Sound and Complete Validation of Graph Transformations

Kazuhiro Inaba (稲葉 一浩)
NII

Feb 26, 2010
GRACE Seminar
Graphs

Semi-Structured Database

UML Diagram
Graph Transformations

Function from Graphs to Graphs

"retrieve all subgraphs pointed by an edge labeled <name>"

"rename all edges labeled <person> to <personal-data>", etc.
Graph Schema

Structural Constraints on Graphs

“all edges must be labeled either <foo> or <bar>”

“a node pointed by <person>-edge can have outgoing edges <name>, <father>, <mother> only”, etc.
What We Want to Do

Transformation $f$

Validation
(Static Typchekeking)

"Any graph satisfying $S_I$ always transformed by $f$ into a graph satisfying $S_O$!"

Schema $S_I$

Schema $S_O$
Our Approach: Use Logic

Covert to Logic Formula

Solver

Transformation $f$

Schema $S_I$

Schema $S_O$

“Input and Output are related by $f$”

“input satisfies $S_I$” “output satisfies $S_O$”

Valid! (True for any graph!)
Agenda

Languages
- Graph Model
- Transformation Language: Core UnCAL
- Graph Schema

Validation using Logic
- Conversion to Logic Formula
- Making it Decidable

Conclusion & Future Work
Graph Data Model of UnCAL [Buneman et al, 2000]

- Rooted
- Directed
- Edge-Labeled
  - (Unordered)

Diagram:
- Nodes: person, name, child, father
- Edges:
  - "John" → "name" → "father" → "person"
  - "person" → "name" → "child" → "John"
  - "person" → "name" → "child" → "Jack"
Graph Data Model (Cont’d)

ε-edges

ε

ε

ε

ε

ε

ε

=
Why $\varepsilon$-edges?

To simplify the transformation language e.g., “union” operation by $\varepsilon$-edge
Why $\varepsilon$-edges?
Formally …

\[ G = (V, E, r) \]

- \( V \) : Set of Nodes
- \( r \in V \) : Root Node
- \( E \) : Mapping from \( V \) to set of \((\text{Label} \times V)\)

\[ V = \{1, 2, 3\} \]
\[ r = 1 \]
\[ E(1) = \{(\text{person}, 2), (\text{person}, 3)\} \]
\[ E(2) = \{(\text{dad}, 3)\} \]
\[ E(3) = \{(\text{son}, 2)\} \]
We use a Core Fragment of UnCAL

\[
E ::= \{L:E, L:E, \ldots, L:E\} \\
| \text{if } L=L \text{ then } E \text{ else } E \\
| G \\
| \& \\
| \text{rec}(\lambda(L,G). E)(E)
\]

\[
L ::= (\text{label constant}) \\
| L
\]
Node Construction

{person: {name: {“John”:{}}},
person: {name: {“Jack”:{}}},
person: {name: {“Mary”:{}}}}
Structural Recursion
= “Visit every edge and do something”

rec(λ($L,$G). E)(E)

E ::= {L:E, L:E, ..., L:E} 
| if $L$=a then {a: $G$} else E 
| $G$ 
| & 
| rec(λ($L,$G). E)(E) 

L ::= (label constant) | $L$

E ::= {L:E, L:E, ..., L:E} 
| if $L$=L then E else E 
| $G$ 
| & 
| rec(λ($L,$G). E)(E) 

L ::= (label constant) | $L$
\[ \text{rec} \left( \lambda (\$L, \$G). \right. \]
\[ \quad \text{if } \$L = a \text{ then } \{ a: \ $G \} \text{ else} \]
\[ \quad \text{if } \$L = b \text{ then } \{ bb: & \} \text{ else} \]
\[ \quad \{ \varepsilon: & \} \left. \right) (\$db) \]
rec(\(\lambda(L,G).\)
  \(\text{if } L=a \text{ then } \{a: G\} \text{ else}\)
  \(\text{if } L=b \text{ then } \{bb: \&\} \text{ else}\)
  \(\{\varepsilon: \&\}\) )($db$)
rec(λ($L$, $G$).
    if $L=a$ then \{a: $G$\} else
    if $L=b$ then \{bb: &\} else
    \{ε: &\}
)(\$db\)
Another Example

Subgraph Querying

\[
\text{rec}(\lambda(\$L1,\$G1). \begin{array}{l}
\text{if } \$L1=\text{person then} \\
\text{  \quad \quad \ \text{rec}(\lambda(\$L2,\$G2).} \\
\text{    \quad \quad \quad \text{if } \$L2=\text{name then} \{\text{name:}\$G2\} \text{ else } \{\}} \\
\text{    \quad \quad \}(\$G1) \\
\text{else } \{\} \\
\end{array}\}($G1) \\
\text{else } \{\} \\
\}(\$db)
\]
\[ \text{rec}(\lambda($L1,$G1). \text{if } $L1=\text{person} \text{ then } \text{rec}(\lambda($L2,$G2). \text{if } $L2=\text{name} \text{ then } \{\text{name}:$$G2\}\text{ else } \{\}\text{ else } \{\}\}($G1) \text{ else } \{\}\})(\$db) \]
$$\text{rec}(\lambda (L_1,G_1). \text{if } L_1 = \text{person} \text{ then } \text{rec}(\lambda (L_2,G_2). \text{if } L_2 = \text{name} \text{ then } \{\text{name}:G_2\} \text{ else } \{\}\}(G_1) \text{ else } \{\}\}(db)$$
Note

SQL $\leftrightarrow$ Relational Algebra

UnQL $\leftrightarrow$ UnCAL : Graph Algebra

UnCAL is not meant to be a user-friendly language. Rather, it is primitive internal algebra.

SELECT {name:$g}
WHERE {person.name: $g} IN $db

$\text{rec}(\lambda($L1,$G1). \text{if } L1=\text{person} \text{ then } \text{rec}(\lambda($L2,$G2). \text{if } L2=\text{name} \text{ then } \{\text{name}:G2\} \text{ else } {} )\text{ else } {} )($G1) \text{ else } {} )($db)
Not Essential (Omitted for simplifying presentation)

Cycle Construction in Transformation

Mutual Recursion

\[
\&_i \mid \text{rec}(\lambda(\$L, \$G). \&_1 := \text{E}, \ldots, \&_n := \text{E})(\text{E})
\]

Truly Nested Recursion

\[
\text{rec}(\lambda(\$L1, \$G1). \\
\quad \text{rec}(\lambda(\$L2, \$G2). \ldots \ $G1 \ldots)(\ldots) \\
)(\ldots)
\]

Emptiness Checking

\[
\text{if isEmpty($g$) then } \ldots
\]
root is PersonSet

class PersonSet {
  person[0-\*] : Person
}

class Person {
  name  [1][0-\*] : String
  father[0-\*] : Person
  child  [0-\*] : Person
}
Be aware on ε-edges

root is ROOT
class ROOT {
    foo[2] : String
}
is satisfied by

ε

“hello”

“world”
A graph $G=\langle V, E, r \rangle$ satisfies a schema $s$ (written as $G \in [[s]]$) when

$$\exists m : V \rightarrow 2^{\{T_1, \ldots, T_n\}}$$

$$T_1 \in m(r)$$

$$\forall v \in V$$

$$\forall T_i \in m(v)$$

$$\forall (e, u) \in E \in \varepsilon(v)$$

$$\exists j . e = e_{ij}$$

$$\& S_{ij} \in m(u)$$

root is $T_1$

```java
class T1 {
    e11 [0-*] : S11
    e12 [0-*] : S12
    ...
    e1k [0-*] : S1k
}
class T2 { ... }
```

```java
... class Tn { ... }
```
Agenda

Languages
- Graph Model
- Transformation Language: Core UnCAL
- Graph Schema

Validation using Logic
- Conversion to Logic Formula
- Making it Decidable

Summary & Future
[Review] Our Approach: Use Logic

Schema $S_I$ \rightarrow\text{Transformation} f \rightarrow Schema $S_O$

Covert to Logic Formula

$\Phi = \text{"} G \in [[[S_I]]] \text{"} \Rightarrow \text{"} f(G) \in [[[S_O]]] \text{"}$

if $G$ satisfies $S_I$ then $f(G)$ satisfies $S_O$

Check whether $G \models \varphi \text{ for all } G$
Question

How to express “G ∈ [[S_i]]” in logic formula?

How to express “f(G) ∈ [[S_o]]” in logic formula?

How can we check “G |= φ for any G”?
### Depends on Choice of Logic

<table>
<thead>
<tr>
<th>Logic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FO</td>
<td>(First-Order Logic)</td>
</tr>
<tr>
<td>FO+TC(^1)</td>
<td>(FO + Transitive Closure)</td>
</tr>
<tr>
<td>MSO</td>
<td>(Monadic 2(^{\text{nd}})-Order Logic)</td>
</tr>
<tr>
<td>FO+TC(^k)</td>
<td>(FO + TC on k-tuples)</td>
</tr>
</tbody>
</table>

We need to choose
- **sufficiently expressive** logic that can express UnCAL transformations and schemas
- **not so strong** logic whose validness is decidable
Choice of Logic

- FO (First-Order Logic)
  - too weak
- FO\(+\text{TC}^1\) (FO + Transitive Closure)
- MSO (Monadic 2\textsuperscript{nd}-Order Logic)
- FO\(+\text{TC}^k\) (FO + TC on k-tuples)
  - too strong (unavoidably undecidable)
Choice of Logic

- **FO** (First-Order Logic)
  - too weak

- **FO+TC^1** (FO + Transitive Closure)
  - may be strong enough…?

- **MSO** (Monadic 2^{nd}-Order Logic)
  - strong enough for Core UnCAL
  - weak enough to be decidable

- **FO+TC^k** (FO + TC on k-tuples)
  - too strong (unavoidably undecidable)
Monadic 2\textsuperscript{nd}-Order Logic (MSO)

MSO is a

Usual 1\textsuperscript{st} order predicate logic

Boolean ops and quantifiers: \( \neg, \land, \lor, \forall, \exists \)

…extended with “set-quantifications”: \( \forall \text{ set} \exists \text{ set} \)

E.g.,

\[
\text{connected}(x,y) := \\
\exists \text{ set } P. (x \in P \land y \in P \land \forall u,v.(u \in P \land \exists e.\text{edge}(u,e,v) \Rightarrow v \in P)) & \ldots
\]
MSO (on Graphs) : Syntax

1st order variables range over nodes and edges

\[ \phi ::= \text{true} | \text{false} | \neg \phi | \phi \land \phi | \phi \lor \phi \\
| \forall x.\phi | \exists x.\phi \\
| \forall \text{set } X.\phi | \exists \text{set } X.\phi \\
| x=x | x \in X | X \subseteq X | X=X \\
| \text{isNode}(x) \\
| \text{edge}(x,e,z) \\
| \text{label}_xxx(e) | \text{label}_\text{string}(e) \]

For a graph G, 
\[ G \models \phi \]
(G makes \( \phi \) true)
is defined as usual.
From Schema to MSO

∃ set PersonSet, Person, String.
root ∈ PersonSet
∧ ∀ v ∈ PersonSet. ...
∧ ∀ v ∈ Person.
  ∃ set E, En, Ec. outgoing(v, E) ∧ E = En ∪ Ec
  ∧ |En| ≥ 0 ∧ ∀ e ∈ En. (label_name(e) ∧ ...)
  ∧ |Ec| ≥ 0 ∧ ∀ e ∈ Ec. (label_child(e) ∧
  ∃ u. (edge(_, e, u) ∧ u ∈ Person))
outgoing(v, E)

“E is the set of non-ε edges reachable from v using only ε-edges”

Essentially the transitive closure operation

outgoing(v, E) :=
\[ \forall e. ( e \in E \iff \exists u. \text{trancl}(v,u) \land \text{edge}(u,e,\_0) \land \neg \text{label}_\varepsilon(e) ) \]

\text{trancl}(v, u) :=
\[ \forall \text{set } P. \]
\[ v \in P \land ( \forall w \in P, z, e. \]
\[ \text{edge}(w,e,z) \land \text{label}_\varepsilon(e) \Rightarrow z \in P ) \]
\[ \Rightarrow u \in P \]
UnCAL to MSO

\[ \text{out\_edge}(v,e,u) := \text{edge}(v,e,u) \]
\[ \text{out\_label\_d}(e) := \text{label\_a}(e) \lor \text{label\_d}(e) \]
\[ \text{out\_label\_a}(e) := \text{false} \]
\[ \text{out\_label\_b}(e) := \text{edge\_b}(v,e,u) \]
\[ \text{out\_label\_c}(v,e,u) := \text{false} \]
\[ \text{out\_label\_ε}(v,e,u) := \text{edge\_c}(v,e,u) \]

\[ \text{rec}(\lambda(L, \_). \]
\[ \text{if } L = a \text{ then } \{d: \&\} \]
\[ \text{else if } L = c \text{ then } \& \]
\[ \text{else } \{L: \&\} \]
\]
UnCAL to MSO

$k$-Copying MSO Transduction [Courcelle94]

\[ \text{rec}(\lambda(L, _). \text{ if } L = a \text{ then } \{d: \{d: &\}} \text{ else if } L = c \text{ then } & \text{ else } \{L: &\}) \]
out\textsubscript{112}_edge(v,e,e) := \exists u. \text{edge}(v,e,u) \\
& \text{\& label}_a(e)

out\textsubscript{112}_label_d(e) := \exists v. \text{(same as above)}

out\textsubscript{231}_edge(e,e,u) := \exists v. \text{edge}(v,e,u) \\
& \text{\& label}_a(e)

out\textsubscript{231}_label_d(e) := \exists u. \text{(same as above)}

out\textsubscript{111}_edge(u,f,w) := \text{edge}(u,f,w) \& \neg \text{label}_a(f)

out\textsubscript{111}_label_d(f) := \exists u,w. \text{edge}(u,f,w) \& \text{label}_d(f)

...
rec(\((L, G). \{a: G\}\)(db))

out[000]_edge(v,e,u) := edge(v,e,u)
out[000]_label_xxx(e) := label_xxx(e)

out[110]_edge(v,e,u) := edge(v,e,u)
out[110]_label_a(e) := \exists v,u. edge(v,e,u)
out{111,101,011,...}_edge(v,e,u) := false
out{111,101,011,...}_label_a(e) := false
Output Schema

Basically as same as the input schema

use out_edge() and out_label() instead of edge() and label()

But we need to make it “copy-aware”

if k-copying

\[ \forall e. \phi \rightarrow \forall e. [\phi]_{e \rightarrow 1} \land [\phi]_{e \rightarrow 2} \land \ldots \land [\phi]_{e \rightarrow k} \]

where, e.g.,

\[ [\text{edge}(v,e,u)]_{v \rightarrow i, e \rightarrow j, u \rightarrow k} = \text{outijk\_edge}(v,e,u) \]
[Review] Our Approach: Use Logic

Covert to Logic Formula

Transformation $f$

Schema $S_I$

Schema $S_O$

$\phi = \text{“} G \in [[[S_I]]] \text{”} \Rightarrow \text{“} f(G) \in [[[S_O]]] \text{”}$

if $G$ satisfies $S_I$ then $f(G)$ satisfies $S_O$

Check whether $G \models \phi$ for all $G$
Theorem [Trakhtenbrot 50]: Validness property is **undecidable** on graphs, even for 1\(^{st}\)-order logic.

Check whether \( G \models \phi \) for all graph \( G \)
To be Decidable…

MSO validness is **decidable on finite trees** [Thatcher&Wright68] on **infinite trees** [Rabin69].

Also decidable on **tree-like graphs** (bounded tree-width), but it doesn’t help us much.
UnCAL is bisimulation-generic. [BFS ‘00]
Our Approach: Infinite Trees

UnCAL doesn’t distinguish bisimilar graphs
\[ G \equiv G' \quad \Rightarrow \quad f(G) \equiv f(G') \]
So are our schemas
\[ G \equiv G' \quad \Rightarrow \quad G \in [[s]] \quad \text{iff} \quad G' \in [[s]] \]
Every graph is bisimilar to its infinite unfolding
\[ G \equiv \text{unfold}(G) \in \text{Tree} \]

Decidable!!

\[ G \models \phi \quad \text{holds for any graph } G \quad \text{if and only if} \quad T \models \phi \quad \text{holds for any tree } T \]
Furthermore…

Core UnCAL is **compact**. [BFS ‘00]

![Diagram showing the process of cutting and preserving with functions](image-url)
Core UnCAL is compact. [BFS ‘00]

... (containing arbitrary long cut...)

f, modified to preserve $\prec$

$\text{Cut}$
Our schema is **compact**.

If cardinality is always $[0..\ast]$ or a graph doesn’t have $\varepsilon$-loops

```java
class T {
    c [0-\ast] : T
}
class CUT{
    \langle [1]:\{\} \ldots \}
class T {
    c [0-\ast] : T \lor CUT
}
```
root is T
class T {
b[0-∗] : T
a[0-∗] : {} }

root is S
class S {
c[0-∗] : S
a[0-∗] : {} }

\[ \text{rec}(\lambda(L,G). \begin{cases} 
  \{c: &\} & \text{if } L = b \\
  \{L: &\} & \text{else}
\end{cases})($db) \]
root is T ∨ CUT

root is S ∨ CUT

rec(\(\lambda(L,G).\) if \(L=b\) then \{c:&\} else \{$L:&\})(\$db)
Validation of Graph Transformation

Validness of MSO Formula on Graphs

\[ G \models "G \in [[S_I]]" \ \Rightarrow \ "f(G) \in [[S_O]]" \]

Validness of MSO Formula on Infinite Trees

\[ T \models "G \in [[S_I]]" \ \Rightarrow \ "f(G) \in [[S_O]]" \]

Validness of MSO Formula on Finite Trees

\[ t \models "t \in [[S_I \prec]]" \ \Rightarrow \ "f_\prec(t) \in [[S_O \prec]]" \]

Checked by the existing MSO Solver MONA
Summary

 Validation of UnCAL Graph Transformation

 Reduced to MSO Validity on graphs
   (generally undecidable)

 Bisimulation-Genericity allows to change the problem to that on (infinite) trees
   (decidable)

 Compactness allows to change the problem to that on finite trees
   (decidable, with existing good implementation)
Current Restriction
(that we want to get rid of)

- No Support for `isEmpty($G)` condition
  - It breaks Compactness of UnCAL
- No Support for Truly Nested Recursion
  - It breaks MSO-describability
- Cardinality limited to $[0-\ast]$}
  - It turned out to break Compactness of Schema
root is `T` 

```java
class T {
    a [0-*] : {}
    b [1] : T
}

class CUT {
    \& \& [1-*] : {} ...
}
```

root is `T ∨ CUT` 

```java
class T {
    a [0-*] : {}
    b [1] : T ∨ CUT
}
```