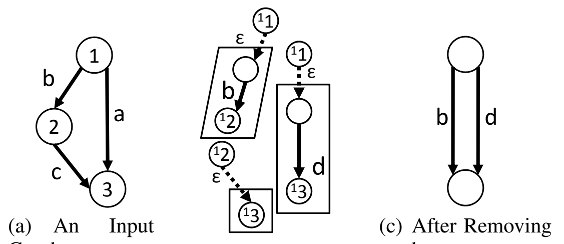
In bulk semantics, the structural recursion $\textbf{rec}(\lambda(\$l, \$g).\ \&_1 := \tau_1, \ldots, \&_n := \tau_n)(\tau_0)$ is evaluated as follows: first evaluate τ_0 and obtain the argument graph, and then, for every non-ε edge (v, l, u) of it, evaluate each τ_i separately under the environment $\{\$l \mapsto l, \$g \mapsto u\}$. After that, the output marker expression $\&_j$ (if any) in τ_i is connected to the root nodes of the result graphs of the evaluation of τ_j at the edges having u as their source node. Formally, the expression $\textbf{rec}(\lambda(\$l, \$g).\ \&_1 := \tau_1, \ldots, \&_n := \tau_n)(\tau_0)$ is evaluated as follows. First, evaluate τ_0 and obtain a graph $g_0 = (V, r, E)$. Then, generate n new nodes from ${}^1v$ to ${}^nv$ for each node $v \in V$, each corresponding to the marker $\&_i$. Then for each edge $p = (v, l, u)$ starting from v, we evaluate each body expression τ_i to obtain a graph $g_{p,i}$. If $l = \varepsilon$, we let $g_{p,i} = (\{{}^iv, {}^iu\}, {}^iv, \{({}^iv, \varepsilon, {}^iu)\})$, i.e., ε-edges are always kept unchanged. If $l \neq \varepsilon$, evaluate τ_i under the environment $\{\$l \mapsto l, \$g \mapsto u, \&_1 \mapsto {}^1u, \ldots, \&_n \mapsto {}^nu\}$ and get $g'_{p,i} = (V', r', E')$. Then we let $g_{p,i} = (V_{p,i}, r_{p,i}, E_{p,i}) = (V' \cup \{{}^iv\}, {}^iv, E' \cup \{({}^iv, \varepsilon, r')\})$, making ${}^iv$ the new root node[3]. The result graph g of the evaluation of the whole expression is the simple aggregation $g = (\bigcup_{p,i} V_{p,i}, {}^1r, \bigcup_{p,i} E_{p,i})$ of all the graphs $g_{p,i}$, making the $\&_1$ output at the root node in the input graph as the root node of the output.

The behavior is illustrated in Fig. 6. Recall the structural recursion *a2d_xc* defined in Sec. 3.2. Applying it to the input graph in Fig. 6(a) yields the graph in Fig. 6(b). The body of the recursion is applied to each of the three edges in the input graph and we obtain three graphs illustrated in the boxes. Then, new root nodes ${}^iv$ are added. Although depicted separately, the two ${}^1i$ nodes for each i denotes the same node and hence glued together. If we eliminate all ε-edges, we obtain a standard graph in Fig. 6(c).

Compared to the recursive interpretation, this bulk semantics rather naturally translates to our logic-based formulation as follows. For each edge (represented by $c' \in J_0 \times J_0 \times J_0$), we evaluate bodies e' and glue them together by simply taking union.

$\text{ft2mso}(c, \Gamma, \varphi)($

[3] This ε-edge introduction will be implicit in the example and depicted as if we unified r' and ${}^iv$



(a) An Input Graph (b) Before Removing ε-edges (c) After Removing ε-edges

Figure 6: Bulk Semantics of Structural Recursion in UnCAL

$$
\begin{aligned}
&\mathbf{rec}(\lambda(\$l, \$g :: \varphi_0).\&_1 :: \varphi_1 := \tau_1, \ldots, \&_n :: \varphi_n := \tau_n)(\tau_0)^p) = \\
&\quad\Big(\ \mathcal{M}_p \cup \mathcal{M}_0 \cup \bigcup_{i,c'} \mathcal{M}_i^{c'}, \quad \{p\}, \quad p, \\
&\qquad O_\$ \cup \bigcup_{1 \le i \le n} O_{\&_i} \cup O_0 \cup \bigcup_{i,c'} O_i^{c'}, \\
&\qquad A_p \cup A_0 \cup \bigcup_{i,c'} A_i^{c'} \quad \Big) \\
&\mathbf{where}(\mathcal{M}_0, J_0, \rho_0, O_0, A_0) = \mathrm{ft2mso}(c, \Gamma, \varphi)(\tau_0) \\
&\quad (\mathcal{M}_i^{c'}, J_i^{c'}, \rho_i^{c'}, O_i^{c'}, A_i^{c'}) = \mathrm{ft2mso}(c', \\
&\qquad\qquad \Gamma[\$g \mapsto \langle c_\mathrm{e}, p, 0\rangle, \&_1 \mapsto \langle c_\mathrm{e}, p, 1\rangle, \\
&\qquad\qquad\quad \ldots \&_n \mapsto \langle c_\mathrm{e}, p, n\rangle], \mathsf{true})(\tau_i) \\
&\qquad\qquad \text{for each } 1 \le i \le n, c' \in J_0 \times J_0 \times J_0 \\
&\quad O_\$ = \{\ (J_0, c_\mathrm{u}, \forall^f v, e, u.\ (\varphi \to \varphi_0[u]))\ |\ \$g' :: \psi \text{ occurs} \\
&\qquad\quad \text{in some } \tau_i \text{ for the current innermost scope variable } \$g'\ \} \\
&\quad O_{\&_i} = \{\ (J_i^{c'}, \rho_i^{c'}, \varphi_i[\mathsf{root}])\ \} \\
&\quad A_p = \{\ (\{p\}, p, \varphi_1^*[\mathsf{root}])\ \}
\end{aligned}
$$

Still, quite a few things must be taken into account. First, we need to generate obligation formulas for the current innermost scope variable, if it is used inside the body of this recursion. Second, we need to generate obligation formulas for markers. Thirdly, we need to add an assumption formula that the result of the recursion conforms to the schema φ_1^*; where φ_1^* representing a set of graphs consisting of unions of graphs satisfying φ_1. To be concrete, it is $\bigwedge_{\mathsf{a} \in Label_\varepsilon} \forall^f e\mathbf{edge}_\mathsf{a}(x, e, y), \exists^s Z(x, e, y, \mathsf{root} \in Z) \wedge \psi_Z)$ where ψ_Z is a restriction of second-order quantification into Z.

5.4 Relaxing the Annotation Burden

In the previous section, we have treated variables $\$g$, markers $\&_i$, and **rec** expressions as something *opaque*. That is, they are assigned new copy-ids and treated as an arbitrary graph that satisfies the annotated schema.

This can be made *transparent* in many situations. For instance in $\mathbf{rec}(\lambda(\$l, \$g :: \psi).\{\mathtt{foo} : \$g\})$, the destination node of the `foo` edge is not an arbitrary graph of type ψ, but it is *the* destination of the currently processed edge, whose copy-id is determined during the translation by ft2mso. In such cases, no annotation is required because our verifier can automatically connect the appropriate nodes and complete the structure information of such variables.

In the following three cases, annotations can be removed: (1) annotation $\$g :: \varphi$ to the innermost scope variables can always be omitted (2) annotation $\&_i :: \varphi$ for markers with $i \ge 2$ can always be omitted (3) annotation $\&_1 :: \varphi$ for the 1st marker of the recursion can be omitted if no other annotations are used inside the structural recursion. In particular, if the transformation never uses nested used of recursion variables, no annotation for intermediate graphs is required to verify the correctness. Programmers just need to specify the intended schema for the input graph $\$db$ and the output graph (i.e., result of the whole expression), our verifier can convert the UnCAL expression into MSO formula fully automatically.

Here is the excerpt of the no-annotation version of ft2mso for the case of structural recursion.

$$
\begin{aligned}
&\mathrm{ft2mso\_na}(c, \Gamma, \varphi)(\\
&\quad \mathbf{rec}(\lambda(\$l, \$g).\&_1 := \tau_1, \ldots, \&_n := \tau_n)(\tau_0)^p) = \\
&\qquad \Big(\ (\mathcal{M}_0 \cup \bigcup_{i,c'} \mathcal{M}_i^{c'})[\\
&\qquad\qquad (\varepsilon, \langle c'_\mathrm{v}, p, 2i-1\rangle v, \langle c'_\mathrm{v}, p, 2i-2\rangle v, \rho_i^{c'} e) \mapsto \varphi], \\
&\qquad\quad J_0 \cup \bigcup_{i,c'} J_i^{c'} \cup \{\langle c_\mathrm{v}, p, x\rangle \mid x < 2n\},\ \langle c_\mathrm{v}, p, 0\rangle\Big) \\
&\quad \mathbf{where}(\mathcal{M}_0, J_0, \rho_0) = \mathrm{ft2mso\_na}(c, \Gamma, \varphi)(\tau_0) \\
&\qquad (\mathcal{M}_i^{c'}, J_i^{c'}, \rho_i^{c'}) = \mathrm{ft2mso\_na}(c', \\
&\qquad\qquad \Gamma[\$g \mapsto c'_\mathrm{u}, \&_1 \mapsto \langle c'_\mathrm{u}, p, 1\rangle], \\
&\qquad\qquad\quad \ldots \&_n \mapsto \langle c'_\mathrm{u}, p, 2n-1\rangle], \mathsf{true})(\tau_i) \\
&\qquad\qquad \text{for each } 1 \le i \le n, c' \in J_0 \times J_0 \times J_0
\end{aligned}
$$

The difference is, for instance, in the translation of subformulas τ_i, $\$g$ is now bound to c_u, which is exactly the copy-id of the destination node of the focused edge c' and is not the newly generated fresh id $\langle c_\mathrm{e}, p, 0\rangle$ as in the type-annotated version. Or, $\&_i$ is bound to $\langle c'_\mathrm{u}, p, 2i-1\rangle$, the $(2i-1)$-th copy of the destination node, which, in the definition $\langle c'_\mathrm{v}, p, 2i-1\rangle v, \langle c'_\mathrm{v}, p, 2i-2\rangle v, \rho_i^{c'} e)$ of the output transduction system, is declared to be connected to the root node $\rho_i^{c'}$ of the transformation result of the destination node.

6. DECISION PROCEDURE

The verification problem of annotated Core UnCAL is now reduced to the problem of validity of a closed MSO formula. This, however, is not a trivial task. Even for the first-order logic, validity of a formula is well-known to be undecidable on general graph structures [27]. Even worse, expressing schemas in logic usually requires involved features like transitive-closures (e.g., to ignore ϵ-edges) that go beyond first-order logic.

Nevertheless, we can avoid the undecidability thanks to the nice property of UnCAL, namely, the bisimulation-genericity. We prove that the MSO formula obtained by the previous section is not valid on some graph if and only if it is not satisfied on some (possibly infinite) tree, on which decidability is known in the literature. Furthermore, a vast range of UnCAL transformations falls into a category called *compact* transformations [7]. For this class of transformations, we can show that there must be a finite-tree counterexample if there are any counterexamples. The property is important for efficient implementation.

6.1 Reduced to Infinite Tree Model

To decide the validity of a bisimulation-generic formula, we only need to consider some representatives of bisimilar graphs. Formally speaking, the following lemma holds.

LEMMA 2. *Let b be a function from graphs to graphs such that $g \equiv b(g)$ for any g. Let φ be a bisimulation-generic formula. Then, the claim "$g \models \varphi$ for any graph g" holds if and only if "$g \models \varphi$ for any graph g in range of b".*

PROOF. The 'only if' direction is trivial. For the 'if' direction, $g \vDash \varphi$ equals $b(g) \vDash \varphi$ by the bisimulation-genericity of φ and the latter holds because $b(g)$ is surely in the range of b. □

By taking the representative function b as the infinite unfolding function, we can focus the range of g on infinite trees rather than arbitrary graphs. Fortunately, there is an effective procedure to check the satisfiability or validity of MSO on infinite trees [22].

THEOREM 2. *Verification problem is decidable.*

The proof of the decidability resorts to the decidability of emptiness of automata. Since the emptiness test procedure easily exhibits a way to produce a counterexample in a nonempty case, our approach can generate a counterexample to the UnCAL verification problem in the case of failure.

6.2 Reduced to Finite Tree Model

Graph transformations are called *positive* if they do not use isEmpty expression that checks whether or not a node has any outgoing edge. Many useful transformations fall into this category. In the appendix of [7], a positive transformation is shown to have a property called *compactness*, by which we can reduce the problem on infinite trees to finite trees.

To formalize the notion of compactness, let us first introduce the operation cut. For trees $T_1 = (V_1, r_1, E_1)$ and $T_2 = (V_2, r_2, E_2)$, we define the prefix relation $T_1 \preceq T_2$ to hold when there is a one-to-one mapping e from V_1 to V_2 such that $e(r_1) = r_2$ and $(v_1, l, u_1) \in E_1$ iff $(e(v_1), l, e(u_1)) \in E_2$. For a possibly infinite tree T, the set of its *finite-cuts* is $cut(T) = \{t \mid t \preceq T, t \text{ is finite}\}$. For instance, the finite-cuts of an infinite tree $cut(\circ \xrightarrow{a} \bullet \xrightarrow{a} \bullet \xrightarrow{a} \bullet \cdots)$ are infinitely many finite trees $\{\circ, \circ \xrightarrow{a} \bullet, \circ \xrightarrow{a} \bullet \xrightarrow{a} \bullet, \ldots\}$.

A set C is said to *cover* T if it is a subset of $cut(T)$ and for any $t \in cut(T)$ there exists $t_c \in C$ such that $t \preceq t_c$. Intuitively, $t \preceq t'$ means that t' contains more information on the original tree T than t. When C covers T, it roughly means that C has enough information to recover T. The following property of positive UnCAL is called compactness. It means that instead of transforming an infinite tree T, we only need to transform each finite-cut for obtaining enough information to construct $f(T)$.

LEMMA 3 ([7], PROPOSITION 8). *Let T be a possibly infinite tree and f be a positive UnCAL transformation. Then, $\{unfold(f(t)) \mid t \in cut(T)\}$ covers $unfold(f(T))$.*

We can extend the notion of compactness to schemas. A schema φ is called compact if for any tree T: (1) $T \vDash \varphi$ implies $t \vDash \varphi$ for all $t \in cut(T)$, and (2) if there exists a set $C \subseteq \{t \mid t \vDash \varphi\}$ that covers T, we have $T \vDash \varphi$. When both schemas and transformation are compact, validity on infinite trees can be checked by testing only on finite trees.

THEOREM 3. *If the schemas are compact and the transformation is positive, the verification problem is reducible to the validity of MSO on finite trees.*

For the detail of the proof of Theorem 3, refer to our technical report [17].

Decidability of MSO on finite trees[4] is proved in [24] by much simpler manner than the infinite case. Indeed, this simplicity is important for having more efficient implementation of the verifier. For MSO on finite trees, there exists a good practical implementation MONA [14], whose efficiency is verified in many applications. Our current prototype is implemented using MONA, leaving the infinite case as future work.

[4]Here we mean by MSO on finite trees what is called weak MSO (WSkS) in the literature. Precisely speaking, it is MSO on the *infinite* k-ary tree domain with no node/edge-labels, whose second-order variables can range over *finite sets* only. Since the finiteness restriction prohibits us to encode infinitely many labeled-edges, we call it MSO on finite trees. Similarly, we mention MSO on the infinite k-ary tree with no restriction (SkS) as MSO on infinite trees.

7. RELATED WORK

In the original paper [7], the logical characterization of UnCAL is given using first-order logic with transitive closures (FO+TC) by showing the logic captures the full expressive power of UnCAL. The problem is that the validity of FO+TC formula is undecidable [25] even on finite trees. Hence, naïvely reducing the problem to FO+TC can only derive either unsound, incomplete, or possibly non-terminating verification algorithms. Rather, our approach is to start from a decidable logic (namely, MSO on trees) capturing some clearly defined fragment of UnCAL, and provide sound and terminating verification algorithm for the fragment, which we hope to be a solid basis towards the complete verification of full UnCAL.

Concerning the choice of logic, in [18], it has been shown that the bisimulation-generic subset of MSO is equivalent in expressiveness to the modal μ-calculus. This suggests that we can use μ-calculus in place of MSO. The problem is, however, there is no established method to represent *transformations* in μ-calculus, Different from predicate logics, there is no way to denote each node or edge individually in μ-calculus, which makes it hard to describe a translation in terms of things like **edge** predicates as in MSO-definable transduction. Nonetheless, if we could overcome the problem, the worst-case EXPTIME complexity of validity of μ-calculus is an attractive candidate regarding the non-elementary complexity upperbound of MSO.

Another group of related work on verification of transformations can be found in the area of XML processing, under the name *exact typechecking* [26, 21, 20, 12]. The main tool there to represent transformations is what is called a tree transducer, a kind of functional programming language. Our approach to construct the inverse image $f^{-1}(\varphi_{\text{OUT}})$ of the output-schema follows the same way as those researches on XML typechecking. Advantage of MSO-definable transduction over tree transducers is, (1) it is straightforward to generalize the notion from trees to graphs, and (2) composition (in UnCAL terminology, **rec** expression inside the argument of another **rec** expression) of transformations can be relatively easily handled. In tree transducers, the number h of composition makes the complexity of typechecking very high, namely, h-exponential (and hence recent work [20, 12] targets a single, non-compositional transducers). While in MSO, it stays single exponential. Note, however, some variants of tree transducers have higher expressiveness that allows to represent nested-recursion without annotations. It is our future work to combine those two approaches and seek a balancing point of complexity and expressiveness.

Unno et al. [28] proposes a verification method for tree processing programs using higher-order macro tree transducers utilizing annotations. Since their method can be applied to infinite-trees, it can also handle bisimulation-generic graph transformations. Compared to our method, the places for required annotations are different. Theirs does not require annotation for nested occurrence of variables (which is needed in our approach), while it requires for compositions (or generally, re-consumption of a temporarily created trees), which is not needed in ours.

Finally, the simulation-based schema [5] compared with MSO in Introduction still has some advantage over our MSO-based approach. Although it is weak for representing structural properties of graphs, it is easily adopted to express properties on data values, because its schema can have unary predicates putting constraints on data edges (like, "it must match some regular expression"), which is left as future work for our approach.

8. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a new approach to verifying graph transformations written in the Core UnCAL against the specified input/output graph schemas in MSO. We show that the Core UnCAL can be represented as an MSO-definable graph transduction, where not only schemas but also transformations are described by MSO formula, and efficiently implemented with MONA [14]. Our verifier can deal with any graph transformation in the type-annotated Core UnCAL, and more advanced structural properties like "either-or" compared to existing simulation-based checking algorithm. Furthermore, when the transformation failed against the verification, our verifier can produce a understandable counterexample with respect to the input rather than the output.

Future plan is to support data values and to broaden the verifiable transformations. Firstly, unary predicates on data values such as a test of the range of integer values or the length of string data can be rather easily incorporated into our framework, by basically regarding them as a normal label, but conformance to a schema is tested by logical subsumption. As long as the conditions are written in decidable logic, the conformance can be decided. Then, for binary or more complex predicates such as asserting that two data values must always be equal, we plan to extend our approach by using a *nondeterministic* MSO-definable transduction and approximate complex branches by a nondeterministic choice. This technique is already used in verification of XML-transformations (see, e.g., [21]).

9. REFERENCES

[1] S. Abiteboul, D. Quass, J. Mchugh, J. Widom, and J. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1:68–88, 1997.

[2] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40:1:1–1:39, February 2008.

[3] ATLAS group. KM3 manual. `http://www.eclipse.org/gmt/atl/doc/`.

[4] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. Technical Report MS-CIS-96-21, Univ. of Pennsylvania, 1996.

[5] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *ICDT*, pages 336–350, 1997.

[6] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM SIGMOD international conference on Management of Data*, pages 505–516. ACM, 1996.

[7] P. Buneman, M. F. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB Journal*, 9(1):76–110, 2000.

[8] J. Clark and M. Murata. RELAX NG specification. `http://www.relaxng.org/`, 2001.

[9] M. P. Consens and A. O. Mendelzon. Graphlog: a visual formalism for real life recursion. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '90, pages 404–416, New York, NY, USA, 1990. ACM.

[10] B. Courcelle. Monadic second-order definable graph transductions: A survey. *Theoretical Computer Science*, 126(1):53–75, 1994.

[11] DTD: Document Type Definition. `http://www.w3.org/XML/1998/06/xmlspec-report.htm`.

[12] A. Frisch and H. Hosoya. Towards practical typechecking for macro tree transducers. In *DBPL*, pages 246–260, 2007.

[13] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 30:444–491, 2005.

[14] J. G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS*, pages 89–110, 1995.

[15] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *ICFP*, 2010.

[16] S. Hidaka, Z. Hu, H. Kato, and K. Nakano. Towards a compositional approach to model transformation for software development. In *SAC*, pages 468–475, 2009.

[17] K. Inaba, S. Hidaka, Z. Hu, H. Kato, and K. Nakano. Sound and complete validation of graph transformations. Technical Report GRACE-TR-2010-04, GRACE Center, NII, 2010.

[18] D. Janin and I. Walukiewicz. On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In *CONCUR*, pages 263–277, 1996.

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

[20] S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In *ICDT*, pages 254–268, 2007.

[21] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *J. Comp. Syst. Sci.*, 66:66–97, 2003.

[22] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of American Mathematical Society*, 141:1–35, 1969.

[23] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[24] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2:57–81, 1968.

[25] H.-J. Tiede and S. Kepser. Monadic second-order logic and transitive closure logics over trees. In *WoLLIC*, pages 189–199, 2006.

[26] A. Tozawa. Towards static type checking for XSLT. In *DocEng*, pages 18–27, 2001.

[27] B. A. Trakhtenbrot. Impossibility of an algorithm for the decision problem for finite classes. *Doklady Akademiia Nauk SSSR*, 70:569–572, 1950.

[28] H. Unno, N. Tabuchi, and N. Kobayashi. Verification of tree-processing program via higher-order model checking. In *Asian Symposium on Programming Languages and Systems (APLAS)*, 2010.

[29] W3C XML Schema WG. W3C XML Schema. `http://www.w3c.org/XML/Schema`.