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

Kazuhiro Inaba
with S. Hidaka, Z. Hu, H. Kato
(National Institute of Informatics, Japan)
and K. Nakano
(University of Electro-Communications)

PPDP 2011, Odense
Graph Transformation

GRoundTram (www.biglab.org)
Two Languages Involved

Transformation:
UnQL / UnCAL [Buneman & Fernandez & Suciu, 2000]

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

Schema: KM3 [ATLAS Group]

class INPUT
{ reference SNS: SNSDB; ... }

class OUTPUT
{ reference result*: MEM; }
Today’s Topic: Static Check

• 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 \( S_o \).”
Example: SNS-Members

Extract all members using the screen-name “John”.

```
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 ...

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; }
```
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; }
```

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

Then, our system automatically verify it!

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.
Outline of the Rest of the Talk

How We Implemented This Verification

– Monadic 2\textsuperscript{nd}-Order Logic (MSO)
– Schema to MSO
– Transformations to MSO
– Decide MSO: from Graphs to Trees
Overall Picture

KM3 Schema [ATLAS Group]

UnQL / UnCAL [Buneman, et.al. 00]

Nice Properties

This Work

MONA : MSO Solver [Møller, et.al. 95-]

MSO Logic

MSO Definable Transduction [Courcelle 94]

Bwd Inference
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) \( x \in S \) \( S \subseteq T \)
Schema to MSO

• Straightforward

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

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

Q: What do we mean by “representing transformations in MSO”?

A: We convert UnQL’s functional core language into a (kind of) logic program in MSO.
Transformation Language

• “Core UnCAL”
  – Internal Representation of “UnQL”
    
    \[ E ::= \begin{align*} & \{ L_1 : E_1, \ L_2 : E_2, \ldots, \ L_n : E_n \} \\ & \quad \text{if } L = L \text{ then } E \text{ else } E \\ & \quad $G \\ & \quad & \\ & \quad \text{rec}(\lambda($L, $G). \ E)(E) \quad | \quad \ldots \end{align*} \]

\[ L ::= (label \ constant) \\ | \quad $L \]
Semantics of `rec` in UnCAL

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

Decompose to a set of edges!

Glue them!
More Precise, MSO-Representable “Finite-Copy” Semantics

if $L = a$ then $\{b: \{c: \&}\}$
else $\{d: \$G\}$

Copy as needed!

Glue them!

Transform to what we want!

\[
\text{edge}[112]_b (v, e, u) \iff \\
\exists v', e', u'. \text{edge}_a(v', e', u') \& v = v' \& e = e' \& u = e'
\]

\[
\text{edge}[231]_c (v, e, u) \iff \\
\exists v', e', u'. \text{edge}_a(v', e', u') \& v = e' \& e = e' \& u = u'
\]
“Finite-Copy” Semantics

if $L = a$ then \{b: \{c: &\}\} else \{d: $G$\}

copy as needed!

glue them!

transform to what we want!

$edge_{110}(v, e, u) \iff \exists v' e' u'. \neg edge_a(v', e', u') \land v = v' \land e = e' \land u = u'$
Theorem:

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

\[
\text{rec}(\lambda(L,G). \begin{cases} \text{if } L = a & \exists v' e' u'. \text{edge}_a(v',e',u') \land v = v' \land e = e' \land u = u' \\ \text{else } \{d: $G}\end{cases})(\text{input}_\text{db})
\]

\[
\text{edge}[112]_b (v, e, u) \iff \\
\exists v' e' u'. \text{edge}_a(v',e',u') \land v = v' \land e = e' \land u = e'
\]

\[
\text{edge}[231]_c (v, e, u) \iff \\
\exists v' e' u'. \text{edge}_a(v',e',u') \land v = e' \land e = e' \land u = u'
\]

\[
\text{edge}[110]_d (v, e, u) \iff \\
\exists v' e' u'. \neg \text{edge}_a(v',e',u') \land v = v' \land e = e' \land u = u'
\]

((Transformation = Definition of the output-graph in terms of the input graph))
"Backward" Inference [Courcelle 1994]

Transformation

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

Output Schema

\[ \exists e. \text{edge}_c(_, e, _) \]

\[ \exists e. \text{edge}[112]_b (v, e, u) \iff \exists v' e' u'. \ldots \]

\[ \text{edge}[231]_c (v, e, u) \iff \exists v' e' u'. \text{edge}_a(v',e',u') \& \ldots \& e=e' \]

\[ \exists e. \text{edge}[110]_d (v, e, u) \iff \exists v' e' u'. \ldots \]

Input Schema

\[ \exists e. \text{edge}_a(_, e, _) \]

\[ \exists e. \text{false} \]

\[ \lor \text{false} \]

\[ \lor \exists v' e' u'. \text{edge}_a(v',e',u') \& \ldots \& e=e' \]
Nested rec

• Nested **rec** (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 ➔

```plaintext
rec(λ($L1,$G1). rec(λ($L2,$G2).
    {pair: {first: ($G1 :: MEM),
        second: $G2}}
 ...)
```
Remark: Why MSO

• It is expressive power is needed.
  – Schemas can encode runs of automata, which is basically equivalent to MSO.
  – Transformation also requires MSO power, for tracking *edge-erasing*.

• It can be made decidable!
  – In contrast to, e.g., FO+TC^k that can capture nested *rec* without annotation.
Two Nice Props of UnCAL

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

UnCAL Transformation

Unfolding

Bisimulation-generic

Unfolding

Cut

Compact

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

**MONA MSO Solver** [Møller, et.al. 95-] can decide validness of **MSO on Finite Trees**.

**MSO (even 1st-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.
Conclusion

Static verification of graph transformations via MSO

- Future work:
  - Complete checking w/o annotations.
  - Support for full UnCAL (with data value comparison).
  - Use MSO-Transduction semantics for checking other properties.
  - Comprehensive experiments on performance.

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

class OUTPUT {
  reference result*: MEM;
}

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

```

"YES"/ "NO"+