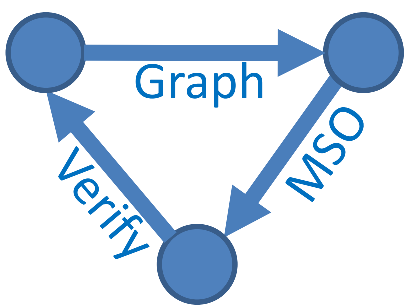
Graph Query Verification using Monadic 2nd-Order Logic

Kazuhiro Inaba (稲葉 一浩)

kinaba@NII.ac.jp

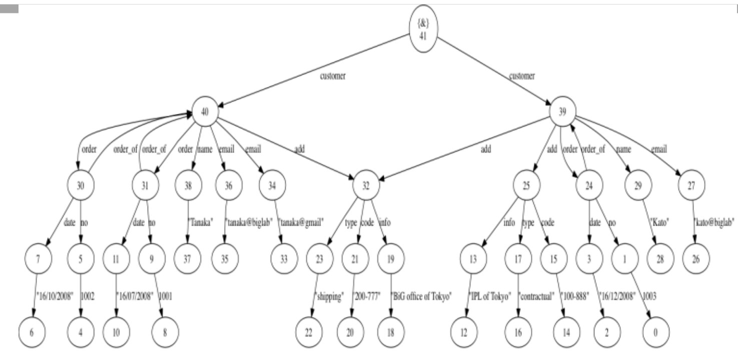


Oct 10, 2010

1st PKU-NII International Joint Workshop on Advanced Software Engineering

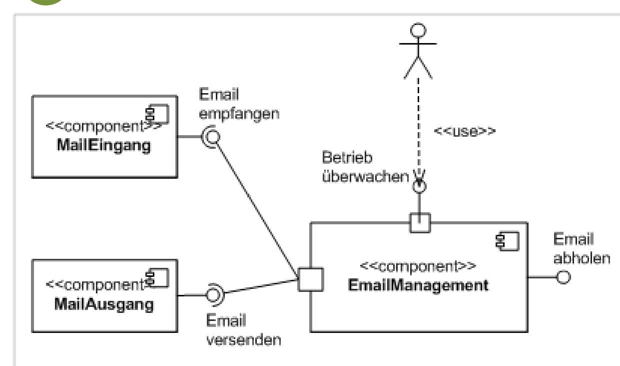
Goal of This Research

(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?

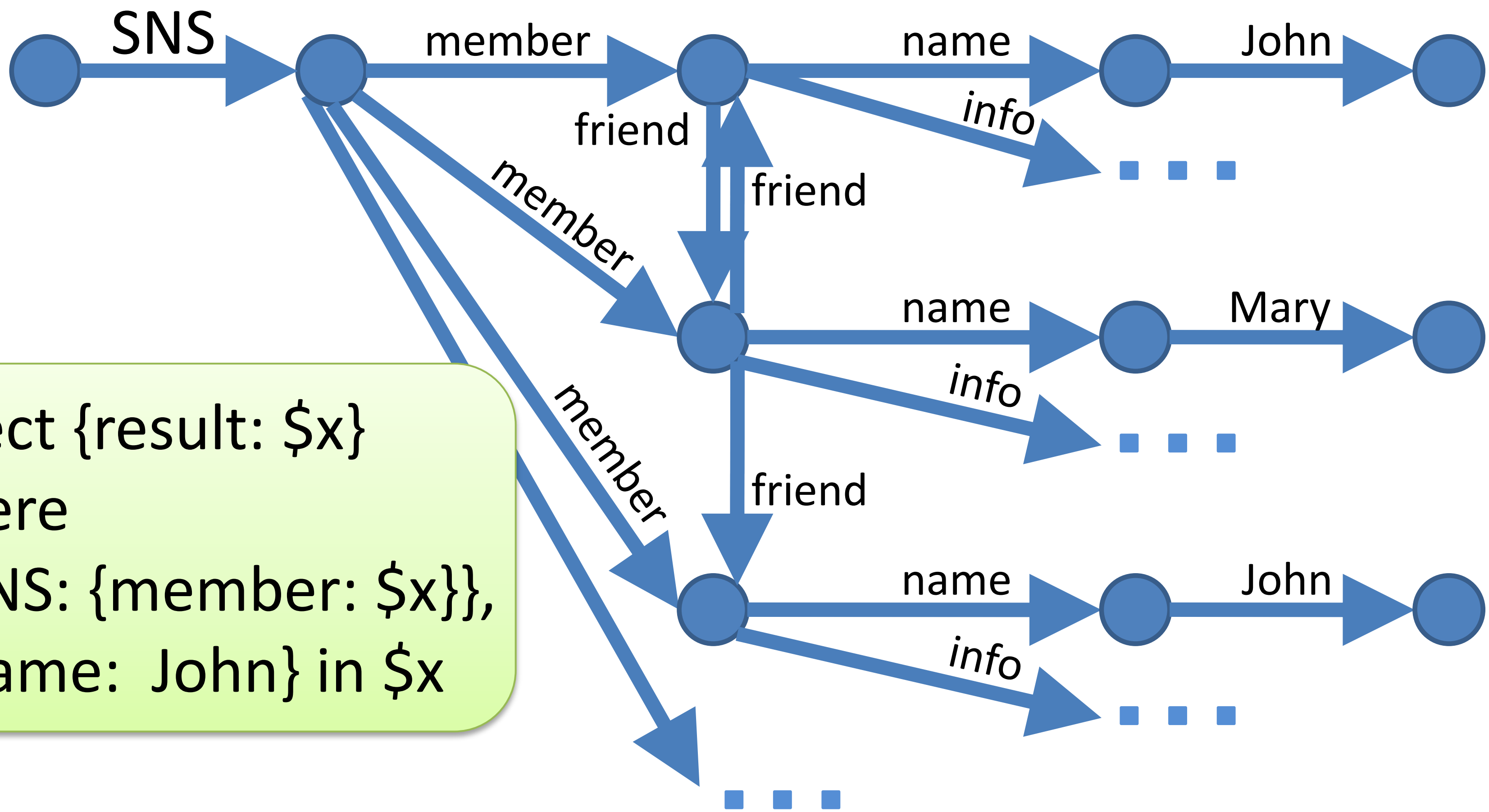


Today's Talk

- 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."**

Example : SNS-Members

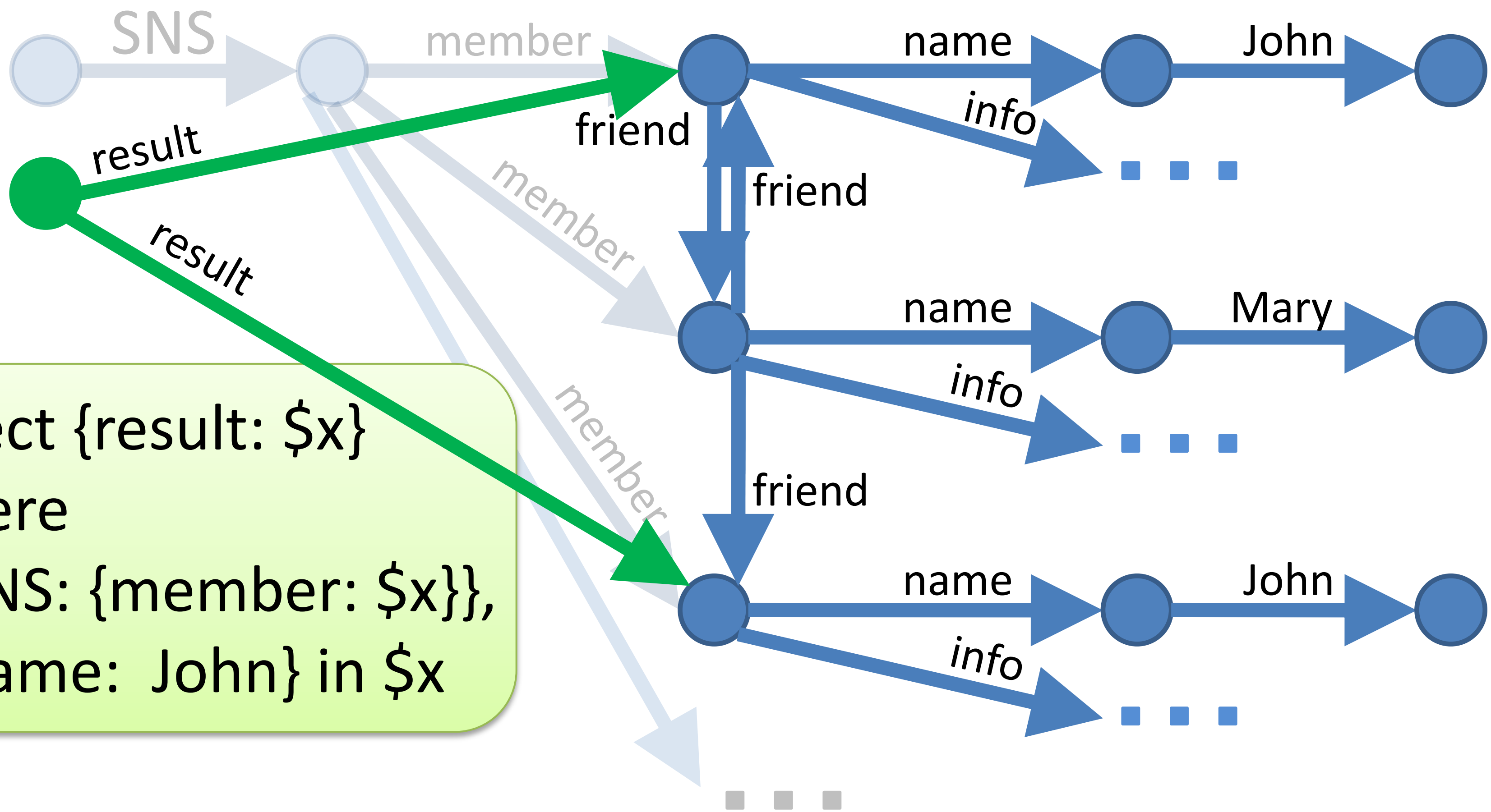
Extract all members using the screen-name “John”.




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

Example

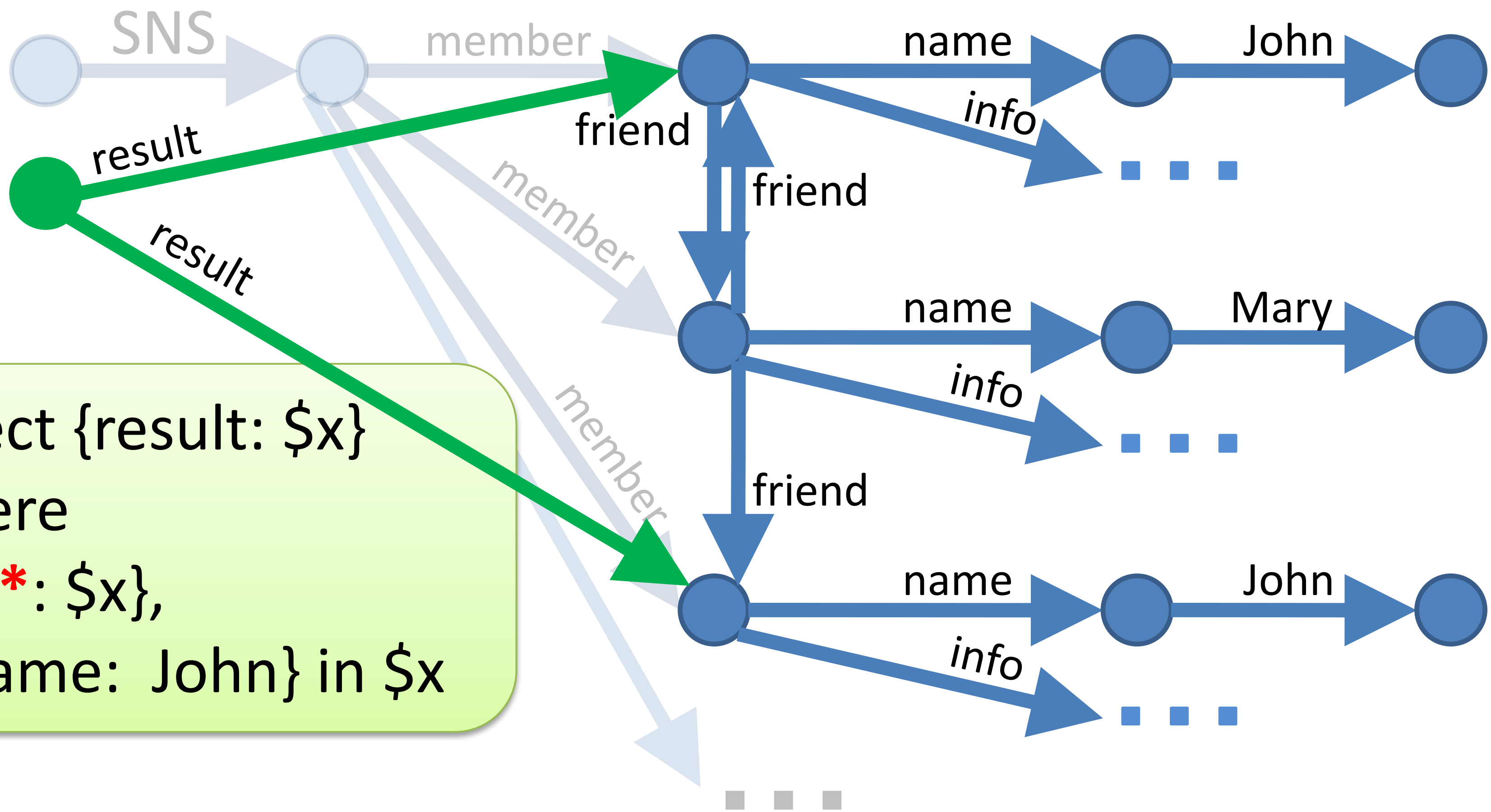
Extract all members using the screen-name “John”.




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

Example

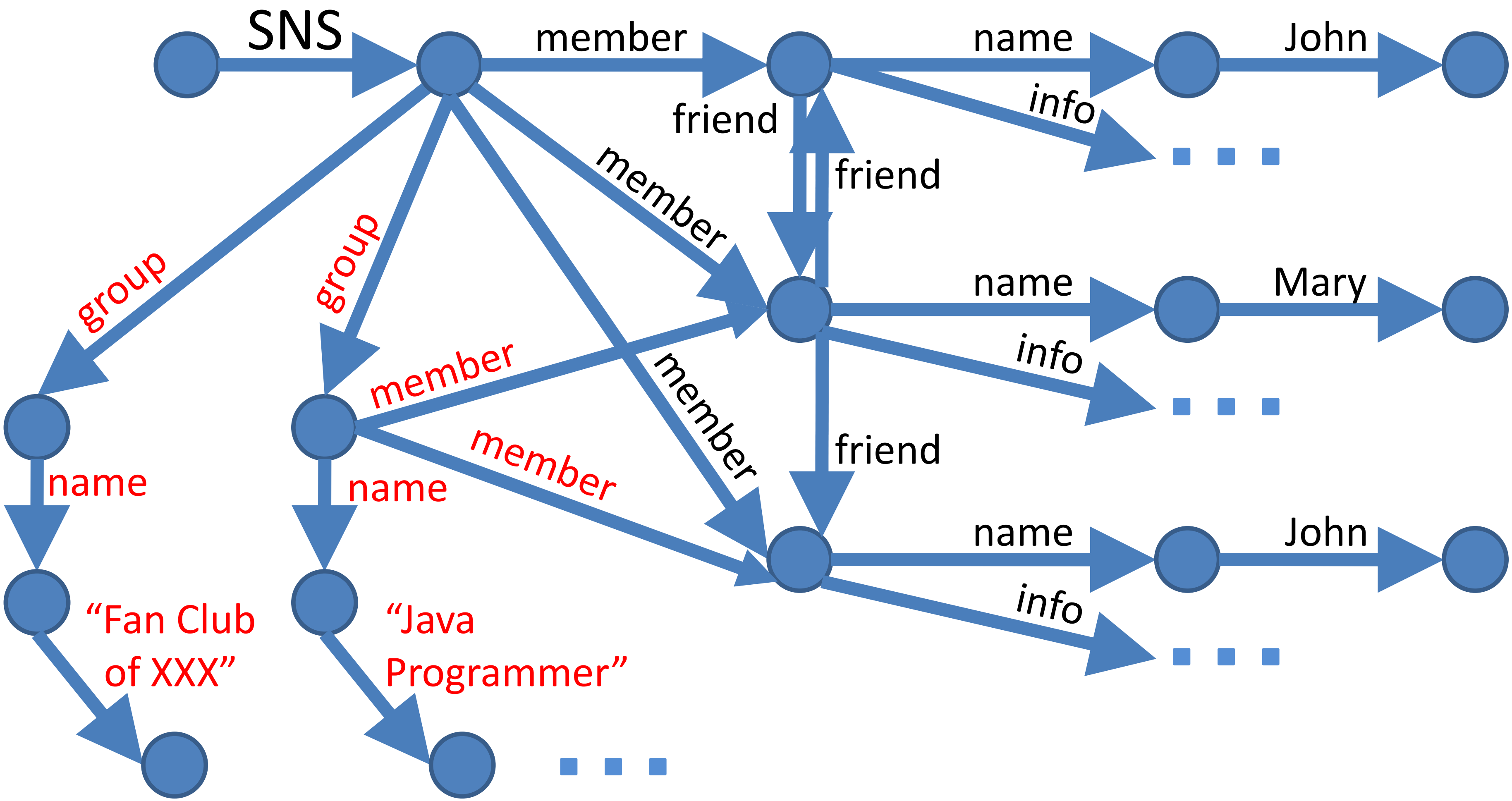
Lazy programmer may write …



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

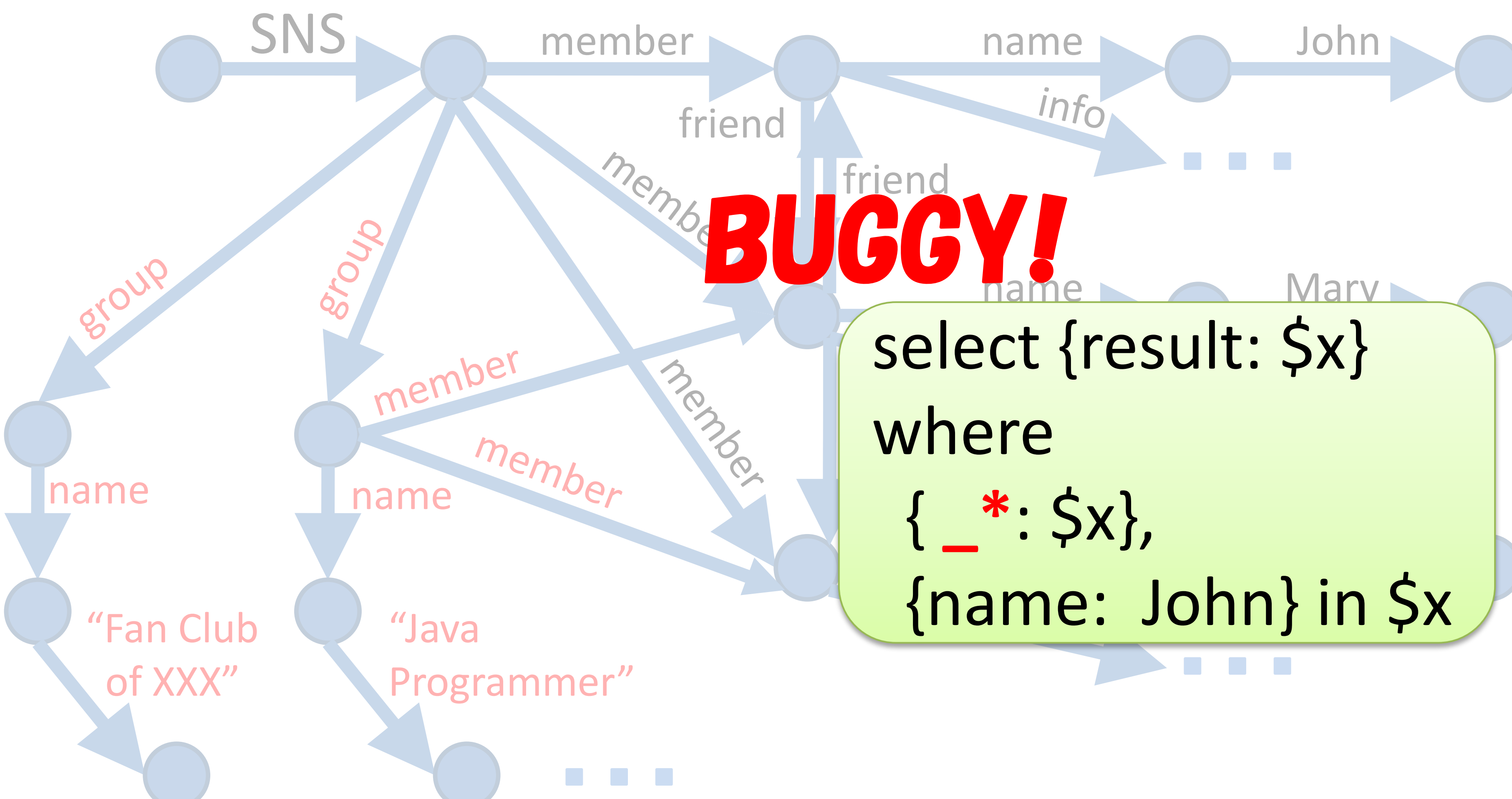
Example

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

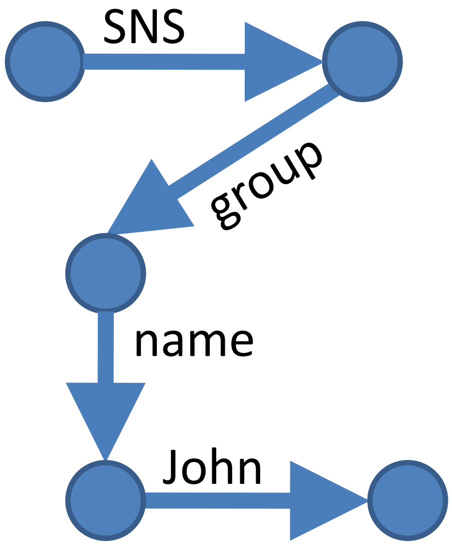
Then, our system automatically verify it!

```
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.

How?

By encoding transformations into a logic formula.

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

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

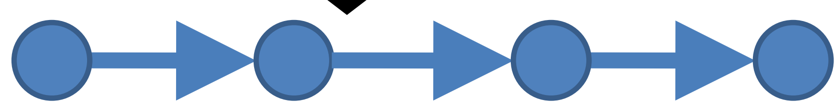
```
class OUTPUT {
 reference result*: MEM; }
```

formula

Check **VALIDNESS** using an existing verifier

"YES" / **"NO"+CE**

Decode CE to a graph



The Challenge

"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 2nd-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^{nd}-Order Logic

MSO is a usual 1^{st} 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) $x \in S$ $S \subseteq T$

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:

class OUTPUT { reference result*: MEM; }
class MEM { reference friend*: MEM;
reference name$^+$: STRING; }

Schema to MSO

- We do need MSO's expressiveness

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

$$
\begin{aligned}
&\exists^{set}\mathrm{OUTPUT}.\ \ \exists^{set}\mathrm{MEM}.\\
&\quad (\forall x.\ \mathrm{start}(x) \rightarrow x \in \mathrm{OUTPUT})\\
&\wedge (\forall x \in \mathrm{OUTPUT}.\ \forall e.\ \forall u.\\
&\qquad \mathrm{edge}(x,e,u) \rightarrow \mathrm{edge}_{\mathrm{result}}(x,e,u)\ \&\ u \in \mathrm{MEM})\\
&\wedge \ldots
\end{aligned}
$$

Transformation Language

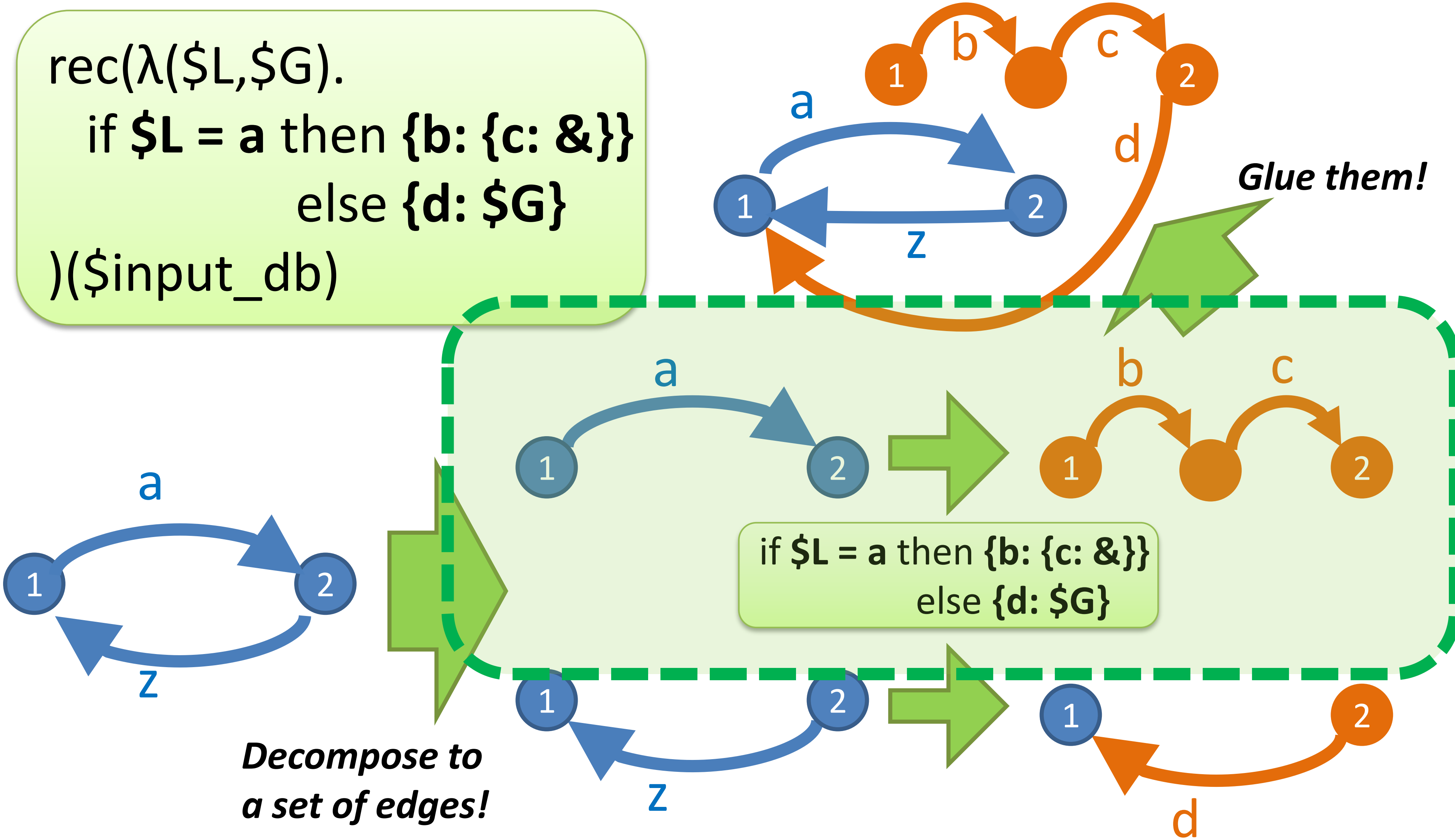
- UnCAL [Buneman et al, 2000]
  - Internal Representation of “UnQL” ➔

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

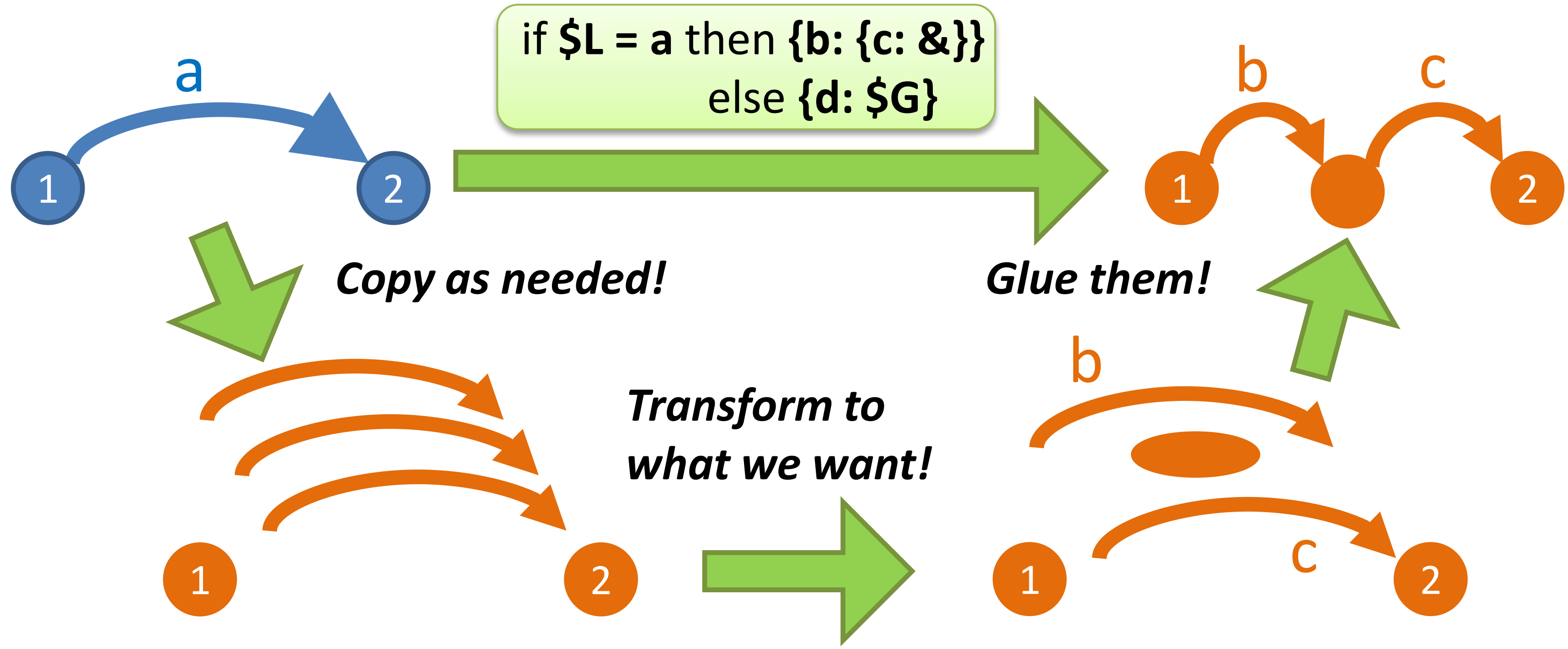
```
E ::= {L:E, L:E, …, L:E}
    | if L=L then E else E
    | $G
    | &
    | rec(λ($L,$G). E)(E)
L ::= (label constant)
     | $L
```

“Bulk” Semantics of UnCAL

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

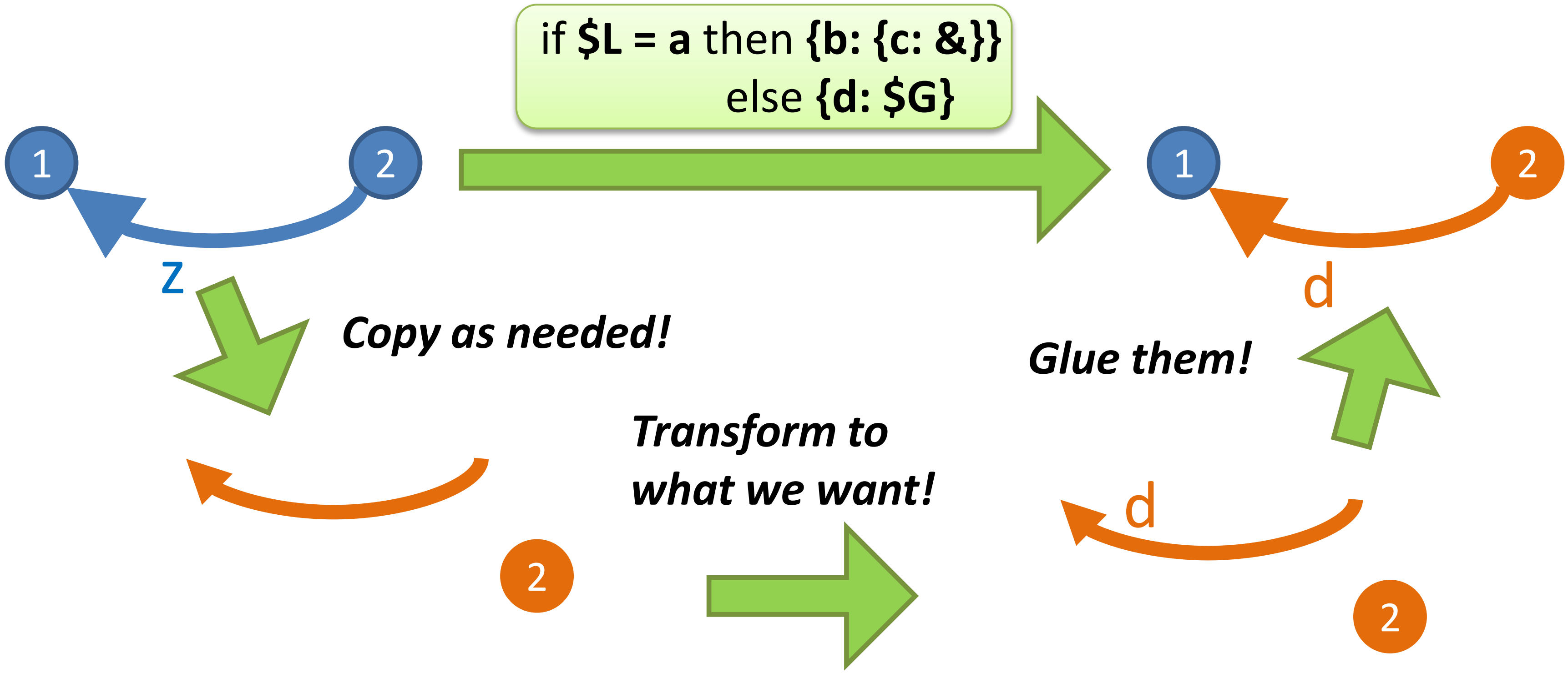


More Precise, MSO-Representable "Finite-Copy" Semantics



$$\mathrm{edge}_b(v,e,u) \Leftrightarrow$$
$$\quad \exists v'\ e'\ u'.\ \mathrm{edge}_a(v',e',u')\ \&\ v=v'_1\ \&\ e=e'_1\ \&\ u=e'_2$$
$$\mathrm{edge}_c(v,e,u) \Leftrightarrow$$
$$\quad \exists v'\ e'\ u'.\ \mathrm{edge}_a(v',e',u')\ \&\ v=e'_2\ \&\ e=e'_3\ \&\ e=u'_1$$

“Finite-Copy” Semantics



$$\mathrm{edge}_d(v,e,u) \Leftrightarrow$$
$$\exists v'\ e'\ u'.\ \neg\mathrm{edge}_a(v',e',u')\ \&\ v=v'_1\ \&\ e=e'_1\ \&\ u=u'$$

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)
```

$edge_b(v,e,u) \Leftrightarrow edge_a(v',e',u')\ \&\ v=v'_1\ \&\ e=e'_1\ \&\ u=e'_2$

$edge_c(v,e,u) \Leftrightarrow edge_a(v',e',u')\ \&\ v=e'_2\ \&\ e=e'_3\ \&\ e=u'_1$

$edge_d(v,e,u) \Leftrightarrow \neg edge_a(v',e',u')\ \&\ v=v'_1\ \&\ e=e'_1\ \&\ u=u'$

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

"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: STRI

$\exists^{set}$OUTPUT. $\exists^{set}$MEM.
(∀x. start(x) → x∈OUTPUT)
∧ (∀x∈OUTPUT. ∀e. ∀u.
edge(x,e,u) → $\text{edge}_{\text{result}}$(x,e,u) & u∈MEM) ∧ ...

OUTPUT GRAPH description
by the INPUT GRAPH

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

$\text{edge}_b(v,e,u) \Leftrightarrow \text{edge}_a(v',e',u')\ \&\ v=v'_1\ \&\ e=e'_1\ \&\ u=e'_2$
$\text{edge}_c(v,e,u) \Leftrightarrow \text{edge}_a(v',e',u')\ \&\ v=e'_2\ \&\ e=e'_3\ \&\ e=u'_1$
$\text{edge}_d(v,e,u) \Leftrightarrow \neg\text{edge}_a(v',e',u')\ \&\ v=v'_1\ \&\ e=e'_1\ \&\ u=u'$

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

Verify this is valid for any INPUT GRAPHS!!

Note: Harder Case

- Nested Recursion (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}} ...
```

(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

$\exists^{set}$ Q_1. $\exists^{set}$ Q_2. … $\exists^{set}$ Q_n. "there is a run of the automaton that reaches states Q's on each node"

MSO Validation

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

?

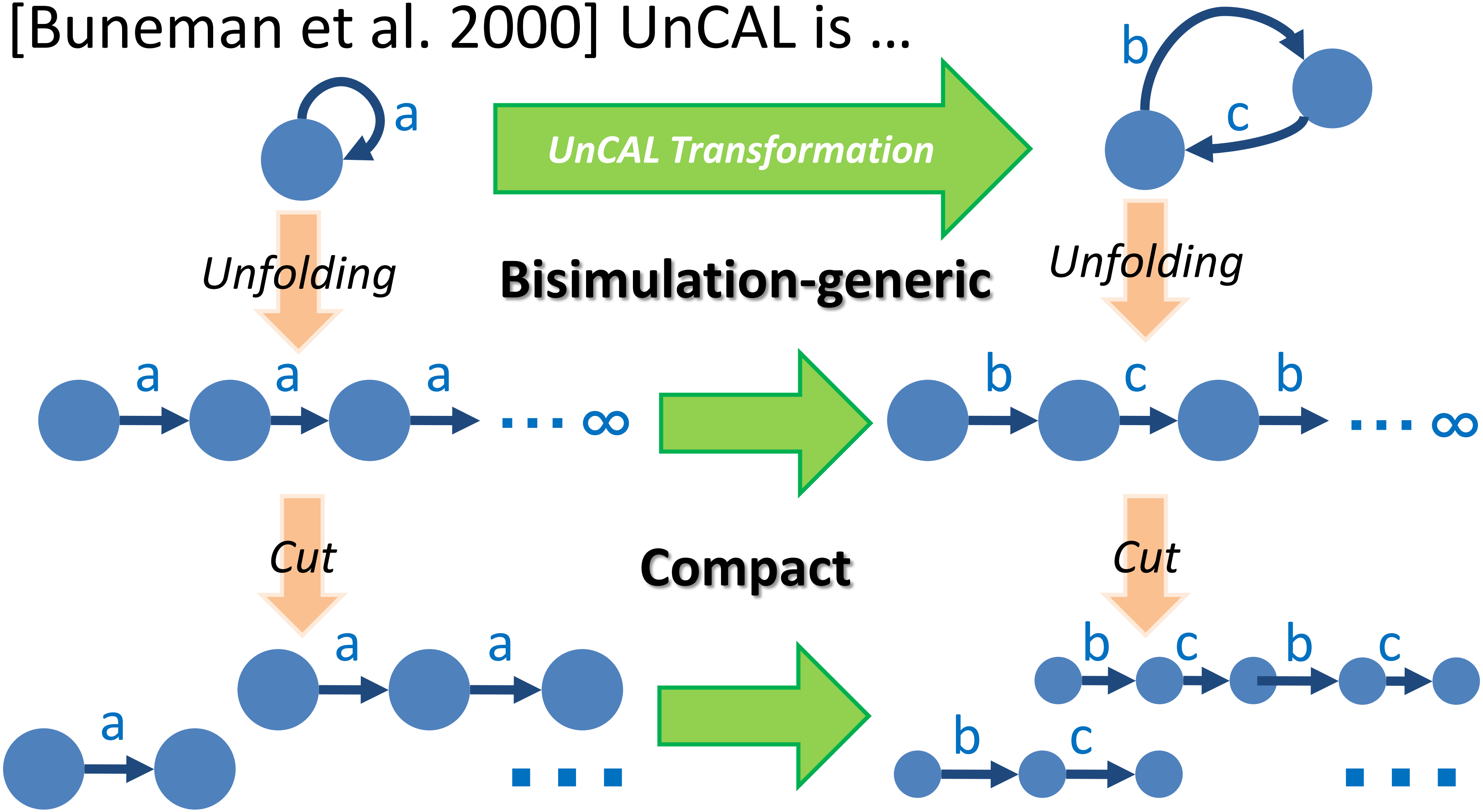
MSO (even 1st-Order Logic) on Graphs is undecidable [Trakhtenbrot 1950].

MONA [Henriksen et al., 1995]

can decide validness of **MSO on Finite Trees**.

Two Nice Props of UnCAL

[Buneman et al. 2000] UnCAL is …



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 [Henriksen et al., 1995]

can decide validness of **MSO on Finite Trees**.

Discussion: Choice of Logic

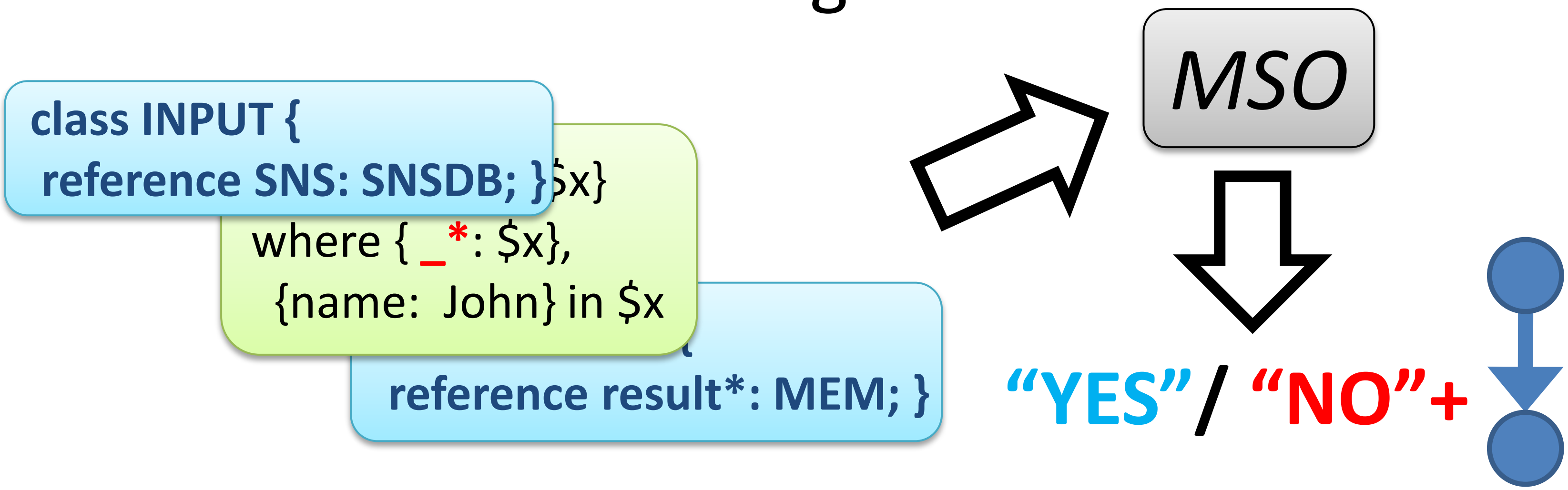
- **MSO**
  - **Powerful, yet decidable (if we fully utilize bisimulation)**
- FO+TC (1st-Order Logic + Transitive Closure)
  - Very powerful; express all UnCAL without annotation
  - Undecidable, even on finite trees
- FO (1st-Order), SMT (Satisfiability Modulo Theory)
  - Very good solvers
  - Too 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…

Summary

Verify "type-correctness" of graph transformations via Monadic 2nd-Order Logic



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