Verification of Pointer Programs: a Tool Comparison

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

September 2, 2015
Outline

1. Introduction
2. Tools and Approaches
3. Case Studies
4. Results
5. Conclusion
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.
  ⇒ 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

⇒ State space explosion (or infinite)

⇒ 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
    ⇒ State space explosion (or infinite)

    ⇒ 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

1 Introduction

2 Tools and Approaches
   - Three-valued Logic
   - Generalized Graph Transformations
   - Hyperedge Replacement Grammars
   - Separation Logic

3 Case Studies

4 Results

5 Conclusion
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
Figure: Kleene’s semantics for operators $\land$, $\lor$ and $\neg$. 
Figure: Kleene’s semantics for operators $\land$, $\lor$, and $\neg$.

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

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

```plaintext
\%
action Set_To_Next(lhs, rhs) {
    lhs(v) = \exists(v_1) rhs(v_1) \land n(v_1, v)
}
```
Let $n(2)$ be a core predicate corresponding to the next field. The action $lhs = rhs \rightarrow n$ is defined as:

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

![Diagram](image-url)
Let $n(2)$ be a core predicate corresponding to the next field.

The action $lhs = rhs \rightarrow n$ is defined as:

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

![Diagram](image-url)
Let \( n(2) \) be a core predicate corresponding to the next field. The action \( lhs = rhs \rightarrow n \) is defined as:

\[
\text{% action Set_To_Next}(lhs, rhs) \{ \\
\quad lhs(v) = \exists(v_1) rhs(v_1) \land n(v_1, v) \\
\}
\]
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!
Rule \texttt{Set\_To\_Next}(lhs, rhs) :
Rule $\text{Set}_\text{To}_\text{Next}(\text{lhs}, \text{rhs})$:

Start graph:

\[ \forall \text{Node} \rightarrow \text{Node} \rightarrow \text{Node} \rightarrow \text{L} \]
Rule \textit{Set\_To\_Next}(lhs, rhs) :

\begin{equation}
\forall \text{Node} \rightarrow \text{Node}_{\text{next}} \rightarrow \text{L}
\end{equation}

\textit{Set\_To\_Next}(x, x)
Rule $Set\_To\_Next(lhs, rhs)$:

\[
\forall Node \xrightarrow{\text{next}} Node
\]

Start graph:

\[
Set\_To\_Next(x, x)
\]
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)
Example: singly-linked list

$$L \rightarrow \begin{array}{c}
\text{1} \\
\text{\rightarrow} \\
\text{\rightarrow} \\
\text{\rightarrow} \\
\text{1} \\
\text{2} \\
\end{array}$$
Example: singly-linked list

\[ L \rightarrow 1 \overset{n}{\longrightarrow} \overset{1}{\rightarrow} \overset{2}{\bullet} \]

Heap configuration:

\[ L \]

Victor Lanvin

Tool comparison

September 2, 2015 12 / 47
Example: singly-linked list

\[ L \rightarrow \quad 1 \xrightarrow{n} \quad \quad 1 \xrightarrow{} L \quad 2 \quad 2 \quad | \quad 1 \xrightarrow{n} \quad 2 \]

Heap configuration:
The two rules form a critical pair:

\[ L \rightarrow 1^n \quad 1 \quad 2 \quad 2 \quad 1 \quad n \rightarrow 2 \]
Juggrnaut (Hyperedge Replacement Grammars) 3/3

The two rules form a critical pair:

\[ L \rightarrow 1^n \quad \text{and} \quad 1 \rightarrow 2 \]

\[ 1 \rightarrow L \rightarrow 2 \]

\[ 1 \rightarrow L \rightarrow 2 \]

\[ 1 \rightarrow L \rightarrow 2 \]
The two rules form a critical pair:

We need to add this third one:

\[
L \rightarrow \quad 1 \quad \overset{n}{\rightarrow} \quad 1 \quad L \quad 2 \quad | \quad 1 \quad \overset{n}{\rightarrow} \quad 2
\]
Input:
Java program, annotated with pre/post conditions
Predicates and actions as inference rules in separation logic

Output:
A proof of the algorithm
Separation logic mainly adds 3 symbols to boolean logic: $\ast$, $\mapsto$, $\text{emp}$.

- $\ast$ is a separating conjunction
- $\mapsto$ can be seen as a pointer property
- $\text{emp}$ holds when the heap is empty
Separation logic mainly adds 3 symbols to boolean logic: *, $\mapsto$, $\text{emp}$.

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

A sequent is defined by the following structure:

$$
\frac{
\Gamma_s \, \ast \, F_{sub}\mid Prem_L \vdash Prem_R
}{\Gamma_s \mid Conc_L \vdash Conc_R}
$$
We can define a node $x$ by a predicate $node(x, y)$ such that:

$$node(x, y) \iff field(x, next, y)( \iff x \rightarrow next = y)$$
We can define a node \( x \) by a predicate \( \text{node}(x, y) \) such that:

\[
\text{node}(x, y) \iff \text{field}(x, \text{next}, y)( \iff x \rightarrow \text{next} = y)
\]

An abstract list can be defined by a predicate \( \text{ls}(x, y) \) where \( x \) is the head and \( y \) is the tail:

\[
\begin{align*}
\text{node}(x, y) & \vdash \text{ls}(y, z) \\
\text{node}(x, y) & \vdash \text{ls}(x, z) \\
\vdash \text{node}(x, y) & \ast \text{ls}(y, z) \\
\vdash \text{ls}(x, z) & \ast \text{ls}(y, z)
\end{align*}
\]

(concretization) (abstraction) (merge)
Outline

1. Introduction

2. Tools and Approaches

3. Case Studies
   - Overview
   - Bubblesort Verification
   - Error Detection

4. Results

5. Conclusion
List reversal

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

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

1. Introduction
2. Tools and Approaches
3. Case Studies
   - Overview
   - Bubblesort Verification
   - Error Detection
4. Results
5. Conclusion
General idea (1/2)
General idea (1/2)

Constant invariant:

\[
\begin{array}{|c|c|}
\hline
\text{list} & \text{sorted list} \\
\hline
\end{array}
\]

External loop invariant:

\[
\begin{array}{|c|c|c|}
\hline
\text{list} & \text{max} & \text{sorted list} \\
\hline
\end{array}
\]
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:
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.
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
Theorem (bounded treewidth of HRLs)

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

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.
Theorem (bounded treewidth of HRLs)

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

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

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

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

- 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.
Theorem (bounded treewidth of HRLs)

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

- 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

---

Victor Lanvin

Tool comparison

September 2, 2015 27 / 47
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).
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).

\[
L \rightarrow \begin{align*}
&\begin{array}{c}
\top & 1 \n\downarrow lts \quad 3
\end{array} \\
&\begin{array}{c}
\bot & 4
\end{array} \\
&\begin{array}{c}
1 & L
\end{array} \\
&\begin{array}{c}
2 & t
\end{array}
\end{align*}
\]
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!
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
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;
}
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
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
Introduction

Tools and Approaches

Case Studies

Results

- User Perspective
- Tools Performance
- Strengths, Weaknesses, Improvements

Conclusion
Algorithm fidelity

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

<table>
<thead>
<tr>
<th></th>
<th>TVLA</th>
<th>Groove</th>
<th>Juggrnaut</th>
<th>jStar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reversal</td>
<td>90%</td>
<td>90%</td>
<td>100%</td>
<td>80%</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>85%</td>
<td>50%</td>
<td>84%</td>
<td>&lt;50%</td>
</tr>
<tr>
<td>DSW</td>
<td>85%</td>
<td>&lt;30%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Lindstrom</td>
<td>63%</td>
<td>39%</td>
<td>100%</td>
<td>-</td>
</tr>
</tbody>
</table>
## Approximate amount of work

<table>
<thead>
<tr>
<th>Lines</th>
<th>TVLA*</th>
<th>Groove</th>
<th>Juggrnaut*</th>
<th>jStar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reversal</td>
<td>120</td>
<td>80</td>
<td>100</td>
<td>380</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>170</td>
<td>130</td>
<td>200</td>
<td>&gt; 450</td>
</tr>
<tr>
<td>DSW</td>
<td>300</td>
<td>&gt; 300</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Lindstrom</td>
<td>300</td>
<td>120</td>
<td>190</td>
<td>-</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Weeks</th>
<th>TVLA</th>
<th>Groove</th>
<th>Juggrnaut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reversal</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>DSW</td>
<td>3</td>
<td>&gt;3</td>
<td>-</td>
</tr>
<tr>
<td>Lindstrom</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
## Resources and time

<table>
<thead>
<tr>
<th></th>
<th>RAM</th>
<th>TVLA</th>
<th>Groove</th>
<th>jStar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reversal</td>
<td>2.8Mb</td>
<td>110Mb</td>
<td>7Mb</td>
<td></td>
</tr>
<tr>
<td>Bubble sort</td>
<td>880Mb</td>
<td>280Mb</td>
<td>&gt;15Mb</td>
<td></td>
</tr>
<tr>
<td>DSW</td>
<td>275Mb</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Lindstrom</td>
<td>343Mb</td>
<td>760Mb</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>TVLA</th>
<th>Groove</th>
<th>Juggernaut</th>
<th>jStar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reversal</td>
<td>0.66s</td>
<td>0.5s</td>
<td>2.3s</td>
<td>1.5s</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>1189s</td>
<td>20s</td>
<td>1.1s</td>
<td>&gt;3s</td>
</tr>
<tr>
<td>DSW</td>
<td>8.6s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Lindstrom</td>
<td>32.4s</td>
<td>47s</td>
<td>0.9s</td>
<td>-</td>
</tr>
</tbody>
</table>
Verified properties:

<table>
<thead>
<tr>
<th></th>
<th>TVLA</th>
<th>Groove</th>
<th>Juggrnaut</th>
<th>jStar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reversal</td>
<td>4/4</td>
<td>3/4</td>
<td>4/4</td>
<td>2/4</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>4/4</td>
<td>3/4</td>
<td>4/4</td>
<td>1/4</td>
</tr>
<tr>
<td>DSW</td>
<td>4/4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Lindstrom</td>
<td>4/4</td>
<td>3/4</td>
<td>4/4</td>
<td>-</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Tool</th>
<th>TVLA</th>
<th>Groove</th>
<th>Juggrnaut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reversal</td>
<td>2/2</td>
<td>1/2</td>
<td>2/2</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>3/6</td>
<td>4/6</td>
<td>6/6</td>
</tr>
<tr>
<td>Lindstrom</td>
<td>3/3</td>
<td>3/3</td>
<td>3/3</td>
</tr>
</tbody>
</table>

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

1. Introduction
2. Tools and Approaches
3. Case Studies
4. Results
5. Conclusion
Groove is good for modelling and rapid prototyping.

Juggernaut 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

    jstar sources.
    github.com/seplogic/jstar.

    Juggrnaut homepage.
    moves.rwth-aachen.de/research/projects/juggrnaut/.

    Tvla homepage.
    www.cs.tau.ac.il/~tvla/.

    Groove website.
    groove.cs.utwente.nl.
Thank you for your attention!