Graph-Transformation Verification using Monadic 2nd-Order Logic

Kazuhiro Inaba

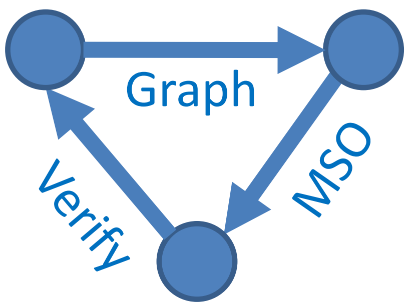
with S. Hidaka, Z. Hu, H. Kato

(National Institute of Informatics, Japan)

and K. Nakano

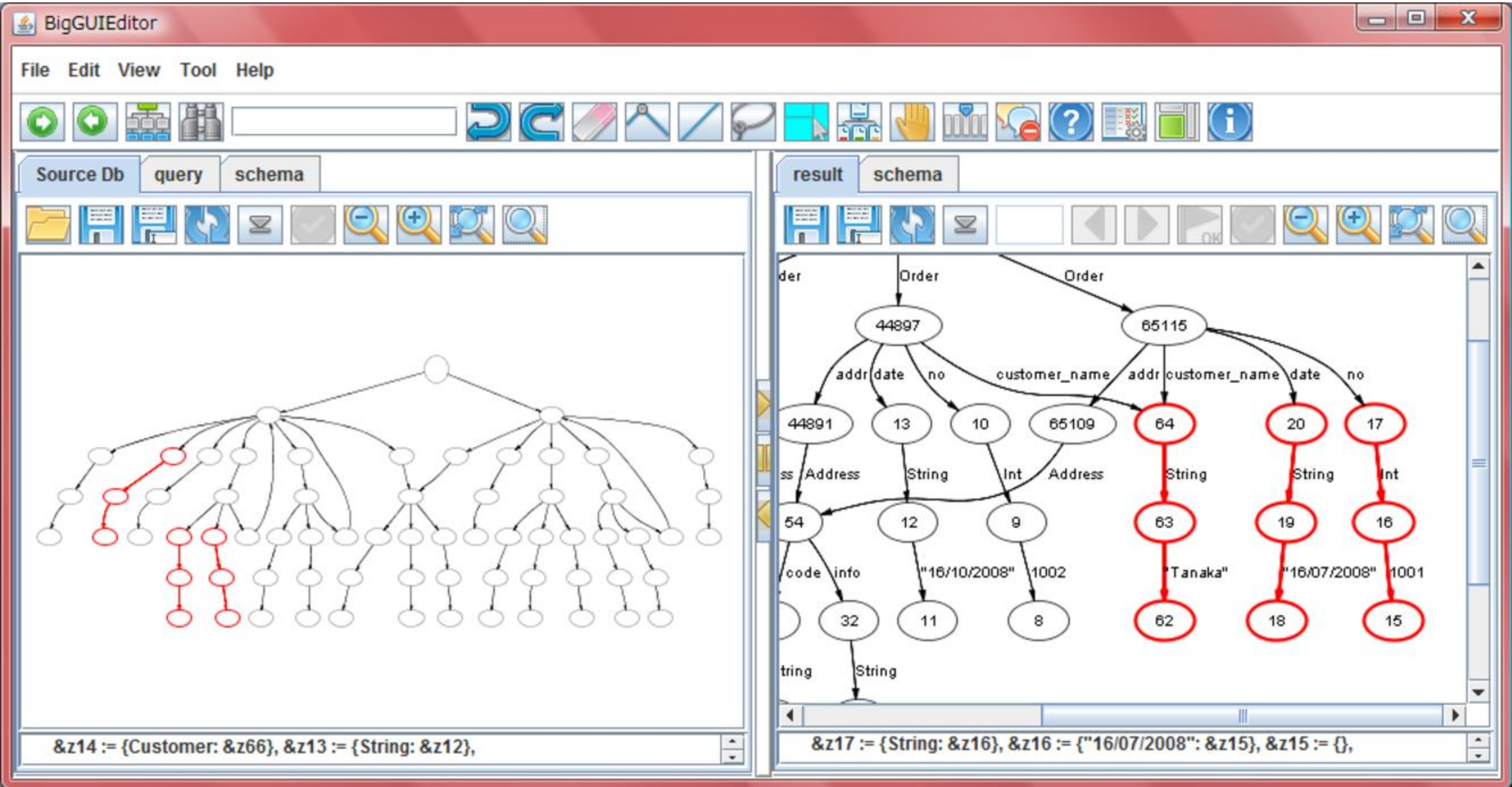
(University of Electro-Communications)

PPDP 2011, Odense



Graph Transformation

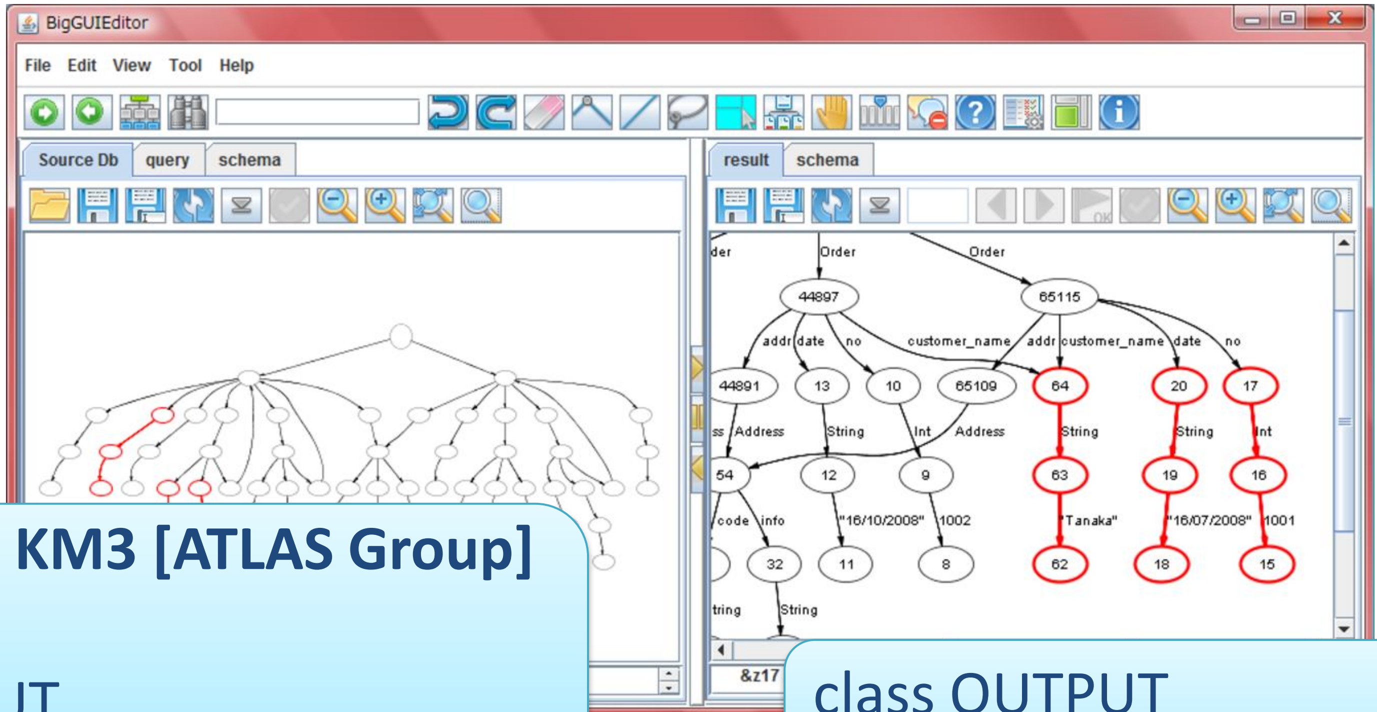
GRoundTram (www.biglab.org)



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; … }

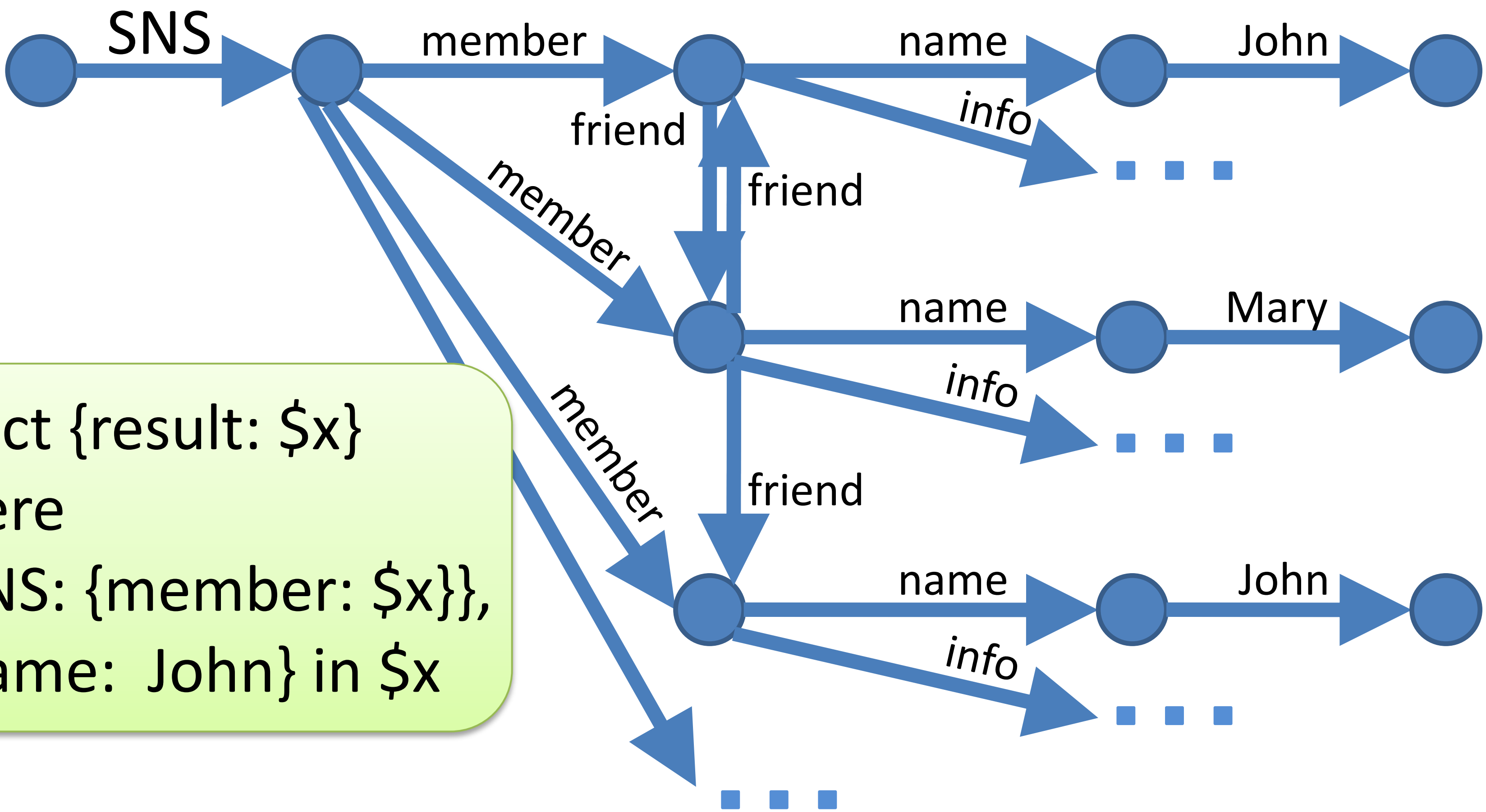
class OUTPUT
{ reference result*: MEM; }

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

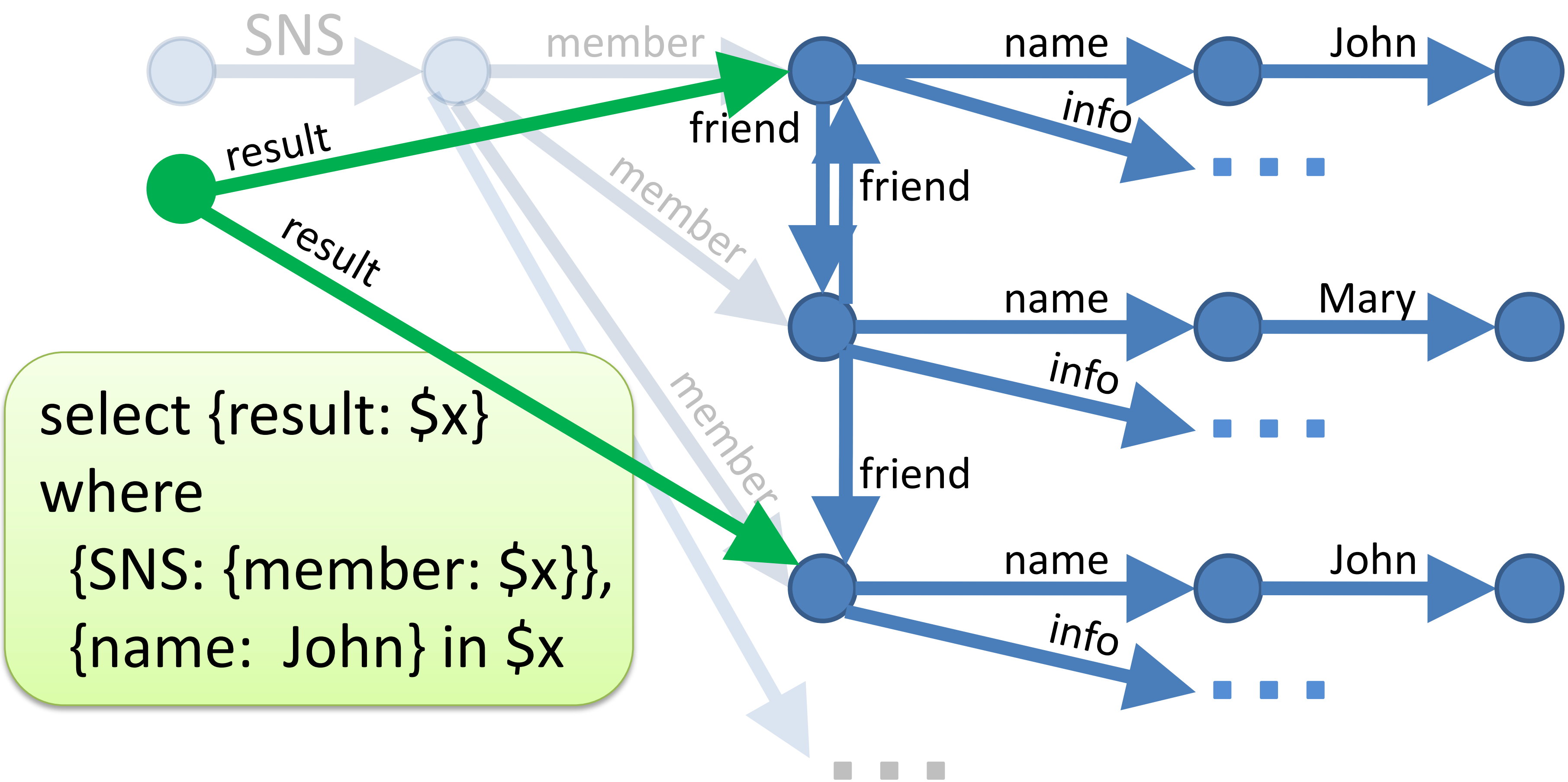
Extract all members using the screen-name “John”.




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

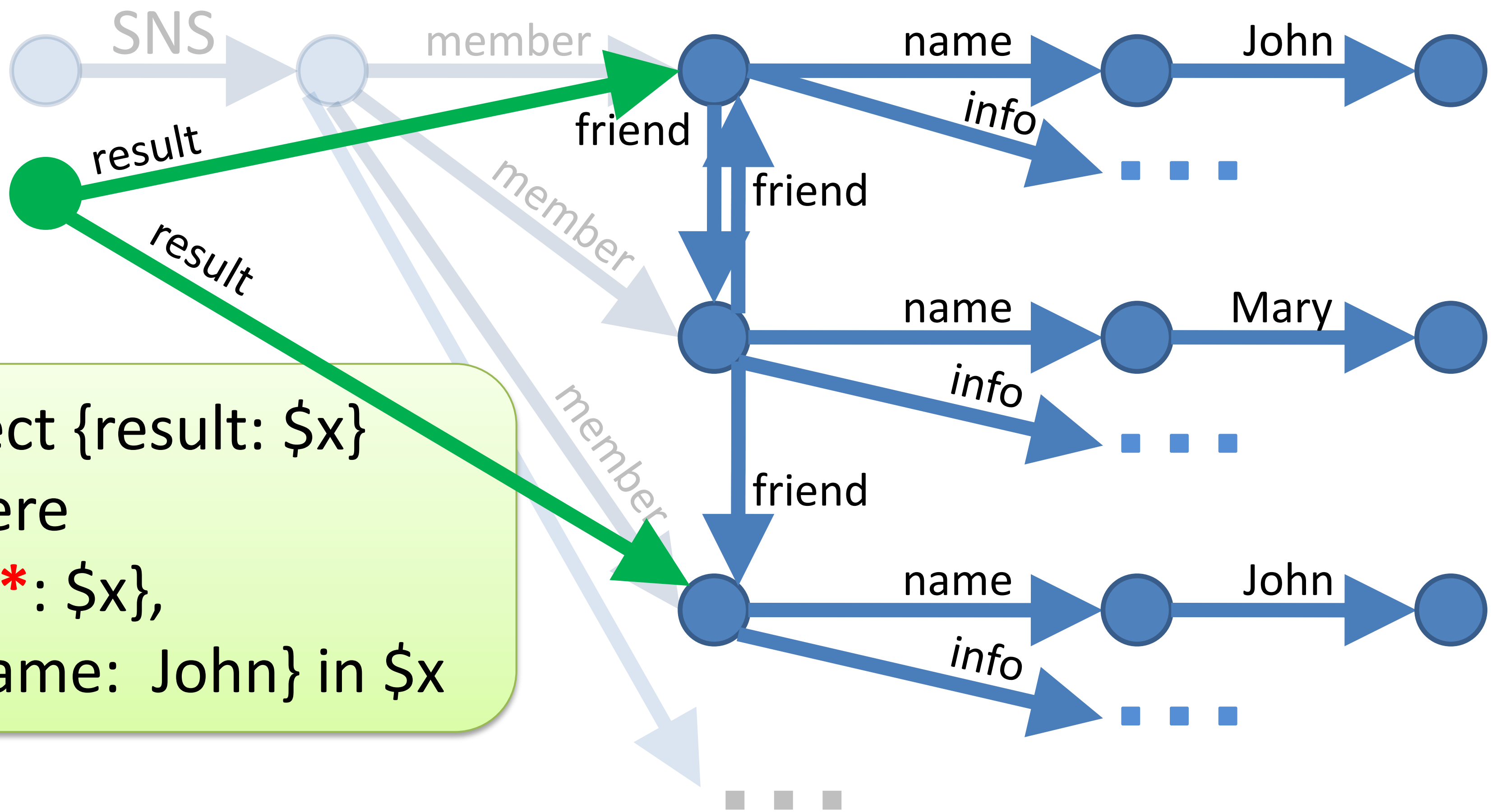
Example

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



Example

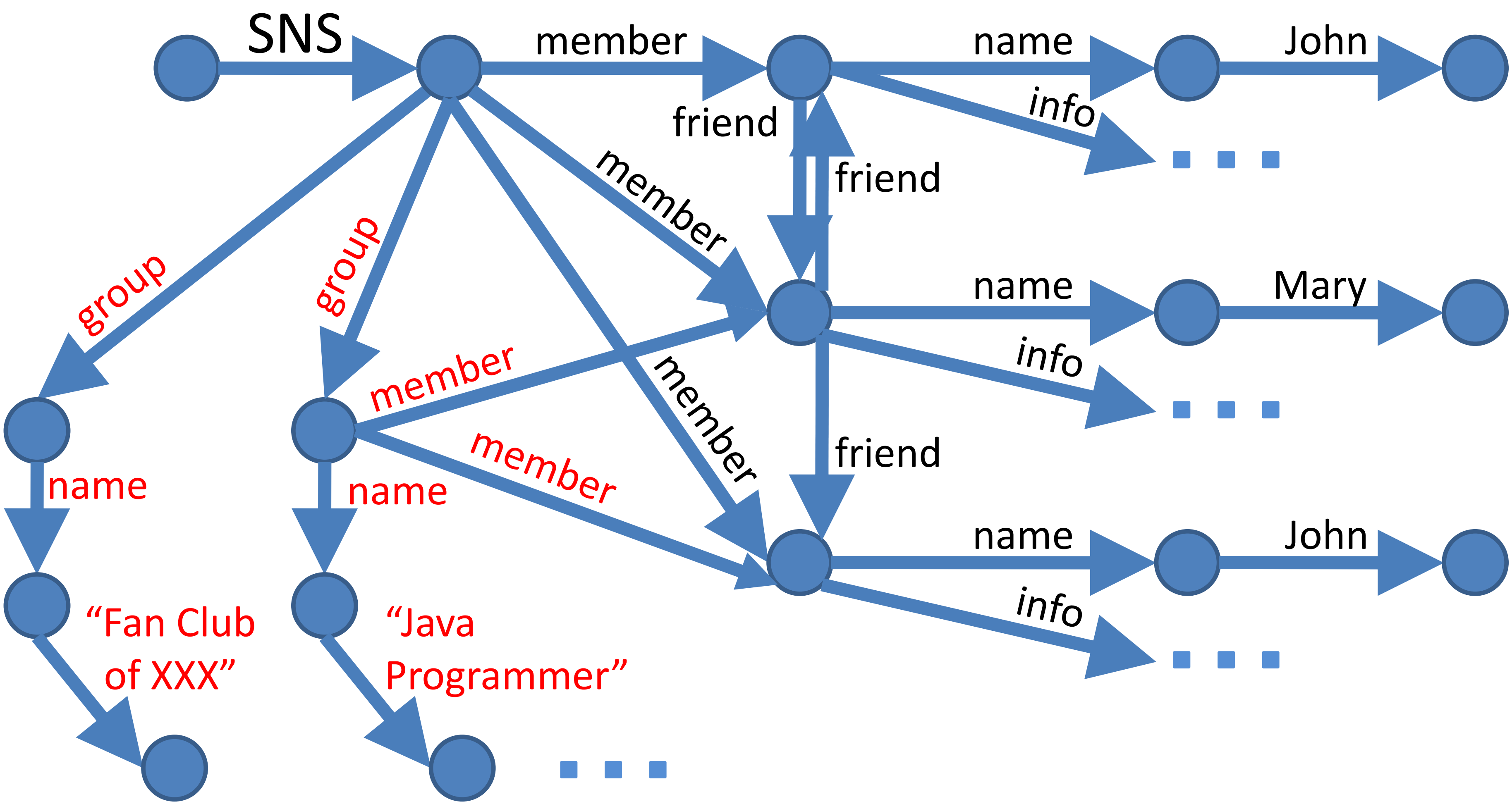
Lazy programmer may write …




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

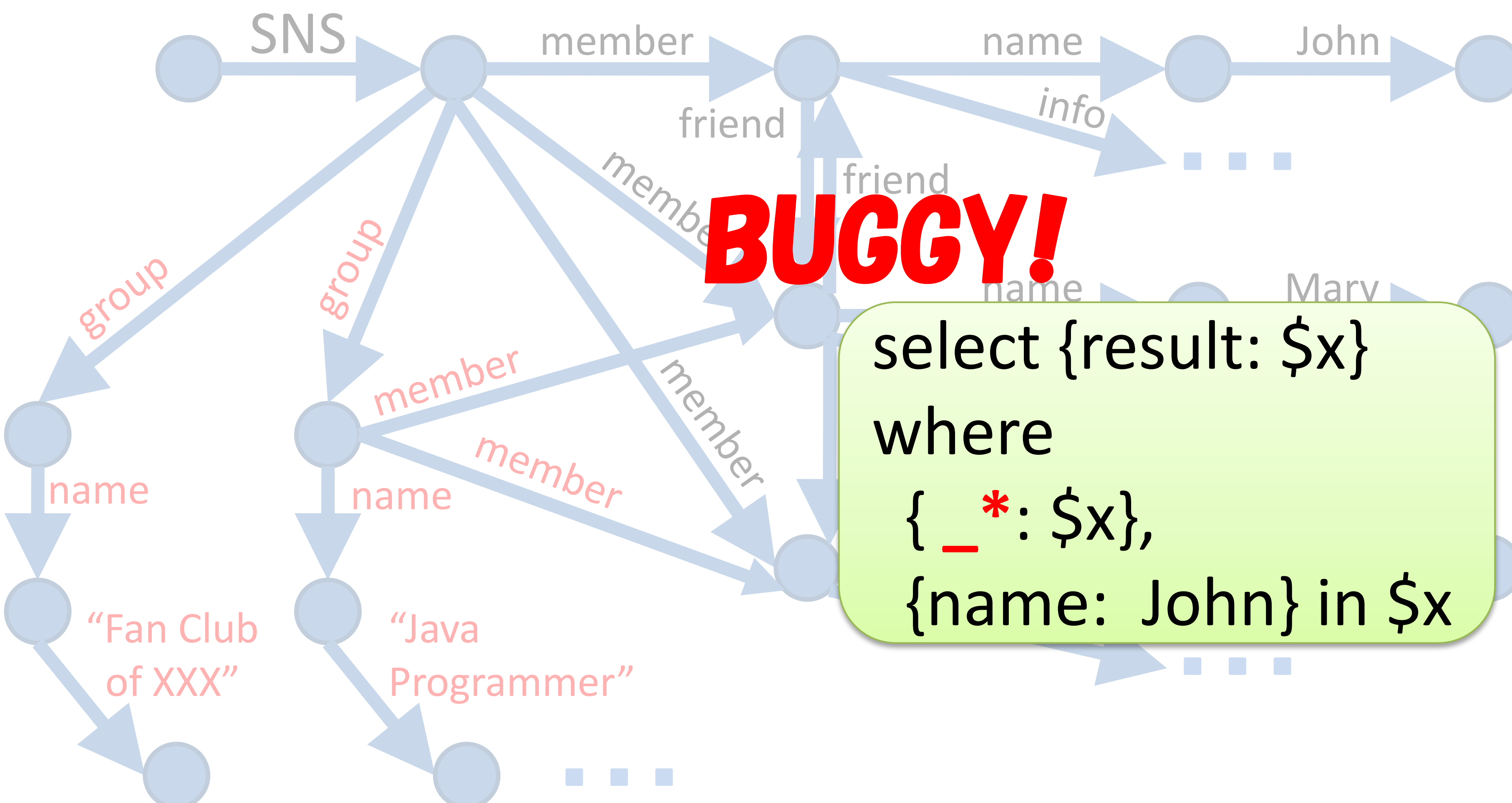
Example

In fact, the graph contained “group” data, too!



Example

What happens if there's {group: {**name: John**, …}}




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

What We Provide

Programmers specify their intention about the structure of input/output.

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

```
// 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

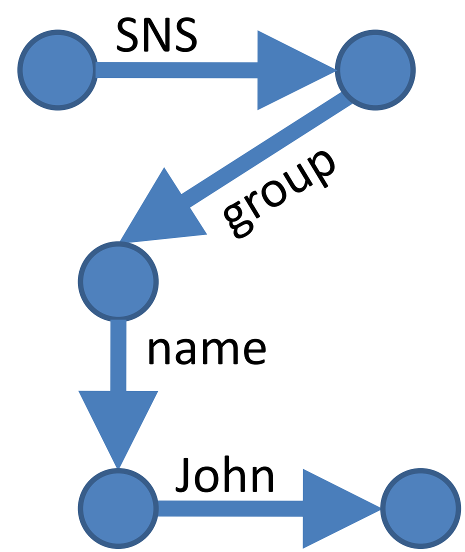
Then, our system automatically verify it!

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

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

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

"BUG!"



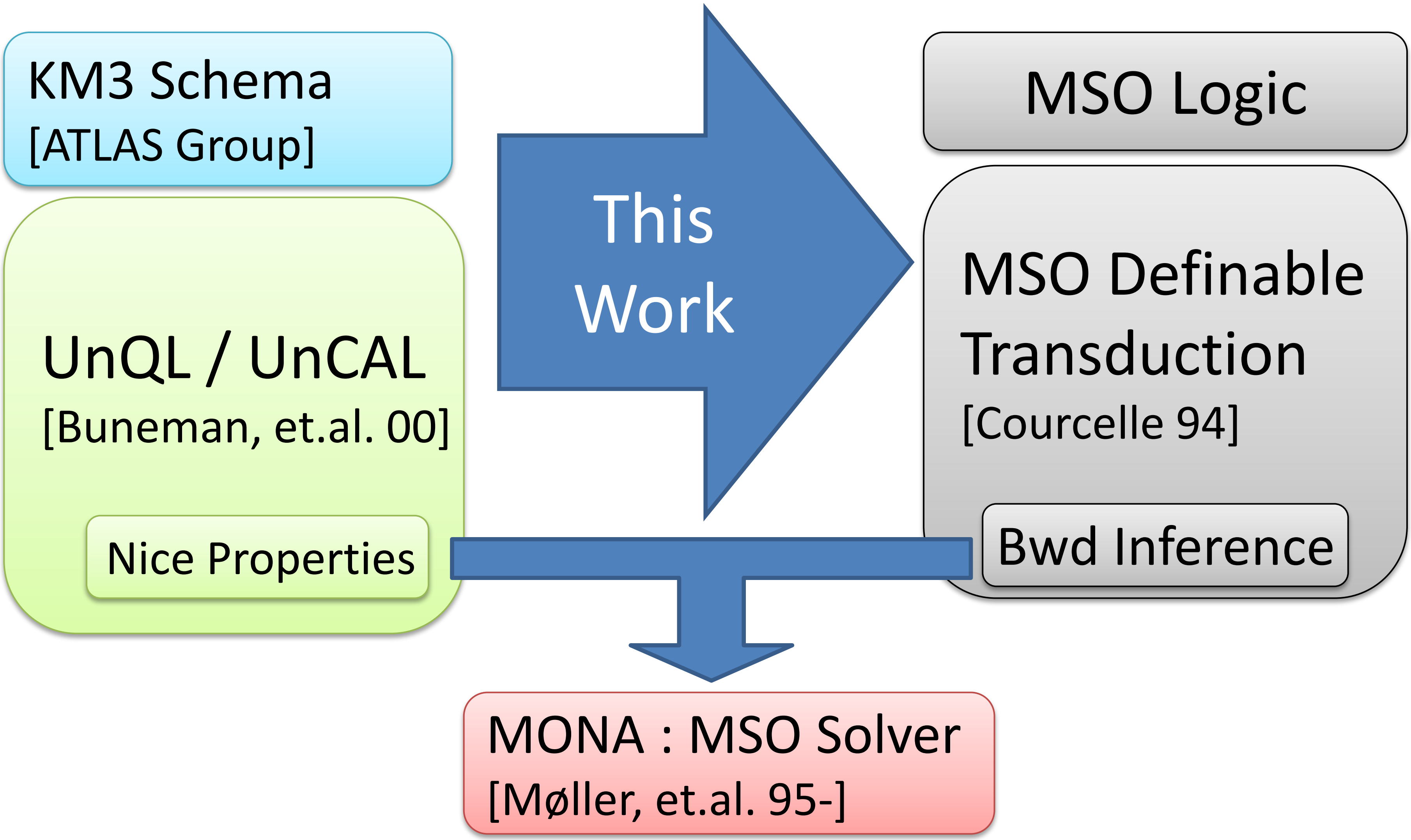
※ Our checker provides a COUNTER-EXAMPLE.

Outline of the Rest of the Talk

How We Implemented This Verification

- Monadic 2nd-Order Logic (MSO)
- Schema to MSO
- Transformations to MSO
- Decide MSO: from Graphs to Trees

Overall Picture



Monadic 2nd-Order Logic

MSO is a usual 1st order logic on graphs …

(primitives)	$edge_{foo}(x, e, y)$ $start(x)$
(connectives)	$\neg P$ $P \& Q$ $P \vee Q$ $\forall x.P(x)$ $\exists x.P(x)$

… extended with

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

$$\exists^{set}\mathbf{OUTPUT}.\quad \exists^{set}\mathbf{MEM}.$$
$$(\forall x.\ start(x) \rightarrow x \in OUTPUT)$$
$$\wedge\ (\forall x \in OUTPUT.\ \forall e.\ \forall u.$$
$$edge(x,e,u) \rightarrow edge_{result}(x,e,u)\ \&\ u \in MEM)$$
$$\wedge\ \ldots$$

Transformation to MSO

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

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

$\mathrm{edge[OUT]}_b\,(v, e, u) \Leftrightarrow$

$\quad \exists v'\ e'\ u'.\ \mathrm{edge}_a(v',e',u')\ \&\ v=v'\ \&\ e=e'\ \&\ u=e'$

$\mathrm{edge[OUT]}_d\,(v, e, u) \Leftrightarrow$

$\quad \exists v'\ e'\ u'.\ \neg\mathrm{edge}_a(v',e',u')\ \&\ v=v'\ \&\ e=e'\ \&\ u=u'$

…

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

Transformation Language

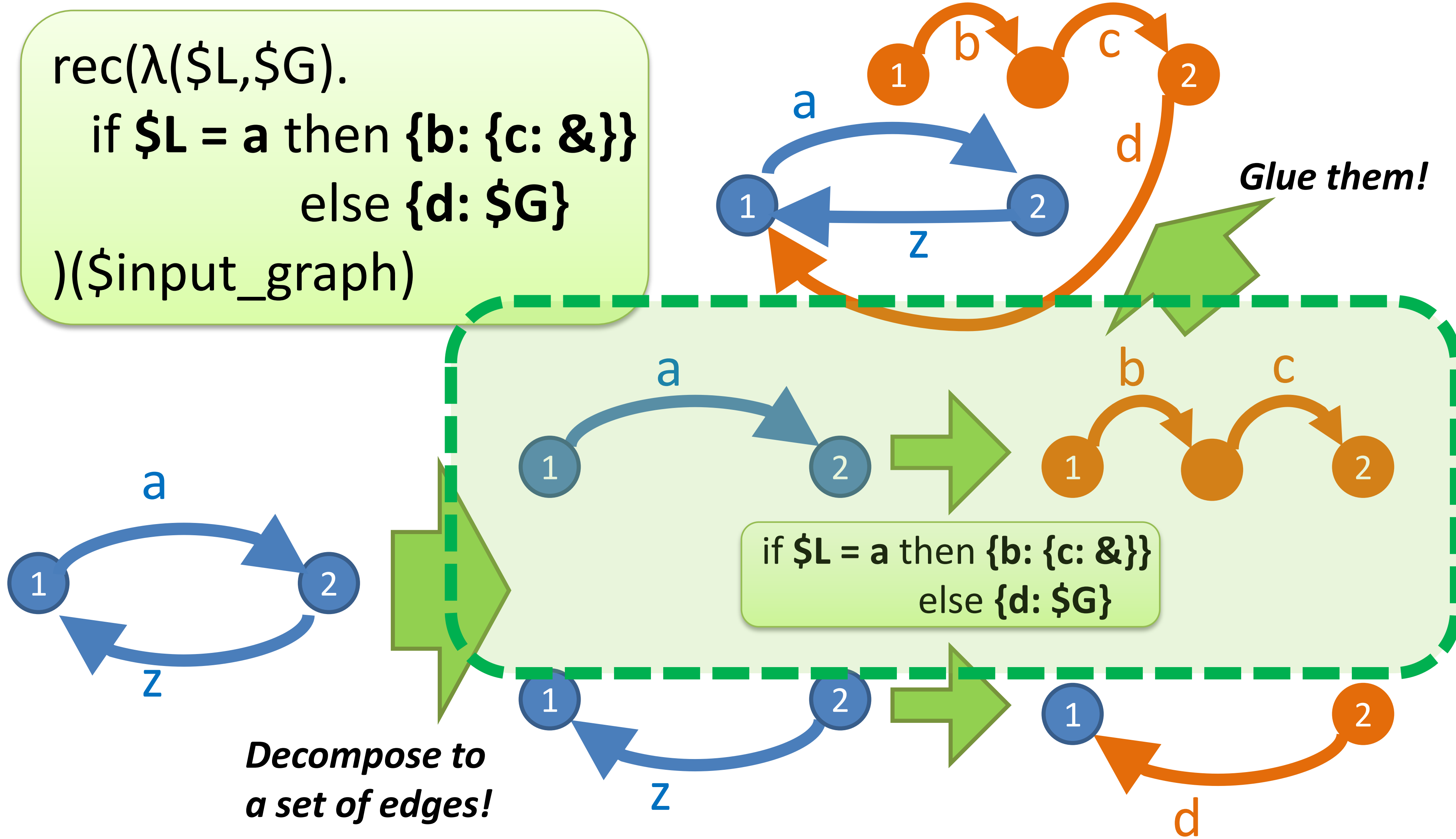
- "Core UnCAL"
  - Internal Representation of "UnQL" ➔

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

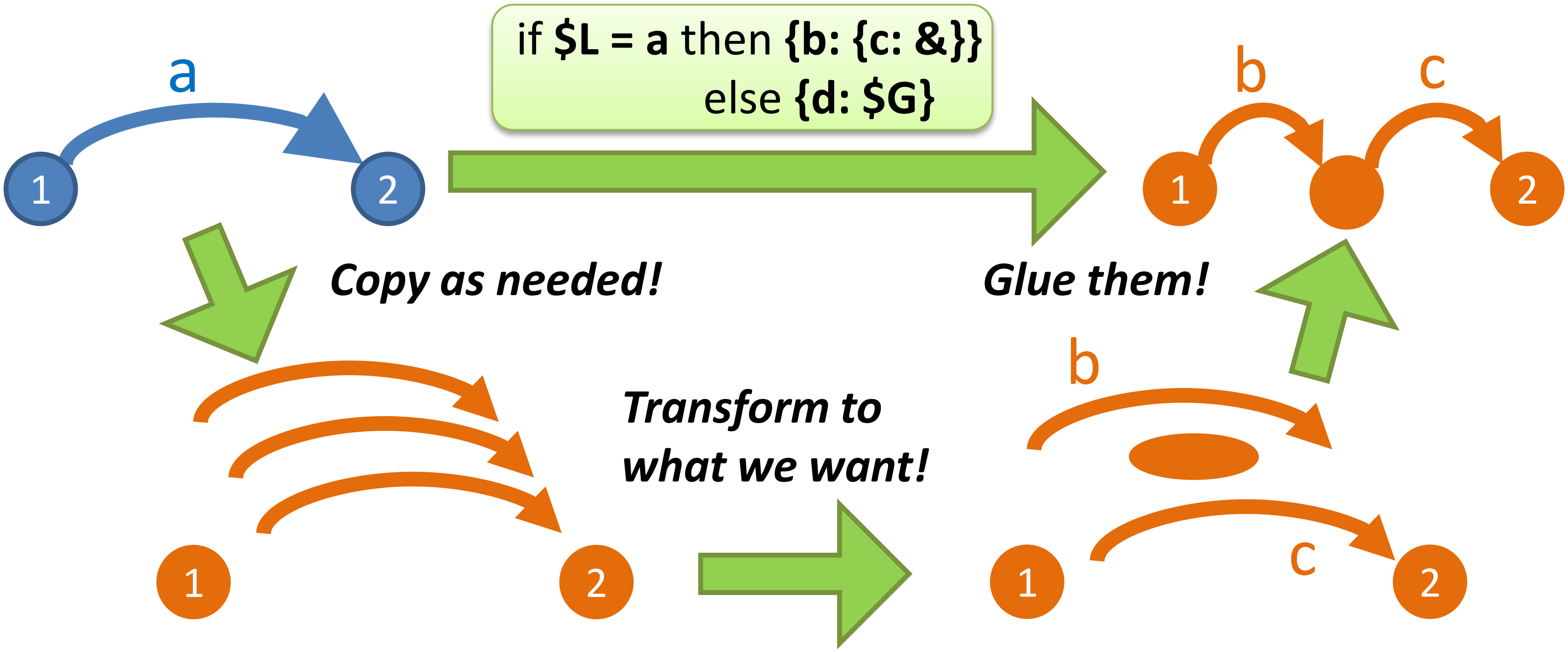
```
E ::= {L1:E1, L2:E2, …, Ln:En}
    | if L=L then E else E
    | $G
    | &
    | rec(λ($L,$G). E)(E)  | …
L ::= (label constant)
     | $L
```

Semantics of **rec** in UnCAL

rec(λ($L,$G).
  if **$L = a** then **{b: {c: &}}**
          else **{d: $G}**
)($input_graph)

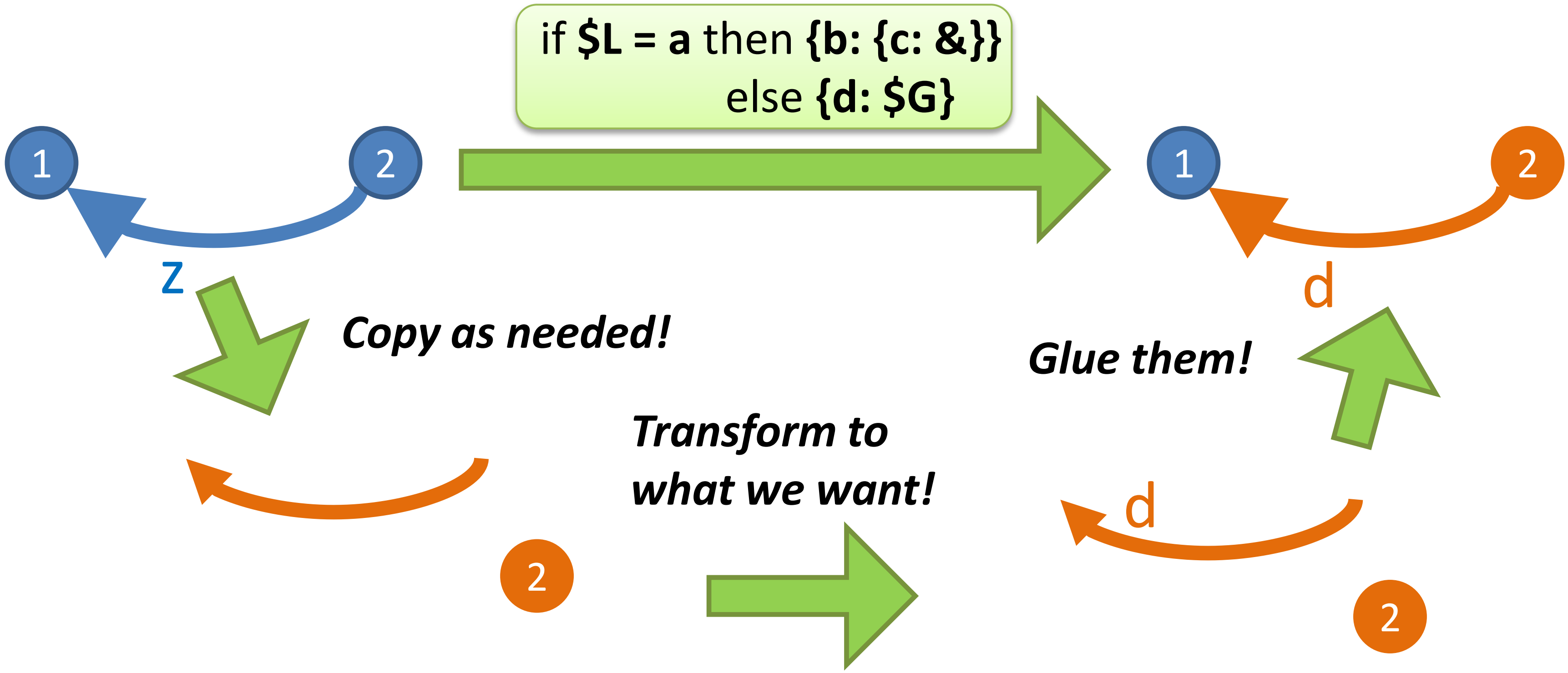


More Precise, MSO-Representable "Finite-Copy" Semantics



$$\text{edge}[112]_b(v, e, u) \Leftrightarrow$$
$$\exists v'\ e'\ u'.\ \text{edge}_a(v',e',u')\ \&\ v=v'\ \&\ e=e'\ \&\ u=e'$$
$$\text{edge}[231]_c(v, e, u) \Leftrightarrow$$
$$\exists v'\ e'\ u'.\ \text{edge}_a(v',e',u')\ \&\ v=e'\ \&\ e=e'\ \&\ u=u'$$

“Finite-Copy” Semantics



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

Transformation to MSO

Theorem:

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

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

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

$\mathrm{edge}[231]_c(v, e, u) \Leftrightarrow$
$\exists v'\ e'\ u'.\ \mathrm{edge}_a(v',e',u')\ \&\ v=e'\ \&\ e=e'\ \&\ u=u'$

$\mathrm{edge}[110]_d(v, e, u) \Leftrightarrow$
$\exists v'\ e'\ u'.\ \neg\mathrm{edge}_a(v',e',u')\ \&\ v=v'\ \&\ e=e'\ \&\ u=u'$

((Transformation = Definition of the output-graph in terms of the input graph))

"Backward" Inference [Courcelle 1994]

Transformation

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

$edge[112]_b (v, e, u) \Leftrightarrow \exists v'\ e'\ u'. \ldots$

$edge[231]_c (v, e, u) \Leftrightarrow$

$\exists v'\ e'\ u'.\ edge_a(v',e',u')\ \&\ \ldots\ \&\ e=e'$

$edge[110]_d (v, e, u) \Leftrightarrow \exists v'\ e'\ u'. \ldots$

Output Schema

$\exists e.\ \ edge_c(\_, e, \_)$

$\exists e.\ \ edge[\_0\_]_c (\_, e, \_)$
$\vee\ edge[\_1\_]_c (\_, e, \_)$
$\vee\ edge[\_2\_]_c (\_, e, \_)$
$\vee\ edge[\_3\_]_c (\_, e, \_)$

$\exists e.\ \ false$
$\vee\ false$
$\vee\ false$
$\vee\ \exists v'\ e'\ u'.\ edge_a(v',e',u')\ \&\ \ldots\ \&\ e=e'$

InputSchema

$\exists e.\ \ edge_a(\_, e, \_)$ $\Rightarrow$

Nested **rec**

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

```
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}} …
```

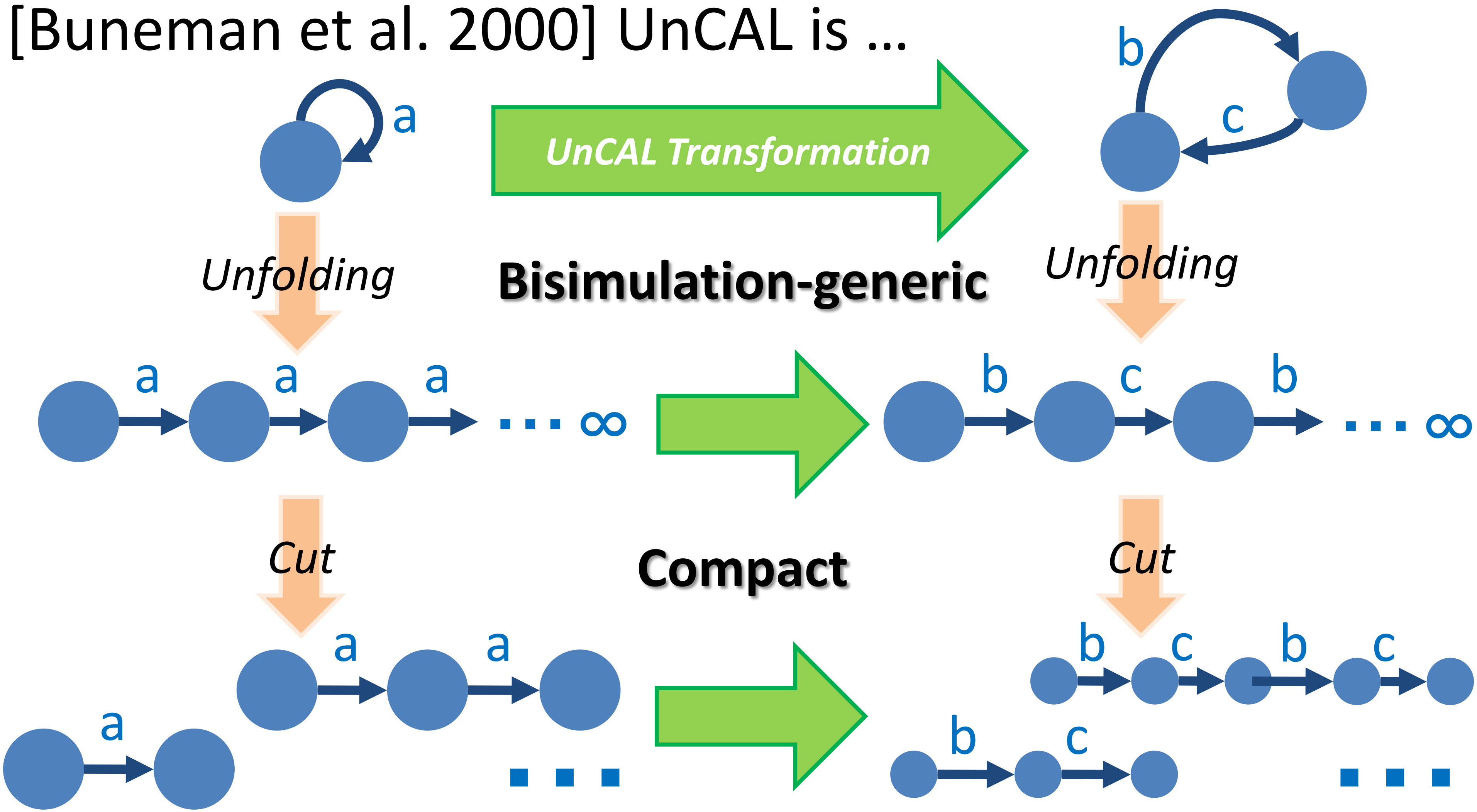
Remark: Why MSO

- 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**s without annotation.

Two Nice Props of UnCAL

[Buneman et al. 2000] UnCAL is …



MSO Validation

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

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

!

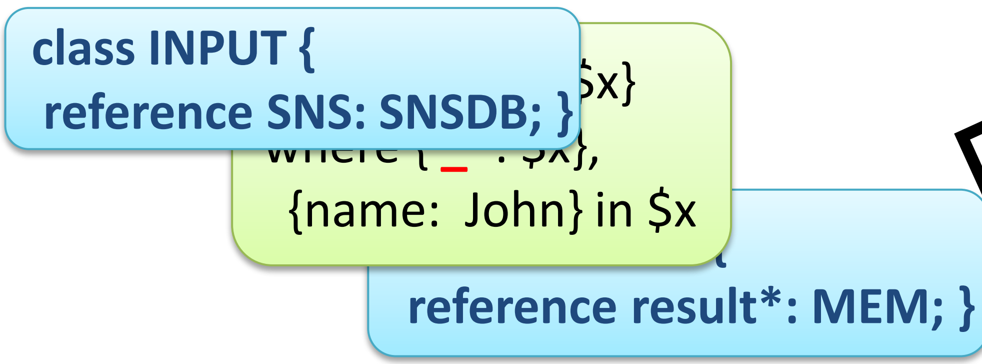
Theorem: If MSO formula is Bisimulation-Generic and Compact, it is valid on graphs iff on finite trees.

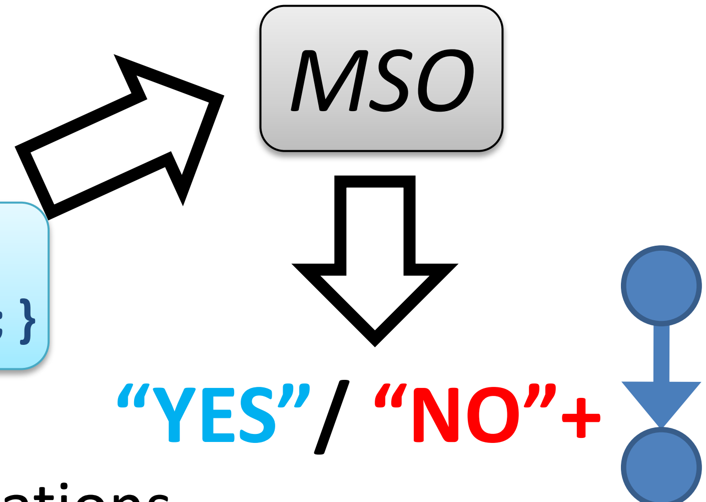
MONA MSO Solver [Møller, et.al. 95-]

can decide validness of **MSO 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.