Verification of Pointer Programs : a Tool Comparison

Victor Lanvin
Supervisors : C. Jansen, J.-P. Katoen, C. Matheja, T. Noll

September 2, 2015

Outline

Example Problem

- Try to prove that BubbleSort(List L) effectively sorts L

Example Problem

- Try to prove that BubbleSort(List L) effectively sorts L

  Naively, you check this for ANY list, of ANY length

  $\Rightarrow$ State space explosion (or infinite)

Example Problem

- Try to prove that BubbleSort(List L) effectively sorts L

  Naively, you check this for ANY list, of ANY length

  $\Rightarrow$ State space explosion (or infinite)

  $\Rightarrow$ Abstraction is necessary

Example Problem

- Try to prove that BubbleSort(List L) effectively sorts L

  Naively, you check this for ANY list, of ANY length

  $\Rightarrow$ State space explosion (or infinite)

  $\Rightarrow$ Abstraction is necessary

- A lot of different approaches have been developed

Objectives

- Compare four approaches to the verification of pointer programs
- Find good criteria of comparison
- Find and implement algorithms on each tool
- Determine the strengths and weaknesses of each approach

Outline

TVLA (Three-Valued Logic Analysis) 1/3

- Input :

  Algorithm defined as locations and actions

  Actions, core and instrumentation predicates in three-valued logic

- Output :

  Generated state space

  Structures on which the properties failed

TVLA (Three-Valued Logic Analysis) 2/3

$\wedge$	0	0.5	1
0	0	0	0
0.5	0	0.5	0.5
1	0	0.5	1

$\vee$	0	0.5	1
0	0	0.5	1
0.5	0.5	0.5	1
1	1	1	1

A	$\neg$ A
0	1
0.5	0.5
1	0

Figure: Kleene's semantics for operators $\wedge$,$\vee$ and $\neg$.

TVLA (Three-Valued Logic Analysis) 2/3

$\wedge$	0	0.5	1
0	0	0	0
0.5	0	0.5	0.5
1	0	0.5	1

$\vee$	0	0.5	1
0	0	0.5	1
0.5	0.5	0.5	1
1	1	1	1

A	$\neg$ A
0	1
0.5	0.5
1	0

Figure: Kleene's semantics for operators $\wedge$,$\vee$ and $\neg$.

$\exists$ and $\forall$ have the same meaning as in FO

atomic propositions are user defined predicates

TVLA (Three-Valued Logic Analysis) 3/3

Let $n(2)$ be a core predicate corresponding to the next field
The action $lhs = rhs \rightarrow n$ is defined as :

TVLA (Three-Valued Logic Analysis) 3/3

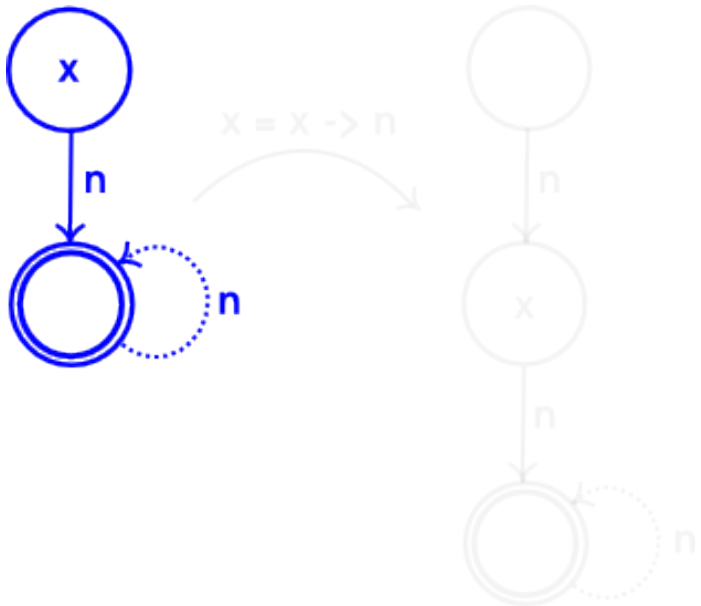
Let $n(2)$ be a core predicate corresponding to the next field
The action $lhs = rhs \rightarrow n$ is defined as :

```
%action Set_To_Next(lhs,rhs) {
```
$$lhs(v) = \exists(v_1) rhs(v_1) \wedge n(v_1, v)$$
```
}
```

TVLA (Three-Valued Logic Analysis) 3/3

Let $n(2)$ be a core predicate corresponding to the next field
The action $lhs = rhs \rightarrow n$ is defined as :

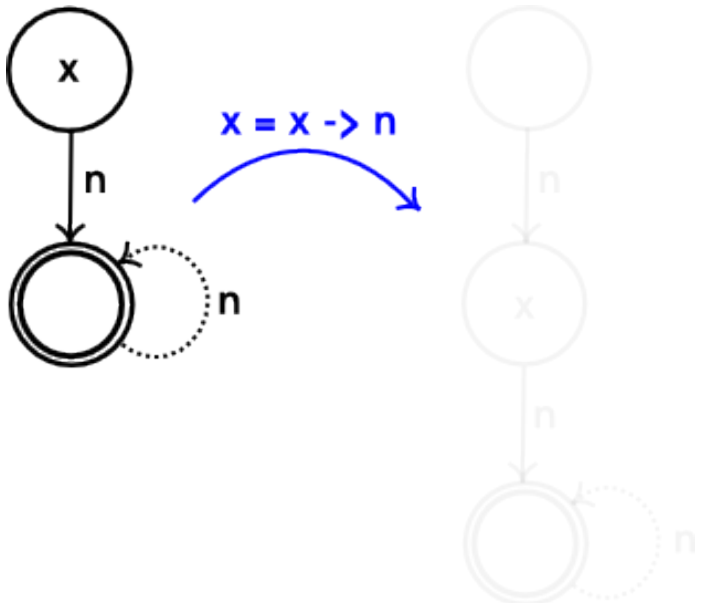
```
%action Set_To_Next(lhs,rhs) {
```
$$lhs(v) = \exists(v_1) rhs(v_1) \wedge n(v_1, v)$$
```
}
```



TVLA (Three-Valued Logic Analysis) 3/3

Let $n(2)$ be a core predicate corresponding to the next field
The action $lhs = rhs \rightarrow n$ is defined as :

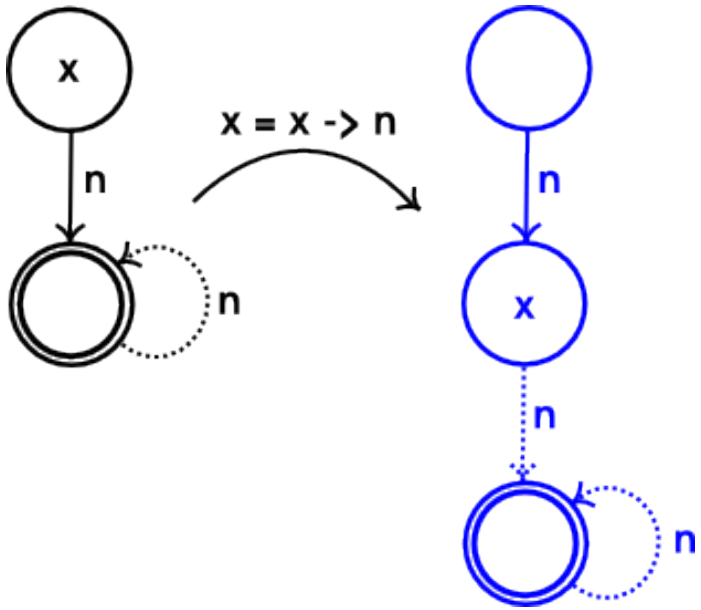
```
%action Set_To_Next(lhs,rhs) {
```
$$lhs(v) = \exists(v_1) rhs(v_1) \wedge n(v_1, v)$$
```
}
```



TVLA (Three-Valued Logic Analysis) 3/3

Let $n(2)$ be a core predicate corresponding to the next field
The action $lhs = rhs \rightarrow n$ is defined as :

```
%action Set_To_Next(lhs,rhs) {
```
$$lhs(v) = \exists(v_1) rhs(v_1) \wedge n(v_1, v)$$
```
}
```



Groove (Generalized Graph Transformations) 1/2

- Input :

  Algorithm as a "control program" using actions

  Variables and pointers as graphs

  Actions as graph transformations

- Output :

  Result of the LTL and CTL checks

  Generated state space

Groove (Generalized Graph Transformations) 1/2

- Input :

  Algorithm as a "control program" using actions

  Variables and pointers as graphs

  Actions as graph transformations

- Output :

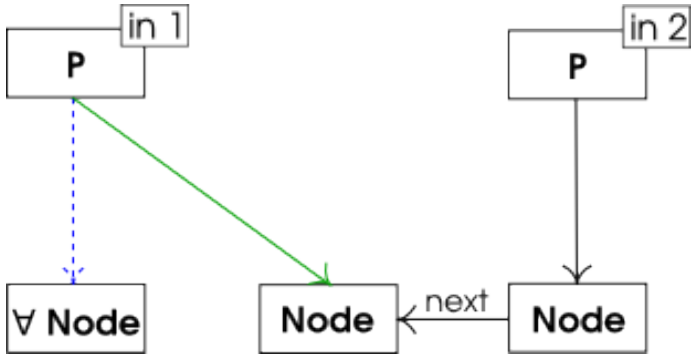
  Result of the LTL and CTL checks

  Generated state space

Note : no automatic abstraction !

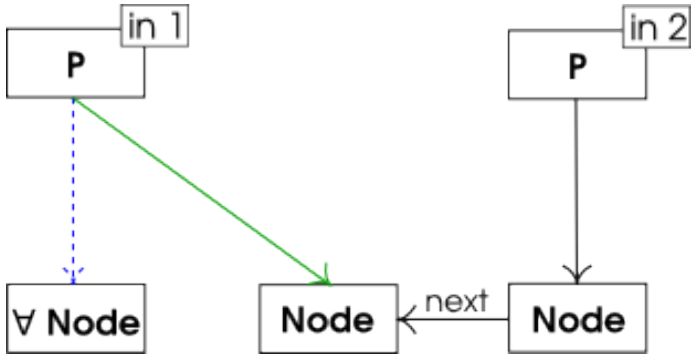
Groove (Generalized Graph Transformations) 2/2

Rule $Set\_To\_Next(lhs, rhs)$:

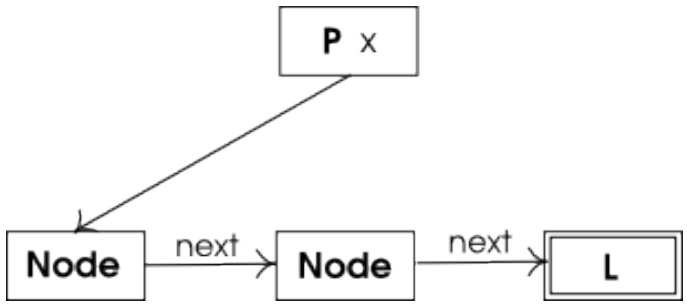


Groove (Generalized Graph Transformations) 2/2

Rule $Set\_To\_Next(lhs, rhs)$:

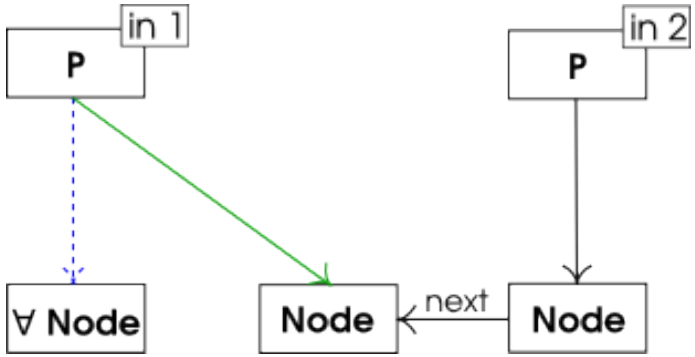


Start graph :

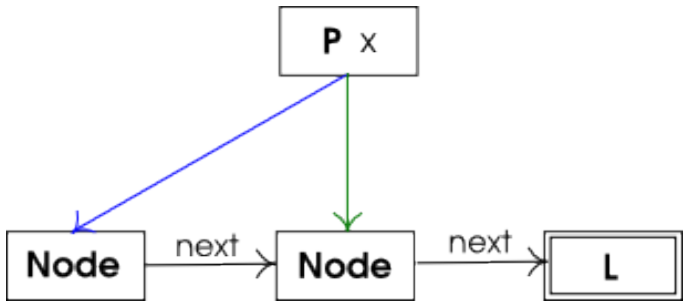


Groove (Generalized Graph Transformations) 2/2

Rule $Set\_To\_Next(lhs, rhs)$:



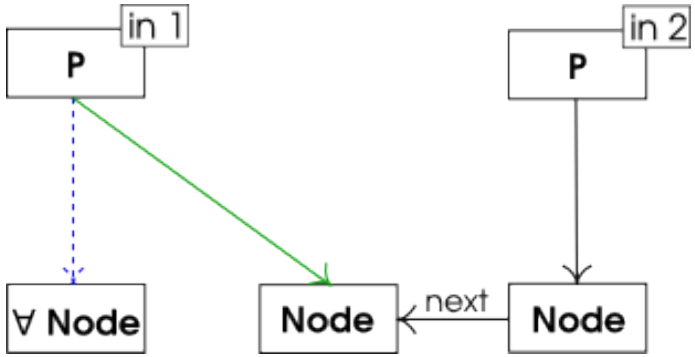
Start graph :



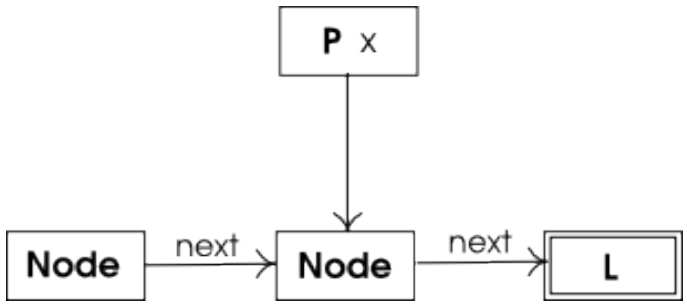
$Set\_To\_Next(x, x)$

Groove (Generalized Graph Transformations) 2/2

Rule $Set\_To\_Next(lhs, rhs)$:



Start graph :



$Set\_To\_Next(x, x)$

Juggrnaut (Hyperedge Replacement Grammars) 1/3

- Input :

  Java program

  Hyperedge replacement grammar

  Starting heap configuration as an hypergraph

- Output :

  Generated state space

  Final heap configurations

Juggrnaut (Hyperedge Replacement Grammars) 1/3

- Input :

  Java program

  Hyperedge replacement grammar

  Starting heap configuration as an hypergraph

- Output :

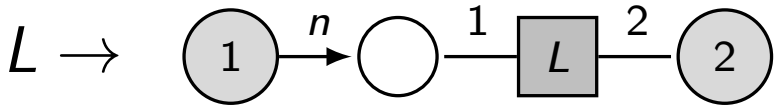
  Generated state space

  Final heap configurations

Note : rules are applied in both ways (concretization/abstraction)

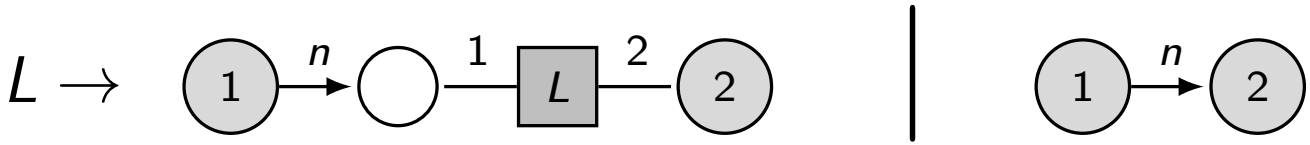
Juggrnaut (Hyperedge Replacement Grammars) 2/3

Example : singly-linked list



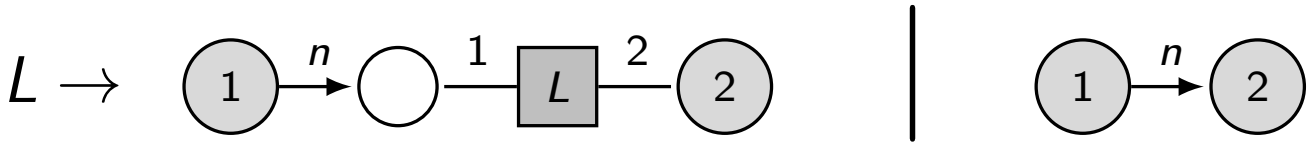
Juggrnaut (Hyperedge Replacement Grammars) 2/3

Example : singly-linked list

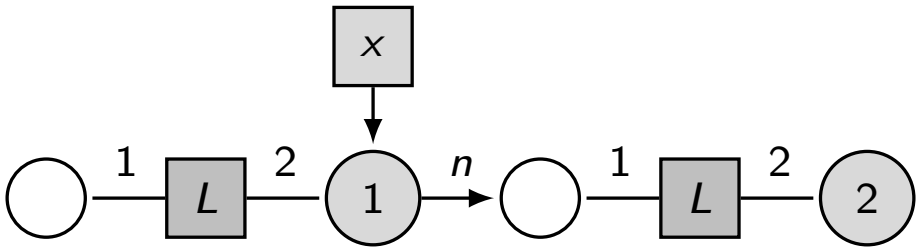


Juggrnaut (Hyperedge Replacement Grammars) 2/3

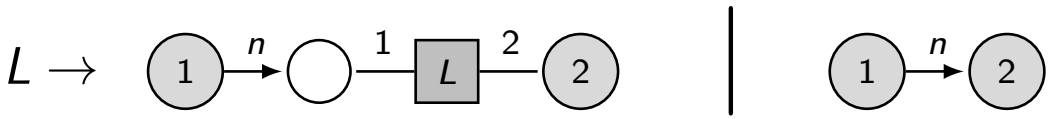
Example : singly-linked list



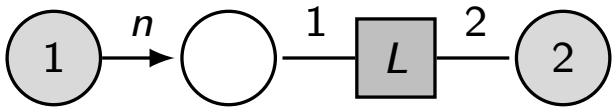
Heap configuration :



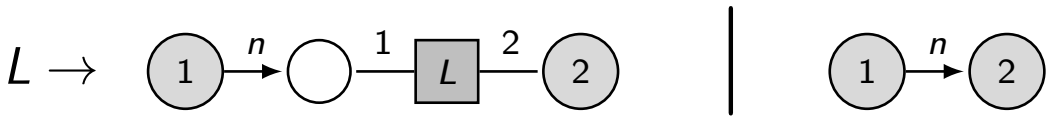
Juggrnaut (Hyperedge Replacement Grammars) 3/3



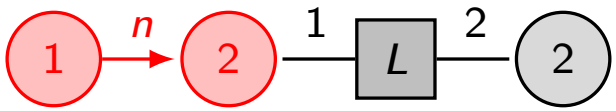
The two rules form a critical pair :



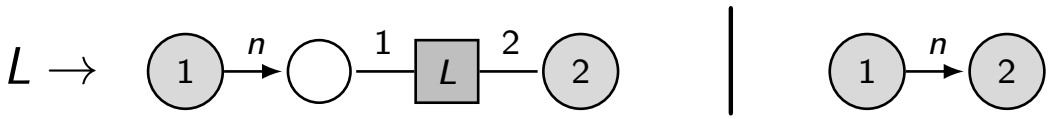
Juggrnaut (Hyperedge Replacement Grammars) 3/3



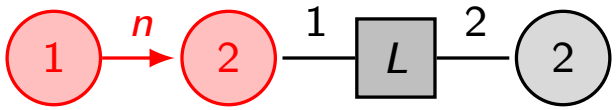
The two rules form a critical pair :



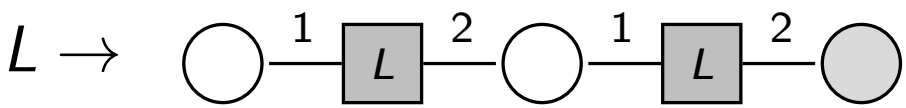
Juggrnaut (Hyperedge Replacement Grammars) 3/3



The two rules form a critical pair :



We need to add this third one :



jStar (Separation Logic) 1/3

- Input :

  Java program, annotated with pre/post conditions

  Predicates and actions as inference rules in separation logic

- Output :

  A proof of the algorithm

jStar (Separation Logic) 2/3

Separation logic mainly adds 3 symbols to boolean logic : $*, \mapsto,$ **emp**.

- $*$ is a separating conjunction
- $\mapsto$ can be seen as a pointer property
- **emp** holds when the heap is empty

jStar (Separation Logic) 2/3

Separation logic mainly adds 3 symbols to boolean logic : $*, \mapsto,$ **emp**.

- $*$ is a separating conjunction
- $\mapsto$ can be seen as a pointer property
- **emp** holds when the heap is empty

A sequent is defined by the following structure :

$$\frac{\Gamma_s * F_{sub} | Prem_L \vdash Prem_R}{\Gamma_s | Conc_L \vdash Conc_R}$$

jStar (Separation Logic) 3/3

- We can define a node x by a predicate $node(x, y)$ such that :

$$node(x, y) \iff field(x, next, y)(\iff x \to next = y)$$

jStar (Separation Logic) 3/3

- We can define a node x by a predicate $node(x, y)$ such that :

$$node(x, y) \iff field(x, next, y)(\iff x \rightarrow next = y)$$

- An abstract list can be defined by a predicate $ls(x, y)$ where x is the head and y is the tail :

$$\frac{node(x,y) \mid\!\vdash ls(y,z)}{\mid node(x,y) \vdash ls(x,z)} \text{ (concretization)} \qquad \frac{\mid\!\vdash ls(x,z)}{\mid\!\vdash node(x,y) * ls(y,z)} \text{ (abstraction)} \qquad \frac{\mid\!\vdash ls(x,z)}{\mid\!\vdash ls(x,y) * ls(y,z)} \text{ (merge)}$$

Outline

List reversal

```
List reverseList(List l)
{
  List y,t = null;
  List x = l;
  while(x != null) {
    t = y;
    y = x;
    x = x.n
    y.n = t;
  }
  return y;
}
```

- Local algorithm
- Keeps the list structure at every iteration
- Only one loop
- No data fields

Bubble sort

```
void BubbleSort(List l)
{
  List y,yn,p,t = null;
  bool change = true;
  while(change) {
    change = false;
    p = null;
    y = list;
    yn = y.next;

    while(yn != null) {
      if(y.data > yn.data) {
        change = true;
        swap(y,yn);
      }
      else {
        move_forward(p,y,yn);
      }
    }
  }
  return y;
}
```

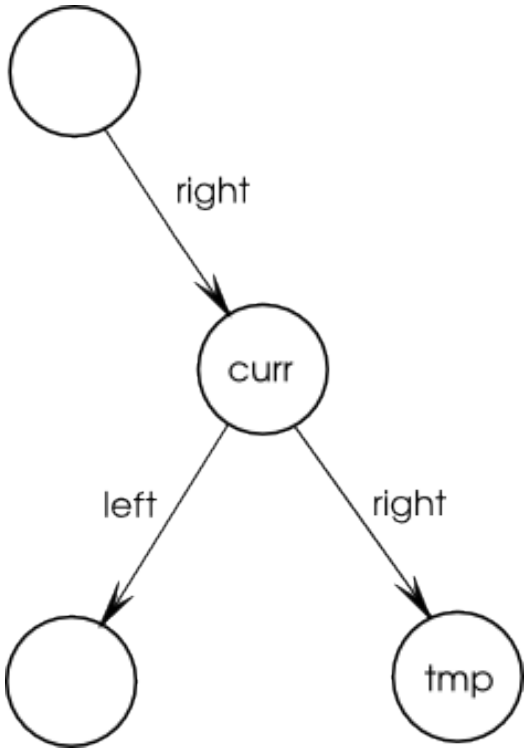
- Less local than list reversal
- Keeps a list structure at every iteration, simple invariant
- Two nested loops
- Data storage and comparison

Lindstrom tree traversal

```
void Lindstrom(Tree t)
{
  //create_pointers
  curr = t;
  [...]

  while(curr != sen) {
    next = curr.left;
    curr.left = curr.right;
    curr.right = prev;

    //Move forward
    prec = cur;
    cur = next

    [...]
  }
}
```

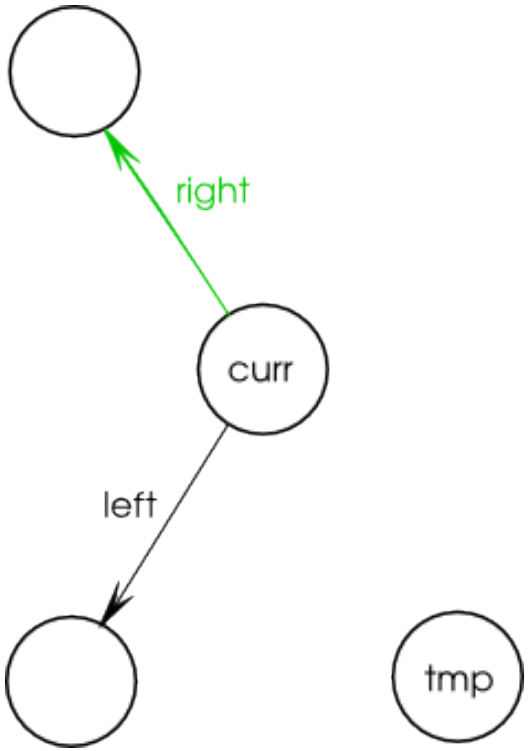
- Local tree traversal
- The tree structure is "mostly" kept
- Constant space
- No data manipulation, no conditionnal branching

DSW (Deutsch-Schorr-Waite) variant



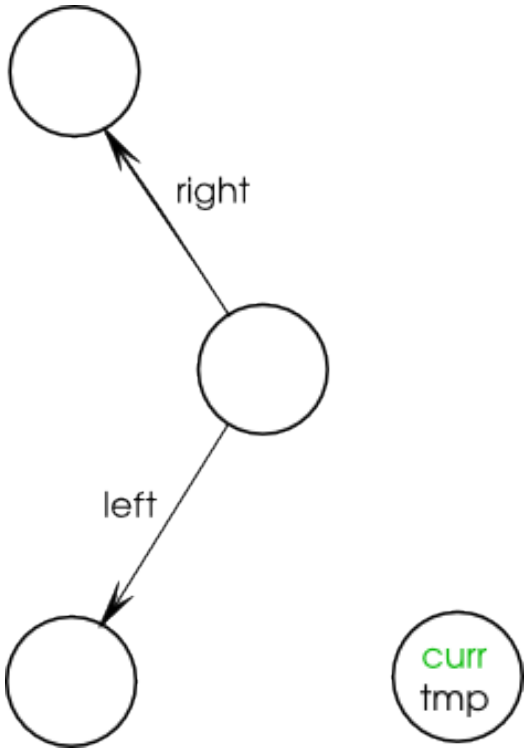
- Non local variant of Lindstrom
- Tree structure is not preserved
- Three possible cases for every node
- No data manipulation

DSW (Deutsch-Schorr-Waite) variant



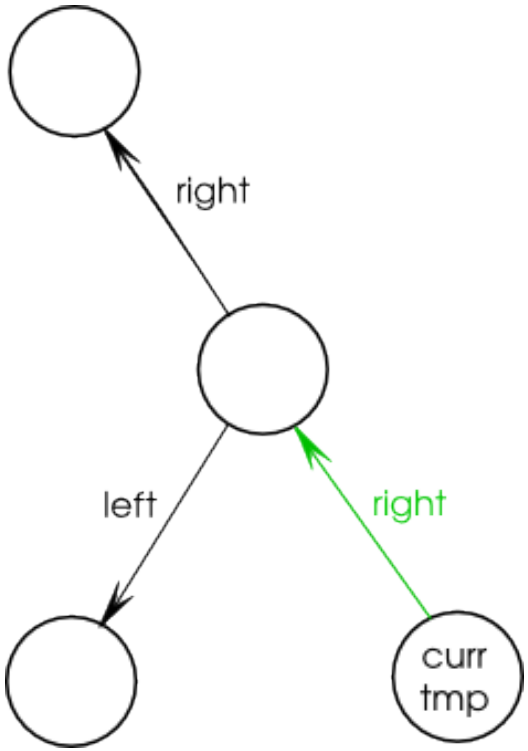
- Non local variant of Lindstrom
- Tree structure is not preserved
- Three possible cases for every node
- No data manipulation

DSW (Deutsch-Schorr-Waite) variant



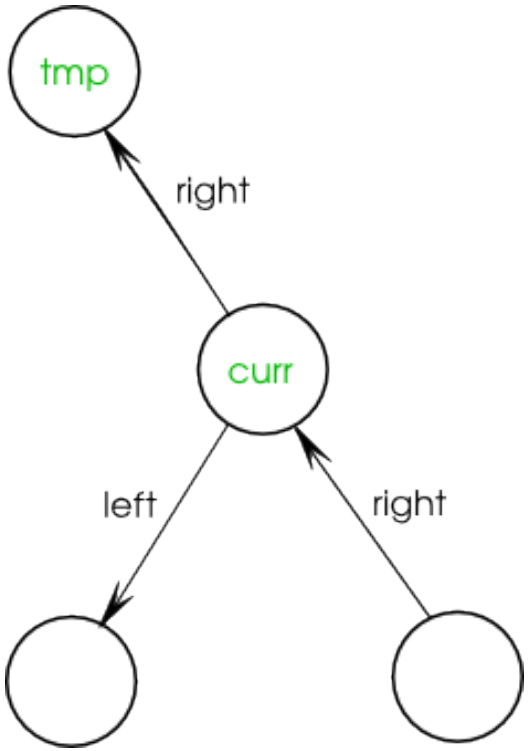
- Non local variant of Lindstrom
- Tree structure is not preserved
- Three possible cases for every node
- No data manipulation

DSW (Deutsch-Schorr-Waite) variant



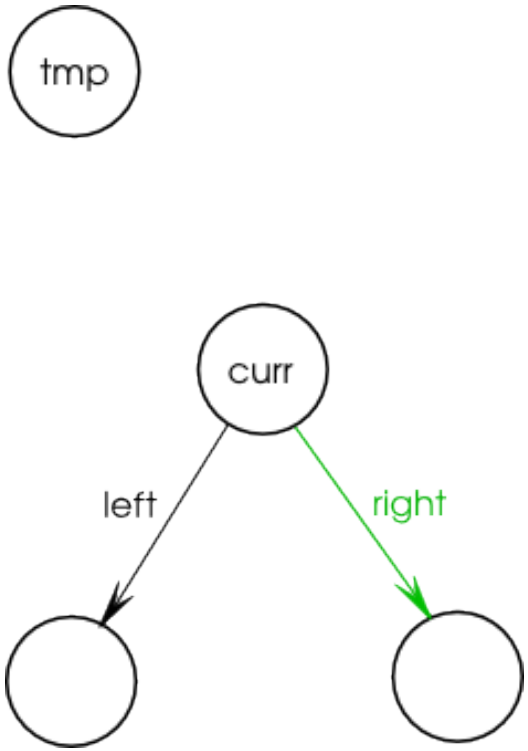
- Non local variant of Lindstrom
- Tree structure is not preserved
- Three possible cases for every node
- No data manipulation

DSW (Deutsch-Schorr-Waite) variant



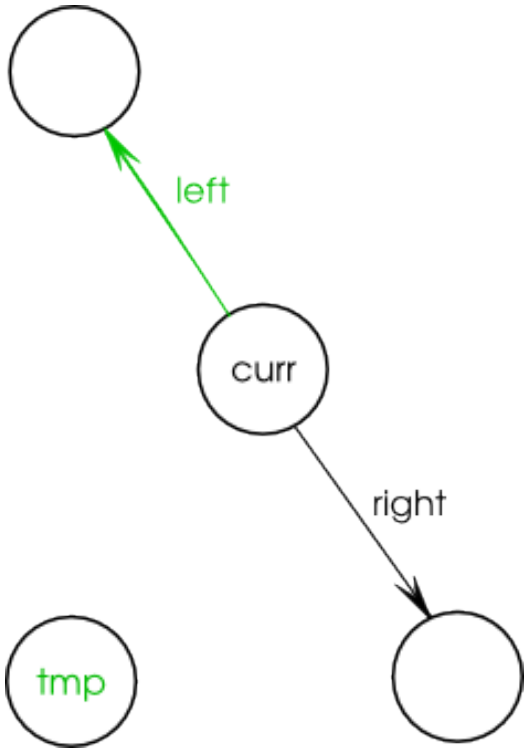
- Non local variant of Lindstrom
- Tree structure is not preserved
- Three possible cases for every node
- No data manipulation

DSW (Deutsch-Schorr-Waite) variant



- Non local variant of Lindstrom
- Tree structure is not preserved
- Three possible cases for every node
- No data manipulation

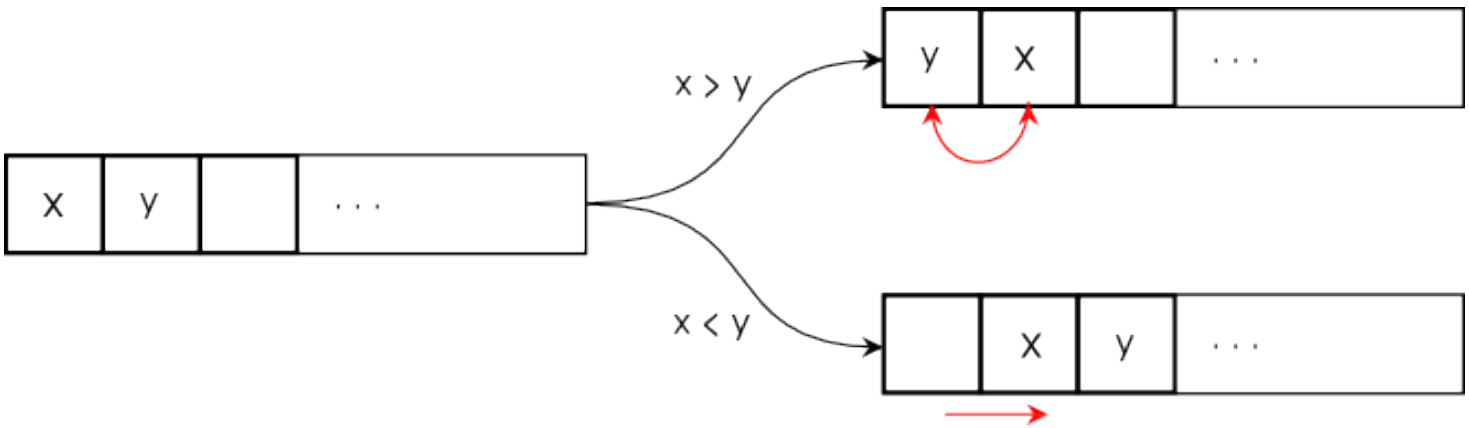
DSW (Deutsch-Schorr-Waite) variant



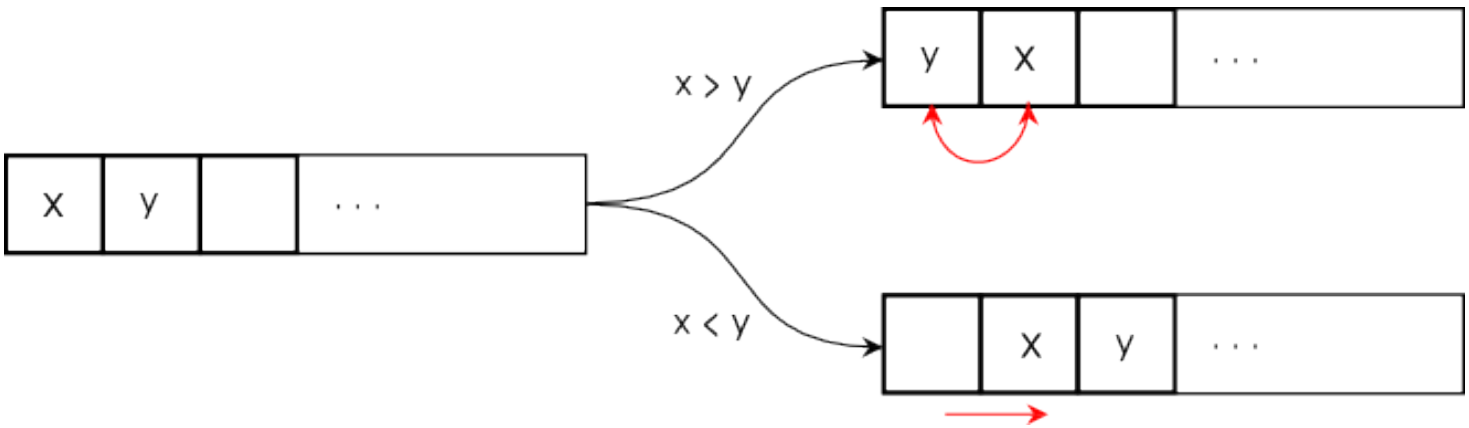
- Non local variant of Lindstrom
- Tree structure is not preserved
- Three possible cases for every node
- No data manipulation

Outline

General idea (1/2)



General idea (1/2)



Constant invariant :

External loop invariant :

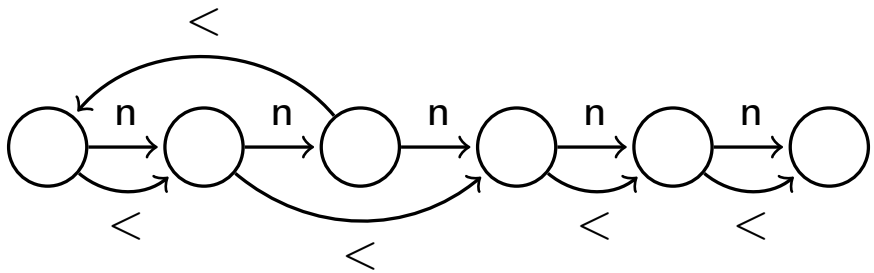
General idea (2/2)

- We need to be able to compare pair of elements
- We want to define a total ordering on the elements, or a permutation
  Note : those two approaches are equivalent
- We need to be able to abstract a list and a sorted list

General idea (2/2)

- We need to be able to compare pair of elements
- We want to define a total ordering on the elements, or a permutation
  Note : those two approaches are equivalent
- We need to be able to abstract a list and a sorted list

Example :



Implementation (TVLA)

The implementation in TVLA is straightforward :

- We force the $<$ predicate to be transitive and antisymmetric
- The $<$ predicate is initialized with 0.5 so that both cases are tested
- The transitivity guarantees the abstraction of the sorted list

Implementation (Groove)

The structures are similar to those used in TVLA :

- We use the $\forall$ quantifier to apply the ordering rule to every pair
- We use a restriction rule to avoid creating cycles in the ordering
- If an element is "lower than" and adjacent to a sorted list, then it is abstracted

Implementation (Juggrnaut) (1/2)

Theorem (bounded treewidth of HRLs)

Let $\mathcal{L}$ be a HRL. There exists an integer N such that, forall $G \in \mathcal{L}$, $\mathit{treewidth}(G) \leq N$

Implementation (Juggrnaut) (1/2)

Theorem (bounded treewidth of HRLs)

Let $\mathcal{L}$ be a HRL. There exists an integer N such that, forall $G \in \mathcal{L}$, $\mathit{treewidth}(G) \leq N$

- A total ordering on a list of length n is an n-clique, and therefore has a treewidth of $n - 1$.

Implementation (Juggrnaut) (1/2)

Theorem (bounded treewidth of HRLs)

Let $\mathcal{L}$ be a HRL. There exists an integer N such that, forall $G \in \mathcal{L}$, $\mathit{treewidth}(G) \leq N$

- A total ordering on a list of length n is an n-clique, and therefore has a treewidth of $n - 1$.
- A permutation is also out of scope

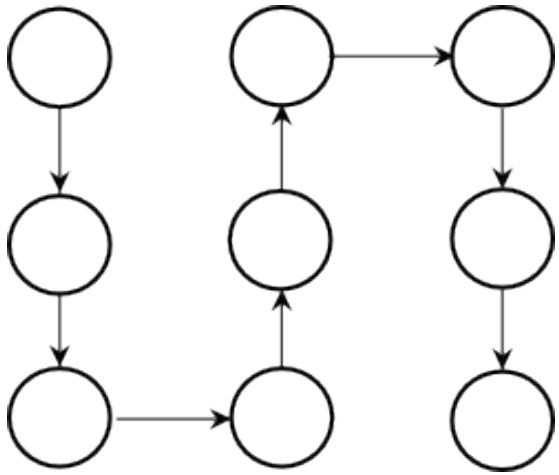
Implementation (Juggrnaut) (1/2)

Theorem (bounded treewidth of HRLs)

Let $\mathcal{L}$ be a HRL. There exists an integer N such that, forall $G \in \mathcal{L}$, $\mathit{treewidth}(G) \leq N$

- A total ordering on a list of length n is an n-clique, and therefore has a treewidth of $n - 1$.

- A permutation is also out of scope



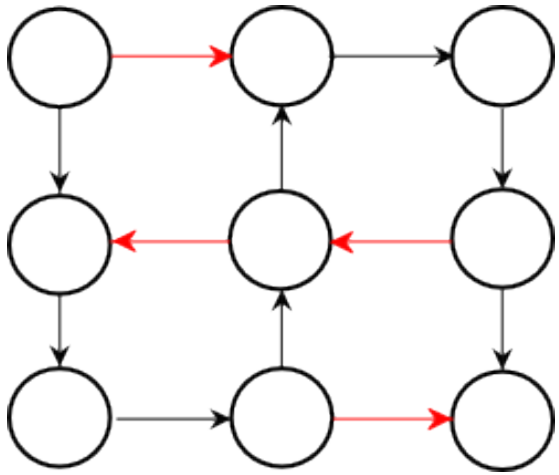
Implementation (Juggrnaut) (1/2)

Theorem (bounded treewidth of HRLs)

Let $\mathcal{L}$ be a HRL. There exists an integer N such that, forall $G \in \mathcal{L}$, $\mathit{treewidth}(G) \leq N$

- A total ordering on a list of length n is an n-clique, and therefore has a treewidth of $n - 1$.

- A permutation is also out of scope



Implementation (Juggrnaut) (2/2)

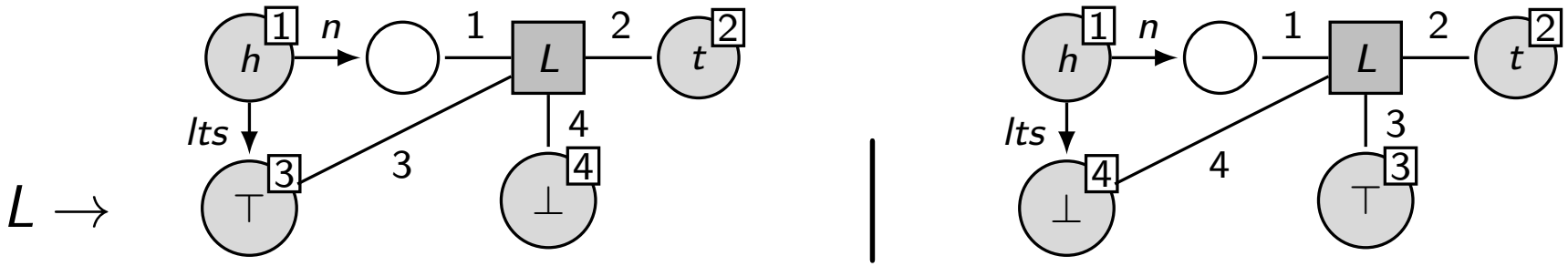
Note : We only need to compare adjacent elements !

- We add a flag to every node, comparing it to its successor
- We create 3 non-terminals : list(4), sorted list(3), and reverse sorted list(3)

Implementation (Juggrnaut) (2/2)

Note : We only need to compare adjacent elements !

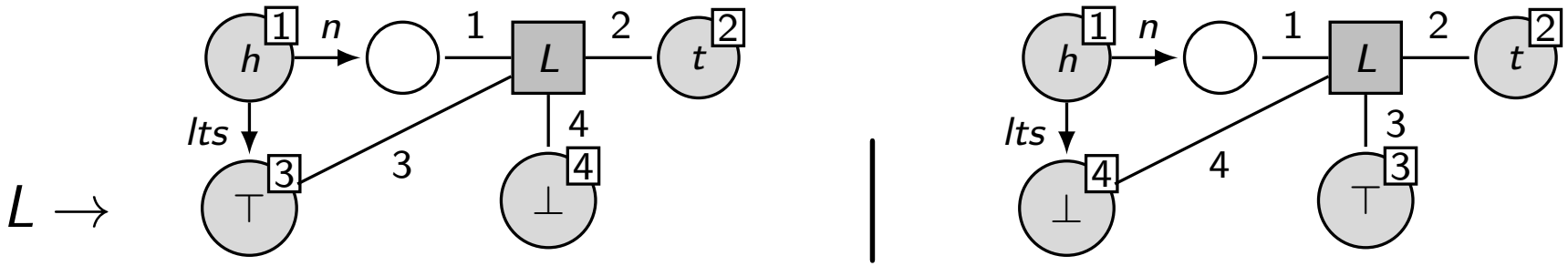
- We add a flag to every node, comparing it to its successor
- We create 3 non-terminals : list(4), sorted list(3), and reverse sorted list(3)



Implementation (Juggrnaut) (2/2)

Note : We only need to compare adjacent elements !

- We add a flag to every node, comparing it to its successor
- We create 3 non-terminals : list(4), sorted list(3), and reverse sorted list(3)



The full grammar contains 16 rules !

Verification

On each of those three tools we were able to verify :

- Absence of null dereferences
- The list is correctly sorted
- The resulting structure is a list

Juggrnaut and TVLA were also able to verify the absence of memory leaks

Error detection

- We now add some errors in the algorithms
- An error is detected if the tool terminates and is not able to prove the same result as before
- The rules/grammars are not modified

Inversing the condition

```
void BubbleSort(List l)
{
  List y,yn,p,t = null;
  bool change = true;
  while(change) {
    change = false;
    p = null;
    y = list;
    yn = y.next;

    while(yn != null) {
      if( y.data < yn.data ) {
        change = true;
        swap(y,yn);
      }
      else {
        move_forward(p,y,yn);
      }
    }
  }
  return y;
}
```

- TVLA detect the error, and is able to prove that the list is sorted in reverse order
- Same for Juggrnaut, thanks to backwards confluence
- Groove did not detect the error as most structures were lost

Removing the first branch

```
void BubbleSort(List l)
{
  List y,yn,p,t = null;
  bool change = true;
  while(change) {
    change = false;
    p = null;
    y = list;
    yn = y.next;

    while(yn != null) {
      move_forward(p,y,yn);
    }
  }
  return y;
}
```

- TVLA detects the error, the list is sorted in 2/5 final states
- Juggrnaut does too, the list is sorted on the corresponding start graph
- Groove detects the error, the list is sorted in 1/3 final states

Making the first branch non-terminating

```
void BubbleSort(List l)
{
  List y,yn,p,t = null;
  bool change = true;
  while(change) {
    change = false;
    p = null;
    y = list;
    yn = y.next;

    while(yn != null) {
      if(y.data > yn.data) {
        change = true;
      }
      else {
        move_forward(p,y,yn);
      }
    }
  }
  return y;
}
```

- TVLA does not detect the error, due to partial correctness
- Juggrnaut does detect it, thanks to separated start graphs
- Groove does not detect it, all terminals are sorted

Outline

Algorithm fidelity

Measured from the number of lines that need to be added or modified

	TVLA	Groove	Juggrnaut	jStar
Reversal	90%	90%	100%	80%
Bubble sort	85%	50%	84%	<50%
DSW	85%	<30%	-	-
Lindstrom	63%	39%	100%	-

Approximate amount of work

Lines	TVLA*	Groove	Juggrnaut*	jStar
Reversal	120	80	100	380
Bubble sort	170	130	200	> 450
DSW	300	> 300	-	-
Lindstrom	300	120	190	-

* Note : TVLA and Juggrnaut rules can be mostly reused for other algorithms

Weeks	TVLA	Groove	Juggrnaut
Reversal	1	2	1
Bubble sort	2	3	4
DSW	3	>3	-
Lindstrom	3	2	2

Resources and time

RAM	TVLA	Groove	jStar
Reversal	2.8Mb	110Mb	7Mb
Bubble sort	880Mb	280Mb	>15Mb
DSW	275Mb	-	-
Lindstrom	343Mb	760Mb	-

Time	TVLA	Groove	Juggrnaut	jStar
Reversal	0.66s	0.5s	2.3s	1.5s
Bubble sort	1189s	20s	1.1s	>3s
DSW	8.6s	-	-	-
Lindstrom	32.4s	47s	0.9s	-

Expressive power

Verified properties :

	TVLA	Groove	Juggrnaut	jStar
Reversal	4/4	3/4	4/4	2/4
Bubble sort	4/4	3/4	4/4	1/4
DSW	4/4	-	-	-
Lindstrom	4/4	3/4	4/4	-

Properties are usually : correctness, structure preserving, null dereferences and memory leaks

Robustness

Errors detected :

	TVLA	Groove	Juggrnaut
Reversal	2/2	1/2 [1]	2/2
Bubble sort	3/6 [2]	4/6	6/6
Lindstrom	3/3	3/3	3/3

[1] : Can be solved by adding two rules

[2] : Not terminating in two cases

Juggrnaut (Hyperedge Replacement Grammars)

+ Quite intuitive, abstraction is automated

+ Can be extended, backward confluence should be automatable

+ Combined to markings, this can be really powerful

– A bit less expressive than the other approaches due to the limitations of HRGs

TVLA (Three-valued Logic)

\+ Abstraction is pretty straightforward

\+ Really intuitive to write

\+ − More expressive than HRGs but less than the other approaches

− TVLA seems to have some problems with non-termination and some special cases

− Debugging is difficult

Groove (Generalized Graph Transformations)

+ One of the most expressive approaches

+ Intuitive, even more with Groove's GUI

– Expressive power means more difficult abstraction (no automation)

– Rules' soundness not always trivial

jStar (Separation Logic)

+ One of the most expressive approaches

+ Combined with sequent calculus, it can reproduce any proof...

– ... but it is sometimes harder than writing the proof by hand

– Hard to automate abstraction while keeping the expressive power

– Debugging is really hard

Outline

Conclusion

- Groove is good for modelling and rapid prototyping
- Juggrnaut and TVLA are almost equivalently powerful, and can check all kinds of properties
- jStar is useful if you need more expressive power, but takes more time to understand

References

[1] D. Distefano.
jstar sources.
github.com/seplogic/jstar.

[2] J. Heinen.
Juggrnaut homepage.
moves.rwth-aachen.de/research/projects/juggrnaut/.

[3] T. Lev-Ami.
Tvla homepage.
www.cs.tau.ac.il/~tvla/.

[4] A. Rensink.
Groove website.
groove.cs.utwente.nl.

Thank you for your attention !