ISSN 1884-0760

GRACE TECHNICAL REPORTS

Sound and Complete Validation of Graph Transformations

Kazuhiro Inaba Soichiro Hidaka Zhenjiang Hu
Hiroyuki Kato Keisuke Nakano

GRACE-TR-2010-04 May 2010

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

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

Sound and Complete Validation of Graph Transformations

Kazuhiro Inaba Soichiro Hidaka Zhenjiang Hu Hiroyuki Kato
National Institute of Informatics
{kinaba,hidaka,hu,kato}@nii.ac.jp

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

May 6th, 2010

Abstract

Transformation of graph structures is becoming more and more important in many fields such as semistructured database or model-driven software development. There, graphs are often associated with schemas that describe *structural constraints* on the graphs. In this paper, we present a static validation algorithm for the core fragment of a graph transformation language UnCAL [7]. Given a transformation and input/output schemas, our algorithm statically verifies that any graph satisfying the input schema is converted to a graph satisfying the output schema.

Our algorithm is enabled by reformulating the semantics of the core UnCAL, using *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. Furthermore, with several insights on the established properties of UnCAL, the problem turns out to be reducible to the validity of MSO over *finite trees*, which have sound and complete decision procedure.

1 Introduction

Transformation of graph structures is becoming important in many fields [7, 10, 18, 3]. For instance, in semistructured database [7], data sources are represented as graphs and therefore queries on the database become graph transformation. In model-driven software development [10], software components in different level of abstraction are modeled as graphs, and their relation is described as executable graph transformations.

In these applications, we often assume, for each graph transformation, that its input and output graphs are not arbitrary graphs but have some structure in it. Let us consider, say, a graph transformation that extracts a list of "person names" from a graph-formed database of an "address book". Input graphs for such a query are assumed to have, e.g., a root node having a bunch of outgoing edges labeled `person`, each pointing to a node with edges `name`, `address`, `phoneNo`, etc. Similarly, for an input graph satisfying such structural constraints, we expect the transformation to return an output graph with a set of `name` edges. Such constraints on the structure of input/output graphs are expressed by some *graph schema language*.

Sometimes, graph transformations written by programmers contain bugs that break these structural constraints imposed on the transformations. For instance, instead of generating a set of `name` edges, programmer may write a transformation that produces a set of `name` edges each preceded by a `person` edge, forgetting to erase the parent edge. It is relatively easy to check such bugs dynamically; for each run of the graph transformation, we can check the conformance of the concrete input/output graphs to the specified schemas. A question arises here is, whether it is possible to ensure beforehand that such structure-breaking bugs can *never* happen? Dynamic check is not satisfactory, because it only checks the correctness for particular given input graphs.

The objective of this paper is to answer the question affirmatively, by providing a *static* validation algorithm of a practical graph transformation language. The problem we would like to verify is the following one:

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

More specifically, we present the validation algorithm for the core fragment of UnCAL graph algebra, which is first introduced as the basis of a graph query language UnQL for unstructured database [6] and later applied to semistructured database [7], and is recently applied to model-driven software development [15]. Our validation is sound, i.e., we are able to know statically that a validated transformation never produces ill-formed output. Furthermore, it is decidable and complete; the validation process always terminates without any false alarm.

Main difficulty of the validation of graph transformations is that it looks very close to undecidable problems. The largest problem resides in the "for any graph" part of the validation problem; first-order properties are well-known to become undecidable [26] on graphs, and even worse, precisely expressing schemas and translation languages in logic usually require involved features like transitive-closures which go beyond first-order logic. Widely adopted compromise for such situation on graphs is *tree-decomposition* [22]. By restricting the set of graphs in consideration to the tree-decomposable

ones (i.e., graphs whose sharing and cycles are limited to some constant distance), essentially the validation problem is reduced to that on trees. Unfortunately, we cannot follow this approach for two reasons. Firstly, since our purpose is to validate transformation programs for more general graphs, limiting the input domain to almost-tree graphs does not make sense. Secondly, the original semantics of UnCAL is given in the first-order logic extended with transitive closures, whose validity is known to be undecidable even on finite trees [24].

In order to overcome the difficulty, our approach is different from the traditional tree-decomposition based method, though the spirit is a little similar as we also reduce the problem on graphs to trees. We focus on the fact that UnCAL transformations are well-structured by *structural recursion* that always quite uniformly traverses over argument graphs. The structural-recursion-based nature of UnCAL enables to derive two nice properties called *bisimulation-genericity* and *compactness*, as shown in [7]. To put it plainly, by exploiting these properties, we prove that if a schema-violation occurs then it must occur within the finite unfolding of the graph. Hence, for the purpose of checking schema conformance, we only need to concentrate on such finite prefixes (called finite-cuts) of graphs. Furthermore, we have found out that for the core fragment of UnCAL under consideration, we can give an alternative presentation of its semantics by using monadic second-order logic (MSO), which is known to be decidable on finite trees [21]. The core UnCAL itself is expressible enough to capture basic subgraph extractions and relabeling/restructuring along the structure of the original input graphs, and we believe that it is a good starting point for constructing a decidable validation algorithm for the full UnCAL.

In summary, our approach for the validation problem consists of three steps. First, we show the validation problem for transformations over graphs can be reduced to the problem over finite trees. Since the reduction is sound and complete, deciding the latter problem solve the validation problem of graph transformations. To make it clear what has enabled the reduction, in this paper we further split this step into two. We show the bisimulation-genericity of our schemas, which, together with the existing result of the bisimulation-genericity of UnCAL, allows to reduce the validation problem to possibly infinite trees. Then, we utilize the compactness that allows to reduce the problem into finite trees. In the second step, we convert the schema and UnCAL transformation into a single MSO logic formula. The formula is valid (i.e., true on any finite trees) if and only if the translation is valid with respect to the schema. Thus in the third step, we determine the validity of the formula by known decision algorithm for MSO on finite trees.

Outline The paper is organized as follows. Section 2 explains the graph transformation language UnCAL and our schema language, which are the target languages of our validation technique. The subsequent two sections discuss how the validation problem on arbitrary graphs can be reduced to that on infinite trees (Section 3), and eventually to that on finite trees (Section 4). In Section 5, the validation problem is shown to be expressible in MSO over finite trees. Since the logic is known to be decidable, the validation problem is proven to be decidable at this point. Section 6 shows related work, and Section 7 concludes and presents future direction of the research.

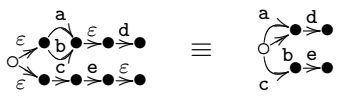
2 Languages

In this section, we introduce two languages concerning our validation technique. One is for describing graph transformations: a core fragment of UnCAL graph algebra [7]. The other is a schema language for describing structural properties of graphs.

2.1 Graph Data Model

We deal 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 $\varepsilon \notin Label$, and denote by $Label_\varepsilon$ the set $Label \cup \{\varepsilon\}$. 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 $g = (V, E, r)$ consists of a set V of *nodes* (sometimes called *vertices*), a function E from V to a finite set of *edges*, and a designated *root node* $r \in V$. Here, an edge is a pair in the set $(Label_\varepsilon \cup Data) \times V$; the first component of each edge is the information conveyed by the edge, and the second component is the destination node of the edge. A graph without any sharing (multiple edges pointing to the same destination node) and any cycles is called a *tree*.

Notable feature of the UnCAL's graph model is that it has ε-edges resembling ε-transitions of automata, which work as shortcuts between nodes. Schemas and transformations will be defined to respect this intuitive meaning 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.

$$
\begin{array}{lcl}
\mathit{Schema} & ::= & \textbf{roottype}\ \mathit{Type}\ \textbf{where}\ \mathit{Decl} \cdots \mathit{Decl} \\
\mathit{Decl} & ::= & \mathit{Name} = \{\mathit{Edge}, \ldots, \mathit{Edge}\} \\
 & | & \mathit{Name} = \{\mathit{Edge}, \ldots, \mathit{Edge}, *\} \\
\mathit{Type} & ::= & \mathit{Name} \quad | \quad \mathsf{Data} \quad | \quad \mathit{Type} \mid \mathit{Type}
\end{array}
$$

Figure 1: Graph Schema Language $\mathcal{GS}$

For instance, we do not need a union operator $e_1 \cup e_2$ of two edge-sets explicitly, because it can be simulated by a construction $\{\varepsilon : e_1, \varepsilon : e_2\}$ of a new node having two ε-edges, as exemplified by the root node of the figure above.

Formally, we define the set $E^{\rightarrow}(v)$ of *outgoing edges* of a node v as the set of non-ε edges reachable from v by traversing only ε-edges. That is, $(l, u) \in E^{\rightarrow}(v)$ if and only if $l \neq \varepsilon$ and there exists a sequence $v = v_0, v_1, \ldots, v_k$ of nodes with $(\varepsilon, v_i) \in E(v_{i-1})$ for $i > 0$ and $(l, u) \in E(v_k)$.

2.2 Schema Language

A schema describes a restriction to the structure of graphs. For example, one can state that all the outgoing edges for the designated root node must be labeled `abc`, and each of the destination nodes of the edges may have edges labeled `xyz` going to the same type of nodes, and several other edges. This claim on the structure of graphs can be stated in our schema language as follows:

$$
\textbf{roottype}\ \mathsf{T}\ \textbf{where}\ \mathsf{T} = \{\mathtt{abc} : \mathsf{S}\}\ \mathsf{S} = \{\mathtt{xyz} : \mathsf{S}, *\}.
$$

The schema language, named $\mathcal{GS}$, has the most similarity with the simulation-based graph schema proposed in [5] for UnCAL, but $\mathcal{GS}$ is more inclined for describing the structural properties of graphs. The difference will be discussed in detail in Section 6.

Figure 1 defines the syntax of $\mathcal{GS}$, where *Name* is a set of *type names* whose elements are written by san-serif symbols like Apple, and Data is a special type name for *Data* edges. We require a schema to be well-formed, i.e., every *Name* in a schema occurs exactly once as a left-hand side of a *Decl*, and in each *Decl*, there are no duplicate *Label*s in the right-hand side. Let us explain the idea of each construct by using the following example:

$$
\begin{array}{rl}
\multicolumn{2}{l}{\textbf{roottype}\ \mathsf{SNS}\ \textbf{where}} \\
\mathsf{SNS} = & \{\mathtt{member} : \mathsf{Person}\} \\
\mathsf{Person} = & \{\mathtt{name} : \mathsf{Data} \mid \mathsf{Name},\ \mathtt{email} : \mathsf{Data}, \\
 & \quad \mathtt{friend} : \mathsf{Person},\ *\} \\
\mathsf{Name} = & \{\mathtt{first} : \mathsf{Data},\ \mathtt{family} : \mathsf{Data},\ \mathtt{middle} : \mathsf{Data}\}
\end{array}
$$

The schema describes structural properties of the set of graphs representing the user-network of a social networking service. According to the schema, the root node must have type SNS, that is, all outgoing edges must be labeled `member` and reach to nodes typable with the Person type. At this point, we only consider the case where the number of the edges is arbitrary; the extension adding cardinality constraints to schema will be discussed later as future work. For a node to have the type Person, its outgoing edges labeled `name` have their destination nodes of type Data | Name, meaning that it must be typed by either one of the types Data (merely having a string representation of one's full name) or Name (storing the name in more structured way). Similarly, outgoing edges of a node of type Person with label `email` must have Data destination nodes, and so on. Since the type definition ends with $*$, it can also have extra edges of other labels with no constraints. Note also that the destination type of `friend` edges are again Person itself; this implies that instances of the schema may contain cycles.

Formally, for a schema s written in $\mathcal{GS}$, we let $\mathit{rtype}(s)$ be the root type of s, $\mathit{tname}(s)$ the set of type names appearing in s, $\mathit{tdecl}_s(\tau)$ the corresponding body b such that the declaration $\tau = b$ is in s, and $ns(\rho) = \{\tau_1, \ldots, \tau_n\} \subseteq \mathit{tname}(s)$ for a type $\rho = \tau_1 \mid \cdots \mid \tau_n$. The set $[\![s]\!]$ of graphs *satisfying* the schema s consists of graphs $g = (V, E, r)$ such that there exists a mapping $m : V \to 2^{\mathit{tname}(s) \cup \{\mathsf{Data}\}}$ with the following properties:

1. $r \lhd_m \mathit{rtype}(s)$ (where $v \lhd_m \rho$ means $m(v) \cap ns(\rho) \neq \emptyset$ and is read as "v has type ρ").
2. For any node $v \in V$, having $\mathsf{Data} \in m(v)$ implies that for any $(l, u) \in E^{\rightarrow}(v)$ we have $l \in \mathit{Data}$.
3. For any node $v \in V$, having $\tau \in m(v)$ for $\tau \in \mathit{tname}(s)$ implies that $E^{\rightarrow}(v)$ satisfies $\mathit{tdecl}_s(\tau)$. Here the set of edges $E^{\rightarrow}(v)$ is defined to satisfy a type declaration $\mathit{tdecl}_s(\tau) = \{l_1 : \rho_1, \ldots, l_n : \rho_n\}$ if and only if for any edge (l, u) in $E^{\rightarrow}(v)$ the label l is equal to one of l_i and in that case $u \lhd_m \rho_i$ holds. When $\mathit{tdecl}_s(\tau)$ has a trailing star $\{\cdots *\}$, we require for any edge (l, u) in $E^{\rightarrow}(v)$ that the label l must either be equal to one of l_i and $u \lhd_m \rho_i$, or be equal to none of them.

For brevity, we sometimes abuse the notation and say that v satisfies $\mathit{tdecl}_s(\mathsf{Data})$ to mean the second condition to hold for the node v. Using the defined terminology, the validation problem can now be stated as the problem of determining the validity of the following proposition: "for any graph g, $g \in [\![s_{\mathrm{IN}}]\!]$ implies $f(g) \in [\![s_{\mathrm{OUT}}]\!]$".

2.3 Core UnCAL

The graph transformation language dealt with in this paper is, the *nest-free* and *positive* fragment of the UnCAL graph algebra, which we call the

$$
\begin{array}{rcll}
e & ::= & \{l : e, \ldots, l : e\} & \text{node with edges} \\
 & | & \$g & \text{variable reference} \\
 & | & \textbf{if } \$l = \mathtt{a} \textbf{ then } e \textbf{ else } e & \text{conditional } (\mathtt{a} \in \mathit{Label}) \\
 & | & \&_i & \text{output marker} \\
 & | & \textbf{rec}(\lambda(\$l, \$g).\ \&_1 := e, \ldots, \&_n := e)(e) & \text{structural recursion} \\
l & ::= & \$l & \text{label variable reference} \\
 & | & \mathtt{a} & \text{label } (\mathtt{a} \in \mathit{Label}_\varepsilon \cup \mathit{Data}).
\end{array}
$$

Figure 2: Core UnCAL Language

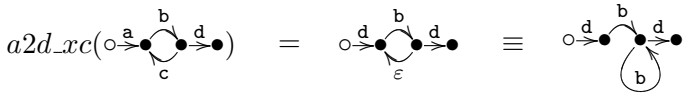
core UnCAL. The concrete syntax is shown in Figure 2. Several syntactic restrictions further applied to the core UnCAL are explained at the end of this section, with comparison to the full UnCAL.

We hope the intuition of the most of the constructs is clear for the reader. Node construction expression $\{l_1 : e_1, \ldots, l_n : e_n\}$ creates a fresh node v with outgoing edges $E(v) = \{(l_1, r_1), \ldots, (l_n, r_n)\}$ where r_i is the root node of the graph obtained by evaluating the expression e_i. Variable reference and conditional branch is defined as usual. The output marker expression $\&_i$ is used only in the body of **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*. The expression $\textbf{rec}(\lambda(\$l, \$g).\ \&_1 := e_1, \ldots, \&_n := e_n)(e_a)$ is evaluated as follows: first evaluate e_a and obtain the argument graph, and then, for every non-ε edge (l, v) of it, evaluate each e_i under the environment $\{\$l \mapsto l, \$g \mapsto v\}$. The output marker expression $\&_j$ (if any) in e_i is connected to the root nodes of the result graphs of the evaluation of e_j at the edges in $E(v)$.

Let us look at some examples. The following UnCAL expression *a2d_xc*

$$
\begin{array}{l}
\textbf{rec}(\lambda(\$l,\$g). \\
\quad \&_1 := \textbf{ if } \$l = \mathtt{a} \textbf{ then } \{\mathtt{d} : \&_1\} \\
\qquad\qquad \textbf{else if } \$l = \mathtt{c} \textbf{ then } \{\varepsilon : \&_1\} \textbf{ else } \{\$l : \&_1\})(\$db)
\end{array}
$$

replaces all labels `a` by `d` and shorts edges labeled `c` by changing them to ε as follows:

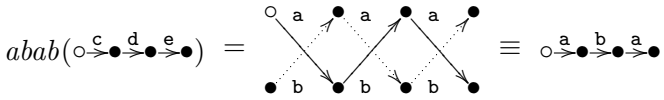


Here, $\$db$ is a designated variable referring to the input graph and $e(g)$ for any UnCAL expression e should be read as "evaluate e under the environment $\$db \mapsto g$".

More involved example is the following UnQL expression *abab*

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

that changes all edges of even distance from the root node to `a`, and odd distance edges to `b`. You may consider the markers $\&_i$ as a mutually recursive call, and the expression *abab* to be consisting of two mutual recursive functions. One is $\&_1$, which, at each edge in the original graph, generates a new `a` edge pointing to the result of $\&_2$ at the original destination node. Another is $\&_2$ that generates `b` edges pointing to the result of $\&_1$ from its destination. The result of the whole expression is defined to be the result of the $\&_1$ at the root node of the argument graph. The following figure should be illustrative. The dotted edges denote the edges unreachable from the output root node.



Formally, the expression $\mathbf{rec}(\lambda(\$l, \$g).\ \&_1 := e_1, \ldots, \&_n := e_n)(e_a)$ is evaluated as follows. First, evaluate e_a and obtain some graph $g_a = (V, E, r)$. Then, generate n new nodes ${}^1v$ from ${}^nv$ for each node $v \in V$, each corresponds to the marker $\&_i$. Then for each edge $p = (l, u) \in E(v)$ of $v \in V$, we evaluate each body expression e_i to obtain a graph $g_{p,i}$. If $l = \varepsilon$, we let $g_{p,i} = (\{{}^iv, {}^iu\}, \{{}^iv \mapsto \{(\varepsilon, {}^iu)\}\}, {}^iv)$, i.e., ε-edges are always kept unchanged. If $l \neq \varepsilon$, evaluate e_i under the environment $\{\$l \mapsto l, \$g \mapsto u, \&_1 \mapsto {}^1u, \ldots, \&_n \mapsto {}^nu\}$ and get $g'_{p,i} = (V', E', r')$. Then we let $g_{p,i} = (V_{p,i}, E_{p,i}, r_{p,i}) = (V' \cup \{{}^iv\}, E' \cup \{{}^iv \mapsto \{(\varepsilon, r')\}\}, {}^iv)$, making ${}^iv$ the new root node[1]. The result graph g of the evaluation of the whole expression is the simple aggregation $g = (\bigcup_{p,i} V_{p,i}, v \mapsto \bigcup_{p,i} E_{p,i}(v), {}^1r)$ 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.

Here is another more realistic example

$$\begin{aligned}&\mathbf{rec}(\lambda(\$l_1,\$g_1).\ \&_1 := \{\texttt{member} :\\ &\quad \mathbf{rec}(\lambda(\$l_2, \$g_2).\ \&_1 :=\\ &\qquad \mathbf{if}\ \$l_2 = \texttt{friend}\ \mathbf{then}\ \$g_2\ \mathbf{else}\ \{\})(\$g_1)\ \ \})(\$db)\end{aligned}$$

that extracts, from a graph satisfying the SNS schema, the set of members who are being friends of some other member.

The differences of the core UnCAL from the full UnCAL are threefold.

Nest-Free Core UnCAL prohibits nested **rec** to refer to outer variables, e.g., for a nested **rec** expression $\mathbf{rec}(\lambda(\$l_1, \$g_1).\cdots \mathbf{rec}(\lambda(\$l_2, \$g_2).\cdots \&_i := e_i \cdots)(\cdots)\cdots)(e)$, the inner body e_i can only use variables $\$l_2$ and $\$g_2$, not $\$l_1$ or $\$g_1$,

[1] This new root/ε-edge introduction was implicit in the preceding examples and depicted as if we unified r' and ${}^iv$

Positive Core UnCAL does not have **if** isEmpty($\$g$) **then** predicate to check whether the graph pointed by $\$g$ is empty or not.

Simplified Markers Uses of markers $\&_i$ are simplified. We require output markers $\&_i$ not to occur directly in the argument expression e_a in an expression $\mathbf{rec}(\cdots)(e_a)$; they can only appear in the body expressions of **rec**s (i.e., $\mathbf{rec}(\cdots\mathbf{rec}(\cdots)(\{\mathtt{a}:\&_1\})\cdots)(e)$ is not allowed due to $\&_1$ in the argument but $\mathbf{rec}(\cdots)(\mathbf{rec}(\cdots\{\mathtt{a}:\&_1\}\cdots)(e))$ is ok because it is wrapped in another **rec**). We also restrict the occurrence of input markers $\&_i{:=}$ only at the root of the body expression of **rec**. Besides, we have dropped the marker-connection operator @ of full UnCAL. In fact, the use of @ is implicit in the core UnCAL; the expression $\mathbf{rec}(\cdots)(\cdots)$ in the core UnCAL should be read as $\&_1 \mathbin{@} \mathbf{rec}(\cdots)(\cdots)$ in the full UnCAL.

Note that the first and the second constraints essentially lower the expressiveness, while the third simplification is not so, because all the UnCAL expressions compiled from its front-end language UnQL can easily be written in the form satisfying the third condition.

As a final remark, let us note one thing about the purpose of the UnCAL language. The reader may find it too primitive and not user-friendly; but this is rather intended. UnCAL is developed as the easy-to-reason-about internal algebra of a more human-friendly graph-query language called UnQL, in the same sense as that the well-known relational algebra is an internal language for the SQL querying language. The validation algorithm for UnCAL as will be presented in this paper can automatically be applied to the UnQL language, by first compiling UnQL to UnCAL and then running the validation algorithm. Roughly speaking, the restrictions of the core UnCAL correspond to the subset of UnQL queries that cannot take the join or the direct-product of multiple query results. Yet, the "core UnQL" is expressible enough to capture basic subgraph extractions (as shown in the previous example) and relabeling/restructuring along the structure of the original input graphs.

3 From Graphs to Infinite Trees

Recall that the validity of a proposition of the form "*for any graph* g, a property φ holds" referring to arbitrary graphs has no general decision procedure [26] even for some first-order expressive property φ. The validation problem we want to verify–at least if literally written–is in that form: "*for any graph* g, if it satisfies the input schema s_{IN}, the output $f(g)$ satisfies the output schema s_{OUT}". To avoid this obstacle, in this section, we decouple the reference to arbitrary graphs from the validation problem and reduce the

problem to that on infinite trees. The concept that plays the most important role here is what is called the *bisimulation*.

Definition 1. Graphs $g_1 = (V_1, E_1, r_1)$ and $g_2 = (V_2, E_2, r_2)$ are defined to be *bisimilar* and written $g_1 \equiv g_2$ if there exists a relation (called *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 $(l, u_1) \in E_1^{\rightarrow}(v_1)$, there exists u_2 such that $(l, u_2) \in E_2^{\rightarrow}(v_2)$ and $(u_1, u_2) \in S$, and (3) for all $(v_1, v_2) \in S$ and $(l, u_2) \in E_2^{\rightarrow}(v_2)$, there exists u_1 such that $(l, u_1) \in E_1^{\rightarrow}(v_1)$ and $(u_1, u_2) \in S$.

In fact, UnCAL is designed carefully to regard two graphs equal if they are *bisimilar*, in the sense that two bisimilar input graphs always generate again bisimilar output graphs. Graph transformations written in UnCAL are said to be *bisimulation-generic* in the sense that the following lemma holds.

Lemma 1 ([7], Proposition 4)**.** *For any transformation f written in UnCAL and any graphs g_1 and g_2, if $g_1 \equiv g_2$ then we have $f(g_1) \equiv f(g_2)$.*

Note that the lemma holds even for infinite graphs. Regarding the known fact that any rooted graph is bisimilar to some possibly infinite tree, the bisimulation genericity can be applied to the validation problem in the following manner. Under the assumption that schemas s_{IN} and s_{OUT} and the transformation f do not distinguish bisimilar instances, the validation problem is shown to be equivalent to determine the proposition: "*for any tree* T that satisfies the given input schema s_{IN}, the output $f(T)$ satisfies the output schema s_{OUT}". Thus we can reduce the problem from general graphs to trees, which is much easier. In fact, validity of MSO becomes decidable on infinite trees [21] unlike on graphs.

Before formalizing this approach, we need to check the assumption that not only UnCAL transformations but schemas are also bisimulation-generic. This is proved in the next lemma.

Lemma 2. *Let s be a schema written in $\mathcal{GS}$ and $g_1 = (V_1, E_1, r_1), g_2 = (V_2, E_2, r_2)$ be graphs such that $g_1 \equiv g_2$. Then, $g_1 \in [\![s]\!]$ implies $g_2 \in [\![s]\!]$.*

Proof. Let $S \subseteq V_1 \times V_2$ be the witness relation of the bisimilarity $g_1 \equiv g_2$ and $m_1 : V_1 \to 2^{tname(s) \cup \{\mathsf{Data}\}}$ be the type assignment that ensures $g_1 \in [\![s]\!]$. We can construct the type assignment $m_2 : V_2 \to 2^{tname(s) \cup \{\mathsf{Data}\}}$ on g_2 as follows.

$$m_2(v') = \bigcup \{m_1(v) \mid (v, v') \in S\}$$

Let us show that this assignment makes the graph g_2 satisfy the schema s. First, for the root node, we have $r_2 \lhd_{m_2} rtype(s)$, because we have $m_2(r_2) \supseteq m_1(r_1)$ due to $(r_1, r_2) \in S$, and $r_1 \lhd_{m_1} rtype(s)$ (recall that $v \lhd_m \rho$ is a shorthand for $m(v) \cap ns(\rho) \neq \emptyset$). Next, let us assume any node $v_2 \in V_2$ assigned a type $\tau \in m_2(v_2)$ and show $E_2^{\rightarrow}(v_2)$ satisfies $tdecl_s(\tau)$. By the construction

of m_2, there exists some v_1 such that $(v_1, v_2) \in S$ and $\tau \in m_1(v_1)$, and hence $E_1^{\rightarrow}(v_1)$ satisfies $tdecl_s(\tau)$. Let $tdecl_s(\tau) = \{l_1 : \rho_1, \ldots, l_n : \rho_n\}$ (the other cases can be proved similarly), (l', u') be any edge in $E_2^{\rightarrow}(v_2)$, and u be some node satisfying $(u, u') \in S$ whose existence is assured because S is a bisimulation. The label l' must be equal to some of l_i's; otherwise, there must be an edge $(l', u) \in E_1^{\rightarrow}(v_1)$, which contradicts the assumption that $E_1^{\rightarrow}(v_1)$ satisfies $tdecl_s(\tau)$. Furthermore, u' satisfies the condition $u' \lhd_{m_2} \rho_i$, because $u \lhd_{m_1} \rho_i$ and $m_2(u') \supseteq m_1(u)$. □

This bisimulation-genericity of schemas and transformations allows us to concentrate only on representatives among bisimilar graphs, instead of dealing with all kind of graphs. Let b be a function from graphs to graphs such that $g \equiv b(g)$. Intuitively, b is a function to obtain the representative among the set of graphs bisimilar to g. Then the following lemma holds.

Lemma 3. *Let b be a function from graphs to graphs such that $g \equiv b(g)$ for any g. Let φ and ψ be a bisimulation-generic (i.e., $\varphi(g) = \varphi(g')$ when $g \equiv g'$) properties on graphs, and f be a bisimulation-generic transformation. Then, the claim "$\varphi(g)$ implies $\psi(f(g))$ for any graph g" holds if and only if "$\varphi(g)$ implies $\psi(f(g))$ for any graph g in range of b".*

Proof. The 'only if' direction is trivial. For the 'if' direction, $\varphi(g)$ implies $\varphi(b(g))$ by the bisimulation-genericity of φ. Then, since $b(g)$ is in the range of b, we have $\psi(f(b(g)))$, which implies $\psi(f(g))$ by bisimulation-genericity of ψ and f. □

It is well-known that any rooted graph is bisimilar to an infinite tree called the *unfolding* of the graph. Let us formally state the property. Let $g = (V, E, r)$ be a graph. The unfolding procedure $unfold(g)$ is defined as (V', E', r') where

$$\begin{aligned} V' &= \{(v, p) \mid v \in V, p \text{ is a path from } r \text{ to } v\} \\ E'((v, p)) &= \{(l, (u, p.(l, u))) \mid (l, u) \in E(v)\} \\ r' &= (r, \epsilon). \end{aligned}$$

Here a *path* from r to v is a finite list $(l_1, u_1) \cdots (l_n, u_n)$ of edges such that $(l_1, u_1) \in E(r)$, $(l_{i+1}, u_{i+1}) \in E(u_i)$, and $u_n = v$ (if any). ϵ denotes the empty path and . denotes concatenation. Note that $unfold(g)$ always yields a tree, i.e., a graph with no loops and sharings, because each invocation of *unfold* creates a fresh node. The resulting tree is infinite when the original graph contains cycles. By taking the bisimulation relation S as $\{(v, (v, p)) \mid v \in V, (v, p) \in V'\}$ it is easy to see that g is bisimilar to $unfold(g)$. Now, applying Lemma 3 with $b = unfold$ proves the following main theorem of this section: validation problem of UnCAL on graphs is reduced to that on infinite trees.

Theorem 1 (Graphs to Infinite Trees)**.** *Let s_{IN} and s_{OUT} be schemas written in $\mathcal{GS}$, and f be a transformation written in UnCAL. Then, the claim "for any graph g, $g \in [\![s_{\text{IN}}]\!]$ implies $f(g) \in [\![s_{\text{OUT}}]\!]$" is equivalent to the claim "for any possibly infinite tree T, $T \in [\![s_{\text{IN}}]\!]$ implies $f(T) \in [\![s_{\text{OUT}}]\!]$".*

It is worth remarking that, theoretically, this theorem in addition to the MSO-definability results in Section 5 already establishes sound and complete validation.

4 Infinite Trees to Finite Trees

Infinite trees are much better domain compared to graphs in that they in fact already allows to give a decidable validation algorithm. Validity of quite a few logics (including MSO [21] that we will use later) cross over the borderline of decidability when we restrict the domain from graphs to infinite trees.

There is, however, a problem with infinite trees regarding practical efficiency. As far as we know, there is no realistic implementation on the validity of MSO on infinite trees. On the other hand, for MSO on *finite trees*[2], there exists a good practical implementation MONA [13], whose efficiency is verified in many applications. In order to implement a practically efficient validation of graph transformations, it is essential to reduce the problem further to the domain of finite trees. To this end, we show in this section that the validation problem over infinite trees can be reduced to that over finite trees.

The key idea for restricting the input domain to finite trees comes from the following observation: if an input infinite tree causes an error (i.e., generates an output graph not satisfying the output schema), it must be due to some edge(s) finitely reachable from the root node of the tree. In such a case, even if we cut off the infinite continuation below the erroneous edges and make the tree finite, it should still reveal the error.

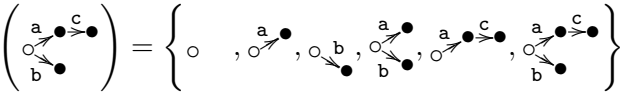
Let us formalize the notion of the "cutting off". For trees $T_1 = (V_1, E_1, r_1)$ and $T_2 = (V_2, E_2, r_2)$, we define the prefix-order relation $T_1 \preceq T_2$ to hold if and only if there is a one-to-one mapping e (stands for *e*mbedding) from V_1 to V_2 such that $e(r_1) = r_2$ and $(l, u_1) \in E_1(v_1)$ iff $(l, e(u_1)) \in E_2(e(v_1))$.

Definition 2. For a possibly infinite tree T, the set of its *finite-cut trees* (or *finite-cuts* for short) is $cut(T) = \{t \mid t \preceq T, t \text{ is a finite tree}\}$.

[2]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. The restriction on the domain of second-order variable essentially prohibits us to encode infinitely many labeled-edges. Hence, we call it MSO on finite trees. Similarly, we mention MSO on the infinite k-ary tree domain with no restriction (called SkS) as MSO on infinite trees.

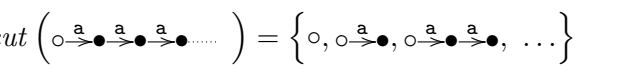
For instance, consider the following example of finite-cuts of a four-node tree.

$$cut\left(\right) = \left\{ \quad , \quad , \quad , \quad , \quad \right\}$$



More interesting example is the finite-cuts of an infinite tree

$$cut\left(\right) = \left\{\circ, \quad , \quad , \ \ldots\right\}$$



that produces infinitely many finite trees.

Definition 3. 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 $t \preceq t_c$.

Intuition is, $t \preceq t'$ 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 central player of this section concerning the notion of cuts is the nice property called *compactness* of core UnCAL. In the appendix of [7], it is proved that all positive UnCAL transformations are compact, i.e., instead of transforming an infinite tree T by an UnCAL transformation f, we only need to transform every finite tree of the set $cut(T)$ in order to obtain enough information to construct $f(T)$.

Lemma 4 ([7], Proposition 8)**.** *Let T be a possibly infinite tree and f be a transformation written in the core UnCAL. Then, $\{unfold(f(t)) \mid t \in cut(T)\}$ covers $unfold(f(T))$.*

The lemma is proved in [7] for a use as an easy-to-use proof method for deriving several optimization laws. Here, we are to show another application of the lemma, to the validation problem of transformations.

Similar property can be proved for our schema language $\mathcal{GS}$, too. Every finite-cut of a tree T satisfy the schema which is satisfied by the original tree T, and more importantly, if all the finite-cuts of T satisfy a schema s, then it means the original tree T also satisfies the schema s. In other words, $cut(T)$ contains enough information to test the schema satisfaction of T.

Lemma 5. *The following properties hold for a possibly infinite tree T and a schema s written in $\mathcal{GS}$:*

1. *$T \in [\![s]\!]$ implies $t \in [\![s]\!]$ for any finite tree $t \in cut(T)$.*
2. *If there exists a set $C \subseteq [\![s]\!]$ that covers T, we have $T \in [\![s]\!]$.*

Proof. Let $T = (V, E, r)$. For the first property, let us assume m to be the witness mapping of $T \in [\![s]\!]$. Let $t = (V_t, E_t, r_t)$ be a finite cut of T and let $e : V_t \to V$ to be the witness of $t \preceq T$. Then by taking the assignment

m_t as $m_t(v_t) = m(e(v_t))$, we can show the schema satisfaction $t \in [\![s]\!]$. For the root node, $m_t(r_t) = m(e(r_t)) = m(r)$ and hence whose intersection with $ns(rtype(s))$ is nonempty. For any node $v_t \in V_t$ with $\tau \in m_t(v_t)$, there cannot be any edge $(l, u_t) \in E_t^{\rightarrow}(v_t)$ violating $tdecl(\tau)$, otherwise the outgoing edge $(l, e(u_t))$ in $E^{\rightarrow}(e(v_t))$ violates the declaration $tdecl(\tau)$.

For the second property, if T is finite then C must contain a tree isomorphic to T itself and thus it is immediate. Consider the case T (and hence C) is infinite. We can assume C to contain a countable chain $t_1 \preceq t_2 \preceq \ldots$ of finite trees covering T. Without loss of generality, we can assume each t_i to have the form $(V_i, E|_{V_i}, r)$ with $V_i \subseteq V$ and $E|_{V_i}$ is the restriction of E to V_i. Let M_i to be the set of all type assignments $m_i : V \to 2^{tname(s) \cup \{\mathsf{Data}\}}$ whose restrictions $m_i|_{V_i}$ to V_i are witnesses for $t_i \in [\![s]\!]$, and $M_i(v)$ for $v \in V$ to be the set $\bigcup_{m_i \in M_i} m_i(v)$. Note that from the proof of the first property of the present lemma, $M_i(v) \supseteq M_j(v)$ for any v when $i \leq j$, i.e., a type assignment for a larger cut works also for smaller cuts. Now, we construct the type assignment $m : V \to 2^{tname(s) \cup \{\mathsf{Data}\}}$ as follows: $m(v) = \{\tau \mid \tau \text{ occurs infinitely often in the sequence } M_1(v), M_2(v), \ldots\}$. Let us verify that the assignment ensures $T \in [\![s]\!]$. First, let us check $r \lhd_m rtype(s)$, i.e., $m(r) \cap ns(rtype(s)) \neq \emptyset$. Suppose not, then there exists some i such that $M_i(v) \cap ns(rtype(s)) = \emptyset$, which contradicts $t_i \in [\![s]\!]$. Next, let us assume $\tau \in m(v)$ and check whether v satisfies $tdecl(\tau)$. Consider the case when $tdecl(\tau) = \{l_1 : \rho_1, \ldots, l_n : \rho_n\}$ (other cases are similar). For any edge $(l, u) \in E^{\rightarrow}(v)$, the label l is either one of l_i's; otherwise, for a cut t_k containing v and u, none of its assignment m_k can have $\tau \in m_k(u)$, which contradicts the assumption that τ occurs infinitely often in the sequence. Thus, w.l.o.g. we assume $l = l_1$. In this case, $m(u) \cap ns(\rho_1)$ cannot be empty. Suppose it is empty, then none of $ns(\rho_1)$ occurs infinitely many in $M_1(u), M_2(u), \ldots$, which implies the existence of sufficiently large k such that $M_k(u) \cap ns(\rho_1) = \emptyset$. But since $M_k(v)$ contains τ, this contradicts the typing of t_k. □

Similarly to the previous section, by exploiting compactness of both transformations and of schemas, we can show that the validation problem of UnCAL on possibly infinite trees is reducible to that on finite trees.

Theorem 2 (Infinite Trees to Finite Trees)**.** *Let s_{IN} and s_{OUT} be schemas written in $\mathcal{GS}$, and f be a transformation written in UnCAL. Then, the claim "for any possibly infinite tree T, $T \in [\![s_{\text{IN}}]\!]$ implies $f(T) \in [\![s_{\text{OUT}}]\!]$" is equivalent to the claim "for any finite tree t, $t \in [\![s_{\text{IN}}]\!]$ implies $f(t) \in [\![s_{\text{OUT}}]\!]$".*

Proof. The former claim immediately implies the latter, because finite trees are the special cases of trees. Assume the latter claim, and T to be a tree satisfying s_{IN}. By Lemma 5 (1), all trees of $cut(T)$ satisfies the schema s_{IN}. Hence, by the assumed claim, every tree in $C = \{f(t) \mid t \in cut(T)\}$ satisfies s_{OUT}. By Lemma 2, $C' = \{unfold(f(t)) \mid t \in cut(T)\}$ also satisfies s_{OUT}. By

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

$$
\begin{array}{lcll}
\varphi & ::= & \mathsf{true} \mid \mathsf{false} & \\
& \mid & \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi & \text{standard logical connectives} \\
& \mid & t_f = t_f \mid t_s = t_s \mid t_f \in t_s \mid t_s \subseteq t_s & \\
& \mid & \exists^1 v_f.\varphi \mid \forall^1 v_f.\varphi \mid \exists^2 v_s.\varphi \mid \forall^2 v_s.\varphi & 1^{\text{st}} \text{ and } 2^{\text{nd}} \text{ order quantifiers} \\
& \mid & \mathbf{vert}(t_f) \mid \mathbf{edge}_l(t_f, t_f, t_f) & \text{graph primitives} \\
& & & (l \in \mathit{Label}_\varepsilon \cup \{\texttt{"data"}\})
\end{array}
$$

Figure 3: Syntax of Monadic Second-Order Logic

Lemma 4, C' covers $\mathit{unfold}(f(T))$. Thus, by Lemma 5 (2), we have that $\mathit{unfold}(f(T))$ satisfies the schema s_{OUT}. By Lemma 2, this implies that $f(T)$ satisfies s_{OUT}, which derives the former claim. □

Whether a tree T satisfies a schema s is equivalent to whether *all* the cut trees in $\mathit{cut}(T)$ satisfy s. Be aware that, even for a tree T *not* satisfying the schema s, there may be some tree $t \in \mathit{cut}(T)$ that does satisfy s (actually, there indeed exists such a tree: single-node tree $(\{\bullet\}, \bullet \mapsto \{\}, \bullet)$ satisfies any schema and is always a cut of other trees). This way of correspondence is adequate for our purpose, because we are considering the reduction from a universal property "transformation conforms to the schemas for all infinite tree T" to another universal property.

5 Validation through Monadic Second-Order Logic

So far, we have reduced the validation problem on graphs that determine whether or not "$f(g)$ satisfies s_{OUT} for any graph g satisfying s_{IN}" to the problem on finite trees "$f(t)$ satisfies s_{OUT} for any finite tree t satisfying s_{IN}". In this section, we show the proposition can directly be expressed as a MSO formula, whose validity is known to be decidable on finite trees [23].

Before going into the detail, let us add some explanation on the choice of the logic. The most natural choice of logic for representing UnCAL transformations is first-order logic with transitive closures (FO+TC), which is shown in [7] to capture the full expressive power of UnCAL. The problem of FO+TC is that the validity of its formula is undecidable [24] even on finite trees, let alone graphs. Hence, naïvely reducing the problem to the validity of FO+TC formula can only derive either unsound, incomplete, or possibly non-terminating algorithm for the validation. Rather, our approach is to start from a decidable logic (namely, MSO) that can capture some clearly

defined fragment of UnCAL (i.e., the core UnCAL), and provide sound, complete, and terminating validation algorithm for the fragment, which we hope to be a solid framework towards the complete validation of full UnCAL. Section 7 discusses possible directions for enlarging the class of schemas and transformations that can be captured by the decidable logic.

Another logic worth remarking is MSO on infinite trees. Although the logic is also known to be decidable, as explained in Section 4, we prefer MSO on finite trees, emphasizing practically efficient implementation.

5.1 Review of MSO

The syntax of the formula of MSO over edge-labeled graph structure is in Figure 3. The variant of MSO we have adopted is basically that used to describe $(2,2)$-definable MSO transduction of Courcelle [9], with customizations to adjust for our purpose, namely adding the root constant and making edge predicates $\mathbf{edge}_l$ to be labeled. For a graph $g = (V, E, r)$ and an environment Γ that maps first-order variables to $V \cup E$ and second-order variables to subsets of $V \cup E$, the judgment relation $g, \Gamma \vDash \varphi$ is defined standardly. We present the definition on the two graph-specific primitives:

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

where Γ is extended as $\Gamma(\mathsf{root}) = r$, $\Gamma(t_1 \cup t_2) = \Gamma(t_1) \cup \Gamma(t_2)$, $\Gamma(t_1 \cap t_2) = \Gamma(t_1) \cap \Gamma(t_2)$, and $\Gamma(\emptyset) = \emptyset$, and the judgment relation for other connectives are defined standardly. We write $g \vDash \varphi$ when $g, \Gamma \vDash \varphi$ holds for the empty environment Γ.

One thing we have to note here is that we have single predicate $\mathbf{edge}_{\texttt{"data"}}$ for edges with data-value labels, in contrast to having distinct $\mathbf{edge}_l$ predicate for each label $l \in \mathit{Label}_\varepsilon$. In other words, we are assuming that all data-value edges in graphs and transformations to be labeled by the same unique label `"data"`. This is justified without loss of generality for the following two reasons. First, for schemas, changing the label for data edges never affects schema satisfaction, because our schema has no way to distinguish each different data label. Second, for transformation, since we are considering the nest-free fragment of UnCAL transformations, we cannot compare two label variables and hence there are no ways to distinguish different data labels either.

5.2 Representing Schemas in MSO

The definition of schema satisfaction $g \in [\![s]\!]$ in Section 2.2 almost literally translates to an MSO formula.

Lemma 6. *For any schema s, there exists an MSO formula φ_s such that for any graph g the schema satisfaction $g \in [\![s]\!]$ becomes equivalent to $g \models \varphi_s$.*

Proof. Let $\{\tau_1, \ldots, \tau_n\} = \mathit{tname}(s)$. The concrete construction of φ_s is as follows

$$
\begin{aligned}
&\exists^2 T_{\tau_1} \ldots . \exists^2 T_{\tau_n} . \exists^2 T_{\mathsf{Data}} . \ (\\
&\qquad \mathsf{root} \in \mathit{union}(\mathit{rtype}(s)) \wedge \\
&\qquad \forall^1 v . \ (\mathbf{vert}(v) \rightarrow (\\
&\qquad\qquad (v \in T_{\tau_1} \rightarrow \psi_{\mathit{tdecl}_s(\tau_1)}(v)) \wedge \\
&\qquad\qquad (v \in T_{\tau_2} \rightarrow \psi_{\mathit{tdecl}_s(\tau_2)}(v)) \wedge \\
&\qquad\qquad\qquad \vdots \\
&\qquad\qquad (v \in T_{\tau_n} \rightarrow \psi_{\mathit{tdecl}_s(\tau_n)}(v)) \wedge \\
&\qquad\qquad (v \in T_{\tau_{\mathsf{Data}}} \rightarrow \psi_{\mathit{tdecl}_s(\mathsf{Data})}(v)) \\
&\qquad))) \\
\end{aligned}
$$

where $\mathit{union}(\tau_1 \mid \cdots \mid \tau_k) = T_{\tau_1} \cup \ldots \cup T_{\tau_k}$. Here, the list of second-order variables T_τ corresponds to the type assignment m in the definition of schema satisfaction. Each second-order variable T_τ is meant to denote the set $\{v \mid \tau \in m(v)\}$ of nodes assigned the type τ. Hence, $v \in \mathit{union}(\tau_1 \mid \cdots \mid \tau_k)$ is equivalent to $v \in T_{\tau_1} \vee \cdots \vee v \in T_{\tau_k}$ and therefore it is intended to mean $\tau_1 \in m(v) \vee \cdots \vee \tau_k \in m(v)$, or equivalently $v \lhd_m (\tau_1 \mid \cdots \mid \tau_k)$.

The formula $\psi_{\mathit{tdecl}_s(\tau)}(v)$ means that v satisfies $\mathit{tdecl}_s(\tau)$ and defined as follows. When $\mathit{tdecl}_s(\tau) = \{l_1 : \rho_1, \ldots, l_m : \rho_m\}$, $\psi_{\mathit{tdecl}_s}(v)$ becomes

$$
\begin{aligned}
&\exists^2 O . \ (\mathit{e\_out}(v, O) \wedge \forall e . \ ((e \in O \wedge \neg \mathbf{vert}(e)) \rightarrow \exists^1 x . \exists^1 y . \ (\\
&\qquad (\mathbf{edge}_{l_1}(x, e, y) \wedge y \in \mathit{union}(\rho_1))) \vee \\
&\qquad (\mathbf{edge}_{l_2}(x, e, y) \wedge y \in \mathit{union}(\rho_2))) \vee \\
&\qquad\qquad \vdots \\
&\qquad (\mathbf{edge}_{l_m}(x, e, y) \wedge y \in \mathit{union}(\rho_m))) \\
&\quad))) \\
\end{aligned}
$$

where $\mathit{e\_out}(v, O)$ is a predicate for computing $E^{\rightarrow}(v)$. It is intended to become true only when O denotes the set $E^{\rightarrow}(v)$ of outgoing edges (plus several auxiliary nodes, which are filtered out by the subsequent $\neg\mathbf{vert}(e)$). It is defined by using a standard technique to represent transitive-closure in MSO as the least fixpoint

$$\mathit{e\_out}(v, O) \equiv \mathit{e\_out}'(v, O) \wedge \forall^2 R . \ (\mathit{e\_out}'(v, R) \rightarrow O \subseteq R)$$

$$
\begin{aligned}
e\_out'(v, R) \equiv (v \in R) \wedge \\
&\forall^1 x.\forall^1 e.\forall^1 y.((x \in R \wedge \mathbf{edge}_{\varepsilon}(x, e, y)) \rightarrow y \in R) \wedge \\
&\forall^1 x.\forall^1 e.\forall^1 y.((x \in R \wedge \mathbf{edge}_{l'_1}(x, e, y)) \rightarrow e \in R) \wedge \\
&\quad \vdots \\
&\forall^1 x.\forall^1 e.\forall^1 y.((x \in R \wedge \mathbf{edge}_{l'_p}(x, e, y)) \rightarrow e \in R)\ )
\end{aligned}
$$

with $\{l'_1, \ldots, l'_p\} = \mathit{Label} \cup \{\texttt{"data"}\}$. The definition of e_out says that O is the least set satisfying $e\_out'(v, O)$ and the auxiliary relation $e\_out'(v, R)$ says that R is a fixpoint of the traversal of the graph through ε-edges. Note that, for simplicity of the formulas, R and O contain both nodes (that are reachable from v via ε-edges) and edges (with non-ε labels, outgoing from v or the other nodes reachable from v via ε-edges).

When $\mathit{tdecl}_s(\tau) = \{l_1 : \rho_1, \ldots, l_m : \rho_m, *\}$ having the trail star, $\psi_{\mathit{tdecl}_s}(v)$ becomes

$$
\begin{aligned}
&\exists^2 O.\ (e\_out(v, O) \wedge \forall e.\ ((e \in O \wedge \neg \mathbf{vert}(e)) \rightarrow \exists^1 x.\exists^1 y.\ (\\
&\quad (\mathbf{edge}_{l_1}(x, e, y) \wedge y \in \mathit{union}(\rho_1))) \vee \\
&\quad (\mathbf{edge}_{l_2}(x, e, y) \wedge y \in \mathit{union}(\rho_2))) \vee \\
&\qquad \vdots \\
&\quad (\mathbf{edge}_{l_m}(x, e, y) \wedge y \in \mathit{union}(\rho_m))) \vee \\
&\quad \mathbf{edge}_{l'_1}(x, e, y) \vee \cdots \vee \mathbf{edge}_{l'_p}(x, e, y) \\
&\ )))
\end{aligned}
$$

with $\{l'_1, \ldots, l'_p\} = \mathit{Label} \cup \{\texttt{"data"}\} \setminus \{l_1, \ldots, l_m\}$. When $\tau = \mathsf{Data}$, the formula $\psi_{\mathit{tdecl}_s}(v)$ becomes

$$
\begin{aligned}
&\exists^2 O.\ (e\_out(v, O) \wedge \forall e.\ ((e \in O \wedge \neg \mathbf{vert}(e)) \rightarrow \exists^1 x.\exists^1 y.\ (\\
&\quad \mathbf{edge}_{\texttt{"data"}}(x, e, y) \\
&\ ))),
\end{aligned}
$$

meaning that all the outgoing edges are labeled `"data"`. □

5.3 Representing Core UnCAL in MSO

Next, we express the core UnCAL transformation by using MSO logic. We adopt the formalism for describing graph transformations in MSO introduced by Courcelle [9].

Definition 4. A graph-to-graph transformation f is said to be a *k-copying MSO-definable transduction* if there exists a constant k and a set of formula $\mathbf{vert}_0(x), \ldots, \mathbf{vert}_{k-1}(x)$, $\mathbf{edge}_{l,i,j,m}(x, y, z)$ for $l \in \mathit{Label}_{\varepsilon} \cup \{\texttt{"data"}\}$, $i, j, m \in \{0, \ldots, k-1\}$ satisfying the following conditions for any $g = (V, E, r)$:

- For any pair of $w \in V \cup E$ and $j \in \{0, \ldots, k-1\}$, we have either $g \vDash \mathbf{vert}_j(w)$ or $g \vDash \mathbf{edge}_{l,i,j,m}(v, w, u)$ for at most one combination of i, m, v, u.
- $g \vDash \mathbf{edge}_{l,i,j,m}(v, w, u)$ implies $g \vDash \mathbf{vert}_i(v)$, $g \nvDash \mathbf{vert}_j(w)$, and $g \vDash \mathbf{vert}_m(u)$.
- The output $f(g)$ of the transformation is isomorphic to (V', E', r') where $V' = \{(v, i) \mid v \in V \cup E, g \vDash \mathbf{vert}_i(v)\}$, $E'((v, i)) = \{(l, (u, m)) \mid w \in V \cup E, g \vDash \mathbf{edge}_{l,i,j,m}(v, w, u)\}$, and $r' = (r, 1)$.

Intuitively, k-copying MSO-definable transduction creates k copies of input nodes and edges, and by reorganizing them to form the output graph structure according to the supplied formulas $\mathbf{vert}_i(x)$ and $\mathbf{edge}_{l,i,j,m}(x, y, z)$. The formula $\mathbf{vert}_i(x)$ indicates that the i-th copy of x (which is either a node or an edge in the input graph) becomes a node of the output graph, and $\mathbf{edge}_{l,i,j,m}(x, y, z)$ indicates that the j-th copy of y becomes an edge from the i-th copy node of x to the m-th copy node of z, labeled l.

MSO-definable transductions enjoy several nice properties. In particular, the following two properties are important.

Lemma 7 ([9], Proposition 3.2)**.** *(1) The inverse image of an MSO-definable set of graphs under an MSO-definable transduction is MSO-definable. That is, if f is an MSO-definable transduction and φ is an MSO formula on graphs, then there exists an MSO formula $f^{-1}(\varphi)$ such that $g \vDash f^{-1}(\varphi)$ if and only if $f(g) \vDash \varphi$. (2) The composition of two MSO-definable transductions is MSO-definable.*

The first property enables to convert MSO formulas on output graphs into that on input graphs. More specifically, instead of saying "the output graph $f(g)$ satisfies the schema s_{OUT}", i.e., "$f(g) \vDash \varphi_{s_{\text{OUT}}}$", we can convert it to the formula "$g \vDash f^{-1}(\varphi_{s_{\text{OUT}}})$" on input graphs. Using this conversion, the validation problem "for any graph g satisfying s_{IN}, the output $f(g)$ satisfies s_{OUT}" can be restated as the validity of the formula "$g \vDash \varphi_{s_{\text{IN}}} \to f^{-1}(\varphi_{s_{\text{OUT}}})$" on input graphs.

Transformations in core UnCAL turn out to be realizable as MSO-definable transductions. The construction is basically to follow carefully the semantics given in Section 2.3.

Lemma 8. *Any transformation f written in the core UnCAL is an MSO-definable transduction.*

To illustrate the idea, consider the following simple example:

$$\mathbf{rec}(\lambda(\$l, \$g).\ \&_1 := \{\mathtt{a} : \&_1, \mathtt{b} : \$g\})(\$db).$$

According to the semantics, on every edge e connecting nodes v and u in the input graph, the body expression is evaluated and generates a new fragment of graph as follows

and these graphs are aggregated to form the whole output. Here, recall that each marker $\&_i$ is represented by a newly created node ${}^i u$, and the variable $\$g$ is referring to the destination node u of the current edge. The important property of core UnCAL is that it prohibits access to outer variables, which implies that any variable expression inside a body of **rec** must point to the destination node u of the currently processed edge. Actually, this fact is crucial to make core UnCAL to be MSO-definable.

In order to represent the transformation as a MSO-definable transduction, it is natural to construct each ${}^i u$ node as the i-th copy of the input node u. To represent input subgraphs embedded in the output graph like u in the example above, we use 0-th copy of the input nodes and edges. Other components (i.e., the nodes and edges created during evaluation of the body expressions) of the output graph is constructed as the copies of the current edge e.

Thus, the example of the output shown above is represented as follows, using the notion of copying

$$(v,1) \xrightarrow{(e,1)} (e,2) \begin{cases} \xrightarrow{(e,3)} (u,1) \\ \xrightarrow{\mathtt{b}\ (e,4)} (u,0). \end{cases}$$

where (x,i) denotes the i-th copy of x. From this picture, we obtain the following set of formulas representing the example transformation as a 5-copying MSO-definable transduction

$$\begin{aligned}
\mathbf{vert}_0(x) &\equiv \mathbf{vert}(x) \\
\mathbf{edge}_{l,0,0,0}(x,e,y) &\equiv \mathbf{edge}_l(x,e,y) \\
&\qquad\qquad \text{for } l \in \mathit{Label}_\varepsilon \cup \{\texttt{"data"}\} \\
\mathbf{vert}_1(x) &\equiv \mathbf{vert}(x) \\
\mathbf{vert}_2(x) &\equiv \neg\mathbf{vert}(x) \\
\mathbf{vert}_3(x) &\equiv \mathbf{vert}_4(x) \equiv \mathsf{false} \\
\mathbf{edge}_{\varepsilon,1,1,2}(x,y,z) &\equiv y = z \wedge \exists^1 u.\, \mathbf{edge}_*(x,y,u) \\
\mathbf{edge}_{\mathtt{a},2,3,1}(x,y,z) &\equiv x = y \wedge \exists^1 v.\, \mathbf{edge}_*(v,y,z) \\
\mathbf{edge}_{\mathtt{b},2,4,0}(x,y,z) &\equiv x = y \wedge \exists^1 v.\, \mathbf{edge}_*(v,y,z) \\
\mathbf{edge}_{l,i,j,k}(x,y,z) &\equiv \mathsf{false} \qquad\qquad \text{otherwise}
\end{aligned}$$

where $\mathbf{edge}_*(t_1, t_2, t_3)$ is the shorthand for the formula $\mathbf{edge}_{l_1}(t_1, t_2, t_3) \vee \cdots \vee \mathbf{edge}_{l_p}(t_1, t_2, t_3)$ with $\{l_1, \ldots, l_p\} = \mathit{Label} \cup \{\texttt{"data"}\}$. (To be exact, we had to take into account the semantics of structural recursion that must preserve ε-edges in the input graph as-is in the output. For presentation purpose, we have omitted the part.) For example, the predicate $\mathbf{edge}_{\texttt{b},2,4,0}(x, y, z)$ tells that there is a `b`-edge $(y, 4)$ from $(x, 2)$ to $(z, 0)$ if and only if $x = y$ and y is an edge going to z, as is illustrated in the picture. Let us repeat again here that it is essential that we do not have nested variable reference in the core UnCAL. If it were (say, suppose $\$g$ was an outer-scope variable), the z of the destination node $(z, 0)$ in the output graph need not be the destination node of the current edge in the input graph, and hence there is no way to reach it from the current edge y like $\exists^1 v.\, \mathbf{edge}_*(v, y, z)$.

of Lemma 8. We first show the construction how to represent each structural-recursion defined by **rec** expressions as a MSO-definable transduction. The construction is by induction on the nesting height of **rec**. Let us consider a structural recursion $\mathbf{rec}(\lambda(\$l, \$g).\ \&_1 := e_1, \ldots, \&_n := e_n)$ of nesting height h, assuming that sub **rec** expressions occurring in $e_1, \ldots, e_n$ is by induction hypothesis MSO-definable. The base case of the induction is the case $h = 1$, meaning that there are no **rec** expressions in $e_1, \ldots, e_n$.

We first convert each body expression e_i to the following normal form that has **if** expressions only at the top-level of the expression (except **if** expressions inside the bodies of nested **rec** recursions)

$$
\begin{array}{l}
\textbf{if}\ \$l = l_1\ \textbf{then}\ \mathit{specialize}(e_i, l_1) \\
\textbf{else if}\ \$l = l_2\ \textbf{then}\ \mathit{specialize}(e_i, l_2) \\
\quad \vdots \\
\textbf{else if}\ \$l = l_p\ \textbf{then}\ \mathit{specialize}(e_i, l_p) \\
\textbf{else}\ \&_i
\end{array}
$$

where $\{l_1, \ldots, l_p\} = \mathit{Label} \cup \{\texttt{"data"}\}$ and $\mathit{specialize}(e_i, l)$ is the expression obtained from e_i by removing all **if**-subexpressions in a way that each **if**-subexpression $\textbf{if}\ \$l = l'\ \textbf{then}\ e_t\ \textbf{else}\ e_f$ is recursively replaced with e_t if $l' = l$ and with e_f otherwise, and by changing all the edges $\{\cdots \$l : e \cdots\}$ to $\{\cdots l : e \cdots\}$. Since it exhaustively checks all the labels, the final **else** branch is unreachable in the standard semantics. Note that, however, by placing $\&_i$ there, we get a unified treatment for the rather exceptional ε-edge rule of the structural recursion. That is, instead of dealing with input ε-edges specially, just using the normalized body expression above even for ε-edges would realize the same result. For this reason we prefer the normal form above and do not deal with ε-edges exceptionally.

Let $e_{i,l} = \mathit{specialize}(e_i, l)$ for $l \in \mathit{Label} \cup \{\texttt{"data"}\}$ and $e_{i,\varepsilon} = \&_i$, and consider the new structural recursion $f_l = \mathbf{rec}(\lambda(\$l, \$g).\ \&_1 := e_{1,l}, \ldots, \&_n := e_{n,l})$.

We compute the MSO-representation of each f_l separately. Suppose we have obtained the $k_{l'}$-copying representation of $f_{l'}$ as the set of predicates $\mathbf{vert}_i^{l''}$ and $\mathbf{edge}_{l,i,j,k}^{l'}$. We can combine them into a single $\max_{l'}(k_{l'})$-copying transduction realizing the original structural recursion by a simple case-analysis formula: $\mathbf{vert}_i(x) = \mathbf{vert}(x) \vee \bigvee_{l'}((\exists^1 v.\exists^1 u.\mathbf{edge}_l(v,x,u)) \wedge \mathbf{vert}_i^{l'}(x))$ and $\mathbf{edge}_{l,i,j,k}(x,y,z) = \bigvee'_{l'}(\exists^1 v.\exists^1 u.\mathbf{edge}_{l'}(v,y,u)) \wedge \mathbf{edge}_{l,i,j,k}^{l''}(x))$ for each l, i, j, k.

Representing each f_l as a MSO-definable transduction is done just as illustrated in the preceding example. The markers $\&_i$'s are represented as the i-th copies of nodes, variable $\$g$ is represented as the 0-th copy of the destination node of the currently processed edge, and node/edge-construction expressions are assigned unique numbers j by, e.g., a depth-first traversal on the body expression, and constructed as the j-th copy of the edge. Nested recursion $\mathbf{rec}(\cdots)(e_a)$ is treated as follows. By inductively processing the argument expression, we can assume e_a is represented by a k_a-copying representation. Since by induction hypothesis the recursion is some k'-copying transduction, we can represent its output by k_a times k' copies, assigning fresh copy numbers $j', \ldots, j' + k_a k' - 1$ (this is basically the same technique as the composition of MSO-definable transductions of Lemma 7). Its root is represented as the j'-th copy of the root node of the representation of e_a.

So far, we have shown that each **rec** structural recursion is a MSO-definable transduction. Showing the whole UnCAL transformation f to be MSO-definable can be done in quite the same manner. Note that f can contain the designated input variable $\$db$, **rec** expression, or node-construction expression, but no markers $\&_i$, nor **if** expressions (because no label variable is in the scope). The variable $\$db$ is represented as the 0-th copy of the input root node, and **rec** and node-construction expressions are dealt as same as in f_l, except that each node is constructed as the copy of the root node of the input graph, not as the copy of the "current edge", which does not exist here. □

Wrapping up all the results presented so far, we derive the following main theorem of the paper.

Theorem 3 (Sound and Complete Validation)**.** *Let s_{IN} and s_{OUT} be schemas written in $\mathcal{GS}$, and f be a transformation written in UnCAL. We can effectively determine the validation problem "for any graph g satisfying s_{IN}, the output graph $f(g)$ satisfies s_{OUT}" of graph transformations.*

Proof. By Theorems 1 and 2, the claim is equivalent to "for any finite tree t, $t \in [\![s_{\text{IN}}]\!]$ implies $f(t) \in [\![s_{\text{OUT}}]\!]$". By Lemmas 6, 7 and 8, it is equivalent to "$t \vDash \varphi_{s_{\text{IN}}} \to f^{-1}(\varphi_{s_{\text{OUT}}})$ holds for any finite tree t". Since it is a validity problem of an MSO formula on finite trees, it is decidable [23]. □

6 Related Work

Verification of model (graph) transformations is an important issue in software engineering. The approaches presented so far, however, are applied only to certain simple model transformations that can be easily mapped to Prolog or CSP [16, 2], or only for certain properties such as equivalence between input and output models [20]. In contrast, our verification covers a wide class of model transformations and guarantee that the model transformation will map schema-correct input model to a schema-correct output model.

Another group of related work on validation of transformations can be found in the area of XML processing, under the name *exact typechecking* [25, 19, 17, 12]. Our novelty compared to those work is that we have dealt with graphs and shown the reduction to finite trees. After the reduction and the conversion to MSO-definable transduction, our approach to construct the inverse image $f^{-1}(\varphi_{s_{\text{OUT}}})$ of the output-schema satisfaction formula follows the same way as those researches on XML typechecking.

The most directly related work is the simulation-based schema for UnCAL graph model introduced by Buneman et al. [5]. Sound, complete, and decidable validation algorithm of transformations with those schemas is also given in the same paper. Compared to their schema, our schema language $\mathcal{GS}$ has both enhancement and shortage. Their schema is more suitable for expressing *properties on data values*, because their schema can have unary predicates putting constraints on *Data* edges (like, "it must match some regular expression pattern"), which cannot be dealt with in our framework. On the other hand, our schema is more inclined to representing *structural properties* of graphs. For example, $\mathcal{GS}$ has the trailing star $\{\ldots *\}$ type declaration that allows existence of arbitrary edges in addition to the specified ones. Or, we have the union operator $\tau_1 \mid \tau_2$ on types. For instance, the SNS schema example in Section 2.2 contains a type $\mathsf{Data} \mid \mathsf{Name}$, meaning that the outgoing edge consists of data-value edges *or* a set of edges labeled `first`, `family`, and `middle`, *not both*. Such a "not both"-type condition cannot be expressed in Buneman et al.'s schema. Such feature is, however, crucial for writing structural constraints, regarding the situation that all standard XML schemas [4, 27, 8] has the notion of unions, or, the notion of inheritance in metamodel language like [1] (which essentially another schema language for graphs) being a variant of union type. Since Buneman et al.'s schema is heavily based on the simulation relation over graphs, it is not at all clear how to extend to these structural properties, while our approach generalizes to any types of schema, as long as it is MSO-definable, bisimulation-generic, and compact.

7 Conclusion and Future Work

We have shown the novel algorithm that verify a graph transformation written in the core UnCAL is correct with respect to the specified input/output schemas describing structural property of graphs. Our algorithm is sound, complete, and decidable, in the sense that all correct transformations are always reported as correct, and all erroneous transformations are always reported as so. The technical contribution of the paper is summarized as follows:

- we have recognized and demonstrated the usefulness of bisimulation-genericity and compactness of graph transformations in the context of validation, which adds another importance of structural-recursion-based graph transformation in addition to optimization [7] or bidirectionalization [14],
- we have proved those two properties also for the schema language $\mathcal{GS}$; together with the first contibution, these properties allow to reduce the validation problem on graphs to that on trees without losing soundness and completeness, and
- we have given a MSO based semantics of the core UnCAL, which enabled decidable validation.

The challenge for the future is to establish the validation algorithm for *full* UnCAL. As explained in Section 2.3, the major differences between the core UnCAL and the full UnCAL are twofold. One is that nested **rec** allowed to refer to outer variables, which breaks the MSO-definability. For instance, nested UnCAL transformation can produce an output graph polynomially larger than an input graph, while MSO-definable transduction has only linear-size increase by definition. We are considering to address the issue by introducing more powerful formalism for describing graph transformations, which still preserves the inverse MSO-definability. Note that, the essentially only property of MSO-definable transductions we really needed in Section 5 is that its inverse image of an MSO-formula is again an MSO-formula. The translation itself need not be MSO-definable! In the area of tree-transformation, such powerful yet MSO-definability-preserving formalism are widely used for the very same purpose (such as, macro tree transducers [11] or pebble tree transducers [19]). We believe that similar technique can be devised for graph transformations.

Another difference from the full UnCAL is the $\mathsf{isEmpty}(\$g)$ predicate, which allows to test the emptiness (= nonexistence of outgoing edges) of a node and in fact breaks the compactness. We think, this is mainly because the current definition of $cut(T)$ is too simple. Trees t in $cut(T)$ are obtained by simply eliminating subtrees of T, and therefore in t, there is not left

any sign whether each empty node was indeed empty in the original tree T or it became empty due to the cut operation. A possible direction is to introduce an extended notion of cuts with richer information, e.g., leaving some special annotation to the cut-nodes so that the transformation can distinguish different kinds of empty nodes.

The other important challenge is to support richer schema languages. We are mainly interested in supporting cardinality constraints on the number of edges. For example, we are planning to allow type declaration like $\mathsf{Person} = \{\mathtt{name}[1] : \mathsf{Data}, \mathtt{email}[1..*] : \mathsf{Data}\}$ meaning that there must be *exactly one* edge labeled `name`, and *at least one* edge labeled `email`. Such extension can almost subsume the standard schema language [1] used for model-driven software development. Two things must be considered here. The definition of bisimulation-genericity presented in the paper is based on *set-semantics* where the collection of outgoing edges $E(v)$ is defined to be a set of edges. In this setting, cardinality other than $[0..*]$ and $[1..*]$ are meaningless because duplicating edges are always unified. To sensefully introduce other cardinalities, we need to consider thoroughly the bag- or list-based semantics of UnCAL, which is slightly mentioned in the original paper [7] of UnCAL. More severe issue is that introducing cardinalities like $[1..*]$ (or whatever the one with non-zero lower bound) breaks the compactness of schemas. We need to find some way to address the issue.

References

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

[2] D. Bisztray and R. Heckel. Rule-level verification of business process transformations using CSP. *Electronic Communications of the EASST*, 6, 2007.

[3] D. Blostein, H. Fahmy, and A. Grbavec. Practical use of graph rewriting. Technical report, Queen's University, 1995.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (XML™). `http://www.w3.org/XML/`, 2000.

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

[6] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *ACM SIGMOD International Conference on Management of Data*, pages 505–516, 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: Very Large Data Bases*, 9(1):76–110, 2000.

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

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

[10] 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. In *Model Transformations in Practice*, 2005.

[11] J. Engelfriet and H. Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31:71–146, 1985.

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

[13] J. G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 89–110, 1995.

[14] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing structural recursion on graphs. Technical Report GRACE-TR09-03, GRACE Center, National Institute of Informatics, Aug. 2009.

[15] S. Hidaka, Z. Hu, H. Kato, and K. Nakano. Towards a compositional approach to model transformation for software development. In *ACM Symposium on Applied Computing*, pages 468–475, 2009.

[16] G. Karsai and A. Narayanan. Towards verification of model transformations via goal-directed certification. In *Model-Driven Development of Reliable Automotive Services*, pages 67–83. Springer-Verlag, 2008.

[17] S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In *International Conference on Database Theory*, pages 254–268, 2007.

[18] T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In *International Conference on Graph Transformation*, pages 286–301, 2002.

[19] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *Journal of Computer and System Sciences*, 66:66–97, 2003.

[20] A. Narayanan and G. Karsai. Towards verifying model transformations. *Electronic Notes in Theoretical Computer Science*, 211:191–200, 2008.

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

[22] N. Robertson and P. D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *Journal of Algorithms*, 7:309–322, 1986.

[23] 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*, 2:57–811, 1968.

[24] H.-J. Tiede and S. Kepser. Monadic second-order logic and transitive closure logics over trees. In *Workshop on Logic, Language, Information and Computation*, pages 189–199, 2006.

[25] A. Tozawa. Towards static type checking for XSLT. In *ACM Symposium on Document Engineering*, pages 18–27, 2001.

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

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