Graph Query Verification using Monadic 2\textsuperscript{nd}-Order Logic

Kazuhiro Inaba (稲葉 一浩)

kinaba@NII.ac.jp

Oct 10, 2010

1\textsuperscript{st} PKU-NII International Joint Workshop on Advanced Software Engineering
**Goal of This Research**

(Automated) Reasoning on Graph Transformations

Is this update on the output reflectable to the input?

Does the output of this transformation always have a desired structure?
Today’s Talk

• Given
  – A graph transformation \( f \)
  – Input schema \( S_I \)
  – Output schema \( S_o \)

• Statically verify that “there’s no type error”, i.e., “for any graph \( g \) conforming to \( S_I \), \( f(g) \) always conforms to \( S_o \).”
Extract all members using the screen-name “John”.

Example : SNS-Members

```sql
select {result: $x}
where
{SNS: {member: $x}},
{name: John} in $x
```
Extract all members using the screen-name “John”.

```
select {result: $x}
where
  {SNS: {member: $x}},
  {name: John} in $x
```
Lazy programmer may write ...

```sql
select {result: $x}
where
{ _*: $x},
{name: John} in $x
```
In fact, the graph contained “group” data, too!
What happens if there’s `{group: {name: John, ...}}`
Programmers specify their intention about the structure of input/output.

```cpp
// Input Schema supplied by the SNS provider
class INPUT { reference SNS: SNSDB; }
class SNSDB { reference member*: MEM; reference group*: GRP; }
class MEM { reference friend*: MEM; reference name: STRING; }
class GRP { reference name: STRING; reference member*: MEM; }
```
What We Provide

Then, our system automatically verify it!

class INPUT {
    reference SNS: SNSDB;
}

select {result: $x}
where
    {SNS: {member: $x}},
    {name: John} in $x

class OUTPUT {
    reference result*: MEM;
}

“OK!”

※ Our checker is SOUND. If it says OK, then the program never goes wrong.
What We Provide

Then, our system automatically verify it!

```java
class INPUT {
    reference SNS: SNSDB;
}

select {result: $x}
where
{_: $x},
{name: John} in $x

class OUTPUT {
    reference result*: MEM;
}
```

“BUG!”

※ Our checker provides a COUNTER-EXAMPLE.
By encoding transformations into a logic formula.

class INPUT {
  reference SNS: SNSDB; }

select {result: $x}
where
  {_*: $x},
  {name: John} in $x

class OUTPUT {
  reference result*: MEM; }


How?

Check VALIDNESS using an existing verifier
“YES” / “NO” + CE
Decode CE to a graph
“How should we represent schemas and transformations by logic formulas?”

– The logic must not be too strong
  (otherwise its validness becomes undecidable)

– The logic must not be too weak
  (otherwise it cannot talk about our schemas and transformations)

– What is the “just-fit” logic?
Rest of the Talk

• Our Choice
  – Monadic $2^{nd}$-Order Logic (MSO)

• Schema Language
  – How it can be represented in MSO

• UnCAL Transformation Language
  – How, in MSO

• Decide MSO: from Graph-MSO to Tree-MSO

• Discussion: Why This Approach
Monadic 2\textsuperscript{nd}-Order Logic

MSO is a usual 1\textsuperscript{st} order logic on graphs ...

(primitives) \( \text{edge}_{\text{foo}}(x, e, y) \) \( \text{start}(x) \)

(connectives) \( \neg P \) \( P \& Q \) \( P \lor Q \) \( \forall x. P(x) \) \( \exists x. P(x) \)

... extended with

(set-quantifiers) \( \forall^\text{set} S. P(S) \) \( \exists^\text{set} S. P(S) \)

(set-primitives) \( x \in S \) \( S \subseteq T \)
Graph Schema Language

• Programmers can specify any MSO-expressible property
  (as long as it is bisimulation-generic & compact)

• For example,
  (count-free subset of) KM3 MetaModeling Language:

```plaintext
class OUTPUT { reference result* : MEM; } class MEM { reference friend* : MEM; reference name+ : STRING; }
```
• We do need MSO’s expressiveness

\[
\exists \text{setOUTPUT. } \exists \text{setMEM. } \\
(\forall x. \text{start}(x) \rightarrow x \in \text{OUTPUT}) \\
\land (\forall x \in \text{OUTPUT. } \forall e. \forall u. \\
\text{edge}(x,e,u) \rightarrow \text{edge}_{\text{result}}(x,e,u) \land u \in \text{MEM}) \\
\land ... 
\]
Transformation Language

- UnCAL [Buneman et al, 2000]
  - Internal Representation of “UnQL”

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

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

select \{result: $x\} \\
where \\
\{_*: $x\}, \\
\{name: John\} in $x
“Bulk” Semantics of UnCAL

\[
\text{rec}(\lambda(L,G). \\
\quad \text{if } L = a \text{ then } \{b: \{c: \&\}\} \\
\quad \text{else } \{d: G\})($input\_db$)
\]
More Precise, MSO-Representable
“Finite-Copy” Semantics

\[
\text{edge}_b(v,e,u) \iff \exists v' e' u'. \text{edge}_a(v',e',u') \land v = v'_1 \land e = e'_1 \land u = e'_2
\]

\[
\text{edge}_c(v,e,u) \iff \exists v' e' u'. \text{edge}_a(v',e',u') \land v = e'_2 \land e = e'_3 \land e = u'_1
\]
“Finite-Copy” Semantics

\[
\text{edge}_d(v,e,u) \iff \\
\exists v', e', u'. \neg \text{edge}_a(v',e',u') \land v = v' \land e = e' \land u = u'
\]
Transformation to MSO

Theorem:
Nest-free UnCAL is representable by finite-copying MSO transduction.

\[
\text{rec}(\lambda(L, G). \begin{cases}
\text{if } L = a & \text{then } \{b: \{c: \&\}\} \\
\text{else} & \{d: G\}
\end{cases})(\text{input}_{db})
\]

\[
\begin{align*}
\text{edge}_b(v, e, u) & \iff \text{edge}_a(v', e', u') \land v = v_1' \land e = e_1' \land u = e_2' \\
\text{edge}_c(v, e, u) & \iff \text{edge}_a(v', e', u') \land v = e_2' \land e = e_3' \land e = u_1' \\
\text{edge}_d(v, e, u) & \iff \neg \text{edge}_a(v', e', u') \land v = v_1' \land e = e_1' \land u = u'
\end{align*}
\]
"Backward" Inference [Courcelle 1994]

MSO Formula stating
“output conforms to the schema”
in terminology of OUTPUT GRAPHS

```
class OUTPUT { reference result*: MEM; }
class MEM { reference friend*: MEM;
  reference name: STRING; }
```

\[
\exists \text{set OUTPUT. } \exists \text{set MEM. }
(\forall x. \text{start}(x) \rightarrow x \in \text{OUTPUT}) \\
\land (\forall x \in \text{OUTPUT. } \forall e. \forall u.
\text{edge}(x,e,u) \rightarrow \text{edge}_{\text{result}}(x,e,u) \land u \in \text{MEM}) \land ...
\]

OUTPUT GRAPH description
by the INPUT GRAPH

rec(\lambda($L, \$G). 
  if $L = a$ then \{b: \{c: &\} \} else \{d: $G\}
 )($\text{input_db}$)

\[
\begin{align*}
\text{edge}_b(v,e,u) & \iff \text{edge}_a(v',e',u') \land v = v'_1 \land e = e'_1 \land u = u'_2 \\
\text{edge}_c(v,e,u) & \iff \text{edge}_a(v',e',u') \land v = e'_2 \land e = e'_3 \land e = u'_1 \\
\text{edge}_d(v,e,u) & \iff \neg \text{edge}_a(v',e',u') \land v = v'_1 \land e = e'_1 \land u = u'_1
\end{align*}
\]

Verify this is valid for any INPUT GRAPHS!!

MSO Formula stating
“output conforms to the schema”
in terminology of INPUT GRAPHS: edge(v',e',u')

OUTPUT GRAPH
description
by the INPUT GRAPH

\[
\begin{align*}
\text{edge}_b(v,e,u) & \iff \text{edge}_a(v',e',u') \land v = v'_1 \land e = e'_1 \land u = u'_2 \\
\text{edge}_c(v,e,u) & \iff \text{edge}_a(v',e',u') \land v = e'_2 \land e = e'_3 \land e = u'_1 \\
\text{edge}_d(v,e,u) & \iff \neg \text{edge}_a(v',e',u') \land v = v'_1 \land e = e'_1 \land u = u'_1
\end{align*}
\]
Note: Harder Case

- Nested Recursion (arising from “cross product”) cannot be encoded into finite-copy semantics

```plaintext
select {p: {f: $G1, s:$G2}}
where {_: $G1} in $db,
  {_: $G2} in $db

rec(λ($L1,$G1).
  rec(λ($L2,$G2).
    {pair: {first: $G1, second: $G2}}
  )($db))($db)

Currently we ask programmer to add annotation ➔

rec(λ($L1,$G1). rec(λ($L2,$G2).
  {pair: {first: ($G1 :: MEM),
          second: $G2}} ...
```
(why we need the power of MSO?)

• E.g., for “regular path pattern”

```
select {result: $x}
where
{ _*: $x},
{name: John} in $x
```

```
select {result: $x}
where
{ (a|b).(c|d)*: $x},
{name: John} in $x
```

• MSO can encode finite automata

\[ \exists \text{set } Q_1. \exists \text{set } Q_2. \ldots \exists \text{set } Q_n. \text{ “there is a run of the automaton that reaches states Q’s on each node”} \]
Now we have a **MSO Formula on Graphs**.

**MSO (even 1st-Order Logic) on Graphs is undecidable** [Trakhtenbrot 1950].

**MONA [Henriksen et al., 1995]**

can decide **validness of MSO on Finite Trees**.
Two Nice Props of UnCAL

[Buneman et al. 2000] UnCAL is ...

Unfolding

Bisimulation-generic

Compact

UnCAL Transformation

Cut
Now we have a **MSO Formula on Graphs**.

**MONA** [Henriksen et al., 1995] can decide validness of **MSO on Finite Trees**.

**MSO** (even 1\textsuperscript{st}-Order Logic) on Graphs is undecidable [Trakhtenbrot 1950].

**Theorem:** If MSO formula is Bisimulation-Generic and Compact, it is valid on graphs iff on finite trees.
Discussion: Choice of Logic

• **MSO**
  - Powerful, yet decidable *(if we fully utilize bisimulation)*

• **FO+TC** *(1\textsuperscript{st}-Order Logic + Transitive Closure)*
  - Very powerful; express all UnCAL without annotation
  - Undecidable, even on finite trees

• **FO** *(1\textsuperscript{st}-Order)*, **SMT** *(Satisfiability Modulo Theory)*
  - Very good solvers
  - Too weak for schemas or UnCAL; cannot use repetition

• **mu-Calculus** *(Modal Logic with Fixpoint)*
  - Theoretically, equal to MSO under bisimulation
  - Not clear how to represent transformations
Discussion: Approach

• Our Approach
  – “Backward” (define the output by the input, with logic)

• Other Possible Approaches
  – “Forward” (e.g., abstract interpretation)
    • [Buneman et al., 1997] [Nakano, Today!]
    • Better in range analysis. Worse in counterexample generation.
  – “Type System”
    • Hard, because we need context-dependent types, etc.
    • Two $G$ may differ in types: if $L=a$ then ...$G$... else ...$G$...
Verify “type-correctness” of graph transformations via Monadic 2nd-Order Logic

- **MSO** and **bisimulation** are good tools for graphs!
- Future work: checking other properties