# Comparison of Different Approaches to Automated Verification of Pointer Programs

Victor Lanvin

August 23, 2015

Supervisors : Christina Jansen, Joost-Pieter Katoen,
Christoph Matheja, Thomas Noll

RWTH Aachen University, Germany

**Abstract**

As the complexity of software and algorithms grows, so does the need for formal and automated verification of their behavior. But pointer programs often induce infinite state spaces, as the considered structures can be unboundedly large. Therefore their verification requires some form of abstraction. To achieve this, a lot of different approaches have been developed, giving tools of different capabilities. In this report, we analyze four of those tools -and their underlying approaches- on various algorithms, and give an overview of their strengths and weaknesses.

## 1 Introduction

This report will present the work done during my M1 internship. This internship lasted five months, from March 2nd to July 31st, and took place at the RWTH Aachen University in Aachen, Germany, in the MOVES research group. I would like to thank Christina Jansen and Christoph Matheja for always being there when I had questions, Thomas Noll for taking some of his time to help me during our weekly meetings, and Joost-Pieter Katoen for supervising my internship.

The goal of this internship was to compare four different approaches to the verification of pointer programs; namely three-valued logic (TVL), separation logic (SL), generalized graph transformations (GGT), and hyperedge replacement grammars (HRG). To achieve this, we tried to verify different algorithms (list reversal, bubble sort, Deutsch-Schorr-Waite (DSW) and Lindstrom) using each method. For each approach, we selected a reference tool in which the mentioned algorithms are encoded and verified : TVLA for 3-valued logic [5], jStar for separation logic [2], Groove for GGT [3], and Juggrnaut for HRG [4].

We will first present the different methods, and give an overview of the tools. Then we will give a summary of the work that has been done to adapt and verify the algorithms with those tools, and give more details about a selection of examples that highlight the differences of the approaches. Finally, we will present the results of this comparison. The tools will be compared according to several criteria such as expressive power, usage, and performance.

## 2 Overview of the Different Tools and Approaches

In this section, we will give an overview of the different tools and approaches used during this study. For every tool, we will give more details about its abstraction capabilities, the work that

| $\wedge$ | 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 1: Kleene's semantics for operators $\wedge$,$\vee$ and $\neg$.

is needed to encode an algorithm, as well as some of its particularities.

## 2.1 Three-valued Logic

The first approach is based on three-valued logic and TVLA (standing for Three Valued Logic Analysis), a tool originally developed by Tal Lev-Ami at Tel-Aviv University [5].
In contrast to the usual Boolean logic, three-valued logic (as its name suggests) uses three truth values : true (1), false (0) and a third one standing for unknown, represented by 0.5.

More precisely, TVLA uses first-order three-valued logic with Kleene's semantics. The truth tables for the usual operators are given in Figure 1. In Kleene's semantics, a formula is true for a given interpretation if and only if its truth value is true (1) in this interpretation. Therefore, existential and universal quantifiers have the same meaning as in the usual first-order boolean logic.

As opposed to other tools, TVLA does not reason directly about a program (e.g. written in Java or C++), and has no concepts of pointers or data structures. Instead, you have to rewrite your program in a special, assembly-like language, using predicates and variables. To achieve this, TVLA provides a way to define sets of variables, as well as core and instrumentation predicates. A core predicate can be used to represent a data structure, or an intrinsic property of a pointer (e.g. the value of a pointer); whereas an instrumentation predicate can be used to represent a more high-level property (e.g. the equality of two pointers).

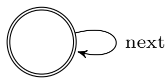
For example, one can define the core predicate $p_y(1)$, which holds for the value pointed by $y$ (i.e. $p_y(v) = 1$ if and only if $y$ points to the value $v$). The definition of a core predicate only describes its intrinsic properties; its value is only defined during the execution of the program. For example, to define $p_y$, TVLA provides the *unique* keyword which conveys the fact that the predicate can hold for at most one individual :

$$\%\text{p } p_y(v_1) \text{ unique}$$

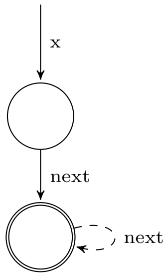
Instrumentation predicates are the predicates that are derived from core predicates. As opposed to the latter, their behavior is entirely determined by their definition. They are useful to reason automatically about a program as they can represent more general properties than core predicates. For example, let $le(2)$ be a core predicate standing for $\leq$, and $p_y$ the core predicate we defined before. Using those, one can define the instrumentation predicate $ple_y(1)$ such that $ple_y(v) = 1$ iff the value pointed to by $y$ is lower of equal than $v$. In TVLA's syntax, this is given by :

$$\%\text{i } ple_y[le, p_y](v) = E(v')\ p_y(v')\ \&\&\ le(v', v)$$

An interpretation of the core predicates and variables represents the state of the heap at some point of the execution of the program. To modify the structure of the heap - and simulate the execution of the program - you have to define *actions*, which reason about predicates. Most actions consist of a *focus* formula (%f) targeting some part of the heap, a precondition (%p), and an *update* formula.
The *focus* formula is applied first, and allows the action to be applied only on the desired structure. The *precondition* is then tested, and, if true, the *update* formula is applied. The role of this update formula is to update the heap according to the action. One can, for example, define the




(a) Fully abstracted non-empty list




(b) Partially abstracted list

Figure 2: TVLA list structures. Arrows represent the predicates. A dashed line stands for the value 0.5, a plain one stands for the value 1. A single circle represents a concrete node while a double circle represents an abstracted structure (here a list).

action that sets a pointer (given by $lhs(1)$) to the value of another pointer (given by $rhs(1)$). Here, $lhs(1)$ and $rhs(1)$ are predicates representing pointers, and evaluate to true iff their parameter is the value of their corresponding pointer :

```
%action Set_Pointer_To(lhs,rhs) {
  \\focus formula
  %f {rhs(v)}
  \\update formula
  {lhs(v) = rhs(v)}
}
```

From there, the translation of the program to TVLA's syntax is quite straightforward : every instruction is given a location and is translated to an action. An action is then given a jump location : if the action has been successfully applied, then the program jumps to this location. If not, then the program tries to apply the next action at the current location.

The strength of TVLA is its capability to automatically abstract structures if they are equivalent, and to concretize them when needed. Two structures are considered equivalent if they satisfy the same predicates (this is observational equivalence). For example, let us consider a list, given by the sole predicate $next(2)$ standing for the successor in a list ($next(x, y) = 1 \iff x.next = y$). All the elements of this list are indistinguishable according to the $next$ predicate. This gives the structure shown in Figure 2a. Now, if we add an $x$ pointer (i.e. a unary predicate) to the first element of the list, the list is automatically concretized to the structure shown in Figure 2b.

Finally, one can check properties (given in three-valued logic) on the structures arising at any point of the execution. TVLA outputs the structures arising in the final state, as well as potential counterexamples to the verified properties.

## 2.2 Hyperedge Replacement Grammars

The second approach has been validated using the tool Juggrnaut, developed at RWTH Aachen University by Jonathan Heinen [4] In this tool, hypergraphs are used to model the heap during the execution of an algorithm. Any modification of the heap (pointer assignation, object creation, ...) causes a modification of the corresponding hypergraph (creation of an edge or a node for example). The grammar is used to concretize and abstract the heap as needed by applying production rules forwards and backwards, respectively.

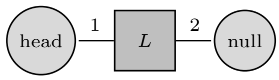



Figure 3: An initial configuration of Juggrnaut. Hyperedges and nodes are respectively represented by squares and circles.

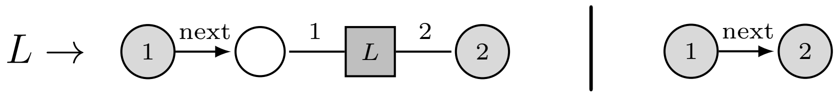



Figure 4: Hyperedge replacement grammar for a list structure represented by a nonterminal $L$

Juggrnaut then produces a state space where every state represents a heap configuration (via a hypergraph), and it is possible to verify properties on this state space using LTL model checking.

**Definition 1.** *(Hypergraph) A* hypergraph *$H$ is defined by a pair $H = (V, E)$, where $V$ is a set of elements (called* nodes *or* vertices*) and $E$ is a set of non-empty subsets of $V$ called hyperedges.*

Informally, a hypergraph can be seen as a graph where edges can connect arbitrarily many elements. Hyperedge replacement grammars (HRGs) are closely related to context-free word grammars, but instead of replacing nonterminal symbols by finite words, nonterminal hyperedges are replaced by finite graphs. The elements connected by a nonterminal hyperedge are seen as *parameters* and can be used in the right hand side of a production rule. A more precise definition is given in [1].

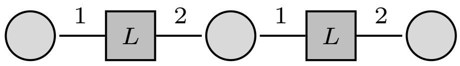
Juggrnaut reasons directly on a Java program via symbolic execution and uses a starting configuration which represent the initial configuration of the heap by a hypergraph. For example, one can consider a nonterminal $L(h, t)$ representing a list linking $h$ to $t$, along with two nodes *head* and *null*, and create the starting configuration shown in Figure 3. This configurations states that there exists a list linking the node *head* to the node *null*. As a hyperedge can connect arbitrarily many vertices, the elements connected by the hyperedge $L$ are numbered (this also gives the orientation of the hyperedge).

Note that the nodes *head* and *null* can represent any value in the program, and the node *null* can even be linked to Java's *null* value.

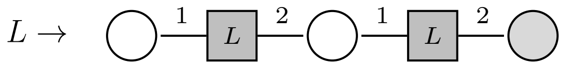
Then, one needs to provide a grammar for the structures manipulated by the program, so that Juggrnaut can "simulate" the behavior of the algorithm. The rules for rewriting the previous nonterminal $L$ are shown in Figure 4. The nodes labeled 1 and 2 are the two parameters of the rewritten hyperedge. This means that when applying a rule to an edge $L$ connecting $v_1$ and $v_2$, $L$ will be replaced by the right hand side where the nodes labeled 1 and 2 are exactly $v_1$ and $v_2$. Basically, those two rules *concretize* the abstract list (represented by $L$) by either adding an element $i$ and a *next* pointer so that *head.next* $= i$, or terminating the list (i.e. joining the head to the tail).

Whenever a program variable is "moved" in the heap, the nonterminals are concretized so that the variable *always* points to a concrete node. If there is no variable in a concrete part of the heap, then this part is abstracted back. Without this mechanism, it would be impossible to keep a finite state space. Informally, Juggrnaut applies the following rule : *"concretize whenever needed, abstract whenever possible"*.

To abstract a node, Juggrnaut applies the rules backwards. This poses the problem of *backward confluence* : for any structures $S_1$ and $S_2$ that can be derived from the same structure $S$ via a sequence of abstractions, then $S_1$ and $S_2$ must be abstracted into a unique structure $S_{abs}$.




(a) A non-abstractable configuration




(b) The rule needed to abstract this structure

Figure 5: Making the grammar backward confluent

Figure 5a shows an example of a hypergraph which is reachable from the previous starting configuration, but cannot be abstracted back with the HRG given in Figure 4. Generally, backwards confluence is resolved in the same way as confluence, that is by resolving all critical pairs. Here, the two rules form a critical pair, therefore we need to add the third HRG rule shown in Figure 5b.

Juggrnaut also provides a way to add *markings*, which can be seen as constant pointers. For example, one can assign a marking to "any List", which means that Juggrnaut will generate a starting configuration for every possible position of this marking in the list before generating the state space. One can then add atomic propositions using those markings (and the program variables), which will be used to mark the states in the resulting state space. For example, adding the proposition $p1 := x = y$ will add the property $p1$ to any state (of the state space) where the variables $x$ and $y$ are equal. This is useful for testing $LTL$ formulae on the resulting state space.

## 2.3 Generalized Graph Transformations

The third approach is based on Generalized Graph Transformations and Groove, a tool originally developed at the University of Twente by Arend Rensink [3].
Groove works with Generalized Graph Transformations. Those can be seen as graph rewriting rules, but with more powerful operators, like *restriction* edges and quantifiers on nodes, features we will explain later on. Groove also supports typed graphs and rewriting rules.

Groove does not reason directly about a Java program. Much like in TVLA, you have to translate your program to use generalized graph rewriting rules and the operators provided by the tool (if/then/else statements, while loops, etc...). Such a program is called a *control program.* With well chosen rules, this is almost a one-to-one translation of the original program.

In a usual graph rewriting system, a rule would be defined by a pair of graphs $R = (lhs, rhs)$. Then, given a graph $G$, if $lhs$ was a subgraph of $G$, then $G$ would be rewritten to $G[rhs/lhs]$ ($G$ where $lhs$ has been replaced by $rhs$). Basically, such a rule creates or deletes nodes and edges in $G$. With typed generalized graph transformations, there is a unification step before applying the rule : in $lhs$, you can mark edges as *restriction* edges, which means that they should *not* be present in the graph for the rule to be applied. One can also add types to nodes, and those types must be unifiable to those defined in the graph.

Moreover, Groove provides a way to add parameters to a rewriting rule. The rule *set_pointer* shown in Figure 6 takes two parameters (indicated by "in1" and "in2") of type *List**, and sets the first parameter to the second. This represents the operation $x = y$, and can be directly called from the control program with the instruction *set_pointer(x,y)* where $x$ and $y$ are nodes of the correct type. In this figure, a dotted line stands for an edge that is deleted by the application of the rule, a dashed line stands for an edge that is created, and a plain edge must be present for the rule to be applied and is left unchanged.
Note that in this rule, we need to take care of removing the previous value of the pointer $x$, which

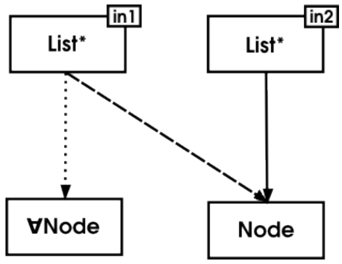



Figure 6: Groove's set_pointer rule.


(a) Groove's list concretization (b) Groove's list abstraction

Figure 7: Groove's list rules.

does not necessarily exist. This is why we add a universal quantifier, which means that for any node (possibly none) pointed by $x$, we remove the edge which points to it.

Unfortunately, the main drawback of Groove is that it is not able to automatically abstract and concretize structures. Therefore, one has to define concretization and abstraction rules by hand, and call them from the control program when needed. For example, one can define the rules for abstracting and concretizing a list as in Figure 7. In those figures, a dashed node is created by the application of the corresponding rule and a dotted one is removed. The universally quantified node unifies with any node having the required edges, and is there to keep the integrity of the list. Note that it would also be possible to use restriction edges to avoid abstracting values stored in a pointer.
From there, those rules should be called every time a pointer is moved in the control program. It is possible, with a simple operator, to apply a rule *"as long as possible"*, which resembles the way Juggrnaut abstracts its structures.

Groove then executes the control program and produces the resulting state space. Each state is a graph, and outgoing transitions are rule applications that are permitted by the control program execution flow. Note that if the same graph arises twice, then this will only create one state, which, with a good abstraction, guarantees that the generated state space is finite.

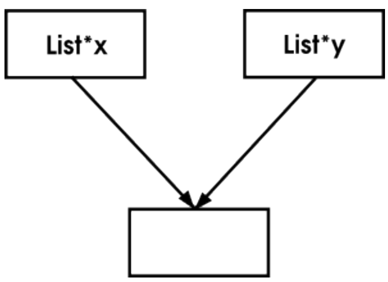



Figure 8: Property rule for "x = y"

Finally, it is then possible to write *property rules* which are rewrite rules with no right hand side. A property rule simply holds in any state where its left hand side is a subgraph of the state's graph. Using those properties, it is possible to check LTL and CTL formulae on the resulting state space. For example, using the rule $x\_eq\_y$ shown in Figure 8, one can show that "x equals y at some point of the execution (always finally)" with the formula $AFx\_eq\_y$.

## 2.4 Separation Logic

The tool used for the last approach is jStar, which was developed at the University of London and the University of Cambridge [2].
The main goal of jStar is to use separation logic to verify object-oriented programs, but it can also be applied to regular pointer programs. Separation logic is an extension of the usual Hoare logic, specifically created to reason about heap-manipulating programs. It makes it easier to reason about pointer data structures, heap segmentation and pointer ownership.

In the fragment of separation logic implemented by jStar, there are three special operators : $*, \mapsto$ and **emp**.

- The binary operator $*$ stands for the separating conjunction : $P * Q$ is true if and only if $P$ and $Q$ hold in two *disjoint* parts of the heap.
- The binary operator $\mapsto$ denotes the points-to assertion. Basically, $p.next \mapsto q$ holds if the field *next* of the value $p$ points to the value $q$ in the current heap. In jStar's syntax, this is written as $field(p, next, q)$.
- Finally, the nullary operator **emp** holds when the heap is empty.

As an example, note that $p.n \mapsto q * p.n \mapsto q$ can never be true, whereas $p.n \mapsto q \wedge p.n \mapsto q$ can. This is because $p.n \mapsto q$ can hold in *at most* one part of the heap, therefore the separating conjunction cannot be satisfied.

More precisely, jStar uses separation logic with sequent calculus and can derive a proof of a program using pre- and post-conditions, as well as some inference rules. Therefore, to verify a program with jStar, one has to provide a Java program annotated with pre- and post-conditions (written in separation logic) and a file with inference rules.

Given three contexts $H_s, H_a, H_g$ (separating conjunctions of formulae), a sequent in separation logic is of the following form :

$$\frac{H_s * SubForm | H_a * Premise_L \vdash H_g * Premise_R}{H_s | H_a * Conclusion_L \vdash H_g * Conclusion_R} \quad (rule)$$

$Premise_L$ and $Conclusion_L$ are the assumed formulae, $Premise_R$ and $Conclusion_R$ are the goal formulae, and $SubForm$ is the subtracted formula. This subtracted formula is used to remove predicates from both sides of the rule without losing information, thus simplifying the rule. Indeed, $H_a * P \vdash H_g * P$ is not equivalent to $H_a \vdash H_g$, thus the need for this subtracted context.

Knowing this, one can define a list structure using a binary predicate $ls$, where $ls(x, y)$ stands for "there exists a list from $x$ to $y$", and a binary predicate *node*, where $node(x, y)$ stands for "$x$ is a list node such that $x.next = y$. Note that in jStar any word (which is not a keyword) is assumed to be a predicate. Therefore, there are no predicate declarations outside of inference rules or pre/post-conditions. Three of the rules defining this list structure would then be :

$$\frac{node(x, y) | \vdash ls(y, z)}{| node(x, y) \vdash ls(x, z)} \quad (concretization)$$

$$\frac{\mid\vdash ls(x,z)}{\mid\vdash node(x,y) * ls(y,z)} \quad (abstraction)$$

$$\frac{\mid\vdash ls(x,z)}{\mid\vdash ls(x,y) * ls(y,z)} \quad (merge)$$

Note that jStar does not distinguish between abstraction, concretization and other rules, and is not able to abstract structures automatically as needed. It unifies and applies rules whenever possible, in the order they are given by the user. Therefore, one has to make sure that the rules (in their given order) are consistent and terminating.

Finally, jStar tries to prove the validity of the postcondition of every function, using the rules and the given conditions for each function. If the proof succeeds, then jStar will output a proof of the program starting from the precondition and ending with the given postcondition. If the proof fails, jStar will either not terminate (if the proof system is not terminating) or output the state of the heap where no rule can be applied (if it finds a deadlock). Note that jStar tries to automatically deduce the loop invariants, and can fail if they are not simple enough.

# 3 Case Studies

To compare the four approaches presented in the previous section, we first had to chose several algorithms, and the properties we wanted to verify for each of those. This choice was important because we wanted algorithms of various difficulties, but they also had to push the tools to their limits and exhibit their strengths and weaknesses.
In this section, we present the four chosen algorithms and the properties we tried to verify. Then we give more details about one of those four case studies, and we finally present one of the robustness studies.

## 3.1 Overview

Firstly, we needed to chose several algorithms of various difficulties, and targeting specific aspects of every tool. Such aspects can be data manipulation, abstraction capabilities, or ease of implementation.
The four chosen algorithms were list reversal, bubble sort, the Lindstrom tree traversal, and the Deutsch-Schorr-Waite variant. For each of those algorithms, we tried to verify the following properties :

1. the structure returned by the algorithm is the same as the input structure (structural correctness)
2. the result contains exactly the same elements as the input (memory preservation)
3. the algorithm behaves as expected, e.g. the list reversal effectively reverses the list (algorithm correctness)
4. there are no null pointer dereferences during the execution

In this part, we will present the four algorithms as well as their most important properties.

**List reversal.** The first chosen algorithm is the list reversal, which simply takes a singly-linked list as input and reverses it. This algorithm was chosen mainly because it is quite simple and short, and therefore constitutes a good introduction to the tools.

A possible implementation using pointers is shown in algorithm 1. The idea is to use three pointers, a *main* pointer $x$ pointing to the current position in the list, a pointer $y$ pointing to the

**Data**: A pointer $x$ to a list $L$
**Result**: A pointer to $L$ reversed
List* $y, t \leftarrow null$;
**while** $x \neq null$ **do**
    `// Swap x and x.next using t and y`
    $t \leftarrow y$;
    $y \leftarrow x$;
    $x \leftarrow x.next$;
    $y.next \leftarrow t$;
**end**
**return** $x$

**Algorithm 1:** List reversal algorithm

previous position, and a temporary pointer $t$ used for swapping elements. The list is then reversed recursively while moving the pointers $x$ and $y$ forward.


Figure 9: List reversal abstract structure

This algorithm can be seen as *local* : it only manipulates three pointers, which always point to adjacent parts of the heap. Therefore, at any point of the execution, the heap can easily be abstracted into the structure shown in Figure 9. This *locality* property will be quite important for this study as it is directly linked to the difficulty of abstracting structures.

The structural correctness usually comes directly from the soundness of the abstraction. Indeed, it is checked by verifying that the resulting structure can be abstracted back into a list.
The algorithm correctness can also be enforced in the abstraction. For example, with graph grammars, this is done by keeping the original order of the elements in memory (using edges), and comparing it to the actual order : if they are identical then the elements are abstracted normally into a *list*, else they are abstracted into a *reversed list*.
The absence of null pointer dereferences is the most implementation-dependent property. For example, TVLA and Juggrnaut automatically detect null dereferences before the state space generation, whereas Groove needs to check this property using a special LTL formula. Overall, this property is easily verified with any tool.
The memory preservation is the most difficult property, because we usually lose too much information about the elements when abstracting them. In TVLA and Juggrnaut, this is handled by keeping track of any element (chosen nondeterministically) during the execution and verifying that it is still reachable in the end. This is more complicated in Groove or jStar and requires a more precise abstraction.

**Bubble sort.** The second algorithm is the bubble sort algorithm, which sorts a list of comparable elements. The implementation given in algorithm 2 sorts the elements in increasing order, and assumes that the nodes of the list are defined with an added integer *data* field.

This algorithm was chosen for a lot of different reasons. First of all, it manipulates and compares data, and efficient data handling is an important point for a verification tool. Secondly, it is less local than the list reversal algorithm as it keeps pointers to various positions in the list, but it is still easier to write and verify than an insertion sort. Finally, there are two nested

**Data**: A pointer $x$ to a list $L$
**Result**: A pointer to $L$ sorted
List* $y, p, yn, t \leftarrow null$;
Bool $change \leftarrow true$;
**while** $change$ **do**
```
    // Initialize pointers to the current position
```
    $p \leftarrow null$;
    $change \leftarrow false$;
    $y \leftarrow x$;
    $yn \leftarrow y.next$;
```
    // Move in the list and swap elements in the wrong order
```
    **while** $yn \neq null$ **do**
        **if** $y.data > yn.data$ **then**
```
            // Swap y.n and yn.n using t as a buffer
```
            $swap(y.next, yn.next, t)$;
```
            // Update position pointers
```
            $p \leftarrow yn$;
            $yn \leftarrow t$;
            $change \leftarrow true$;
        **end**
```
        // Move all pointers forward
```
        $p \leftarrow y$;
        $y \leftarrow yn$;
        $yn \leftarrow y.next$;
    **end**
**end**
**return** $x$

**Algorithm 2:** Bubble sort algorithm

loops, which makes the invariants more complicated and should really push the tools to their limits.

Note that structurally, this algorithm is simpler than the list reversal as there is always only one list, and the global structure is always preserved during the execution. But abstracting the list without losing too much information about the data fields can be complicated. More details about this algorithm in particular will be given in the next section.

**Lindstrom.** The third algorithm is the Lindstrom tree traversal algorithm which, as its name suggests, traverses (in our case) all the elements of a binary tree. Its principal characteristic is that it uses pointers to traverse the tree in-place, i.e. without an additional stack.

A possible implementation is shown in algorithm 3. The main idea behind this algorithm is to traverse the tree with a *cur* pointer and "rotate" the left and right branches of the visited nodes to always keep the whole tree reachable from the *cur* pointer. Using this rotation, one only needs to visit the $left$ branch of the current node to ultimately visit the whole tree.

The Lindstrom tree traversal was chosen mainly because it manipulates another data structure than the two former algorithms. Moreover, it is one of the most local pointer algorithms working on trees (it does not manipulate complex data and keeps the tree structure during the execution), and thus provides a good way to compare the abstraction capabilities of the different tools on simple (yet essential) data structures. A more complicated and challenging variant is proposed in

```
Data: A pointer root to a tree T
Result: All the nodes of T have been traversed
Tree* prev, cur, next, tmp ← null;
// Create a "sentinel" tree and set prev to it
prev ← SENTINEL;
cur ← root;
while 1 do
    // Explore the left part of cur, and "rotate" the cur node
    next ← cur.left;
    cur.left ← cur.right;
    cur.right ← prev;
    // Move forward
    prev ← cur;
    cur ← next;
    // Break if we reached the sentinel, backtrack if we reached a leaf
    if cur = SENTINEL then
        return
    end
    if is_leaf(cur) then
        tmp ← prev;
        prev ← cur;
        cur ← tmp;
    end
end
```

**Algorithm 3:** Lindstrom tree traversal algorithm

the following paragraph.

As in the previous cases, the structural correctness is directly enforced by the abstraction. All the tools use one simple rule for abstraction and concretization which is shown in Figure 10. Proving the absence of null dereferences is again tool-dependent but can always be verified. The memory preservation is the most difficult one, as it requires taking a snapshot of the initial state to compare it to the final state. This is done in the same way as for the list reversal, and was (realistically) not feasible in Groove or jStar.

Finally, algorithm correctness can be verified via a simple LTL formula in Groove and Juggrnaut by marking any node (chosen nondeterministically) and verifying that, at some point, the *curr* pointer was set to it. In TVLA, this is done by marking any visited node and checking that the resulting tree is entirely marked. Note that this approach can also work in Groove or Juggr-

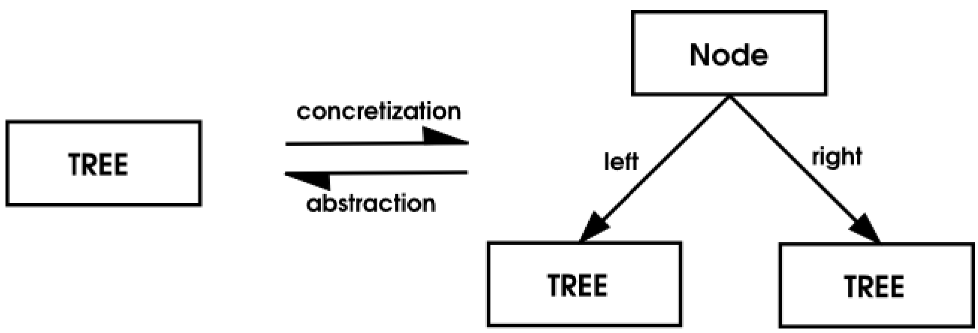



Figure 10: Tree abstraction and concretization rules

naut but would require a more precise abstraction to keep the information about the markings.

**Deutsch-Schorr-Waite.**

---

**Data**: A "current" pointer $x$, a "next" pointer $y$ and a "previous" pointer $t$
**Result**: The pointers and the tree are updated to visit the right part of $x$

```
// The right pointer is reversed so that all the tree is still reachable
   before exploring
```

$x.right \leftarrow t$;

```
// Move pointers forward
```

$t \leftarrow x; x \leftarrow y$;
$x.visits \leftarrow 1$;

---

**Algorithm 4:** Procedure explore_right for DSW

The last algorithm is the Deutsch-Schorr-Waite tree traversal. This is a more complicated variant of the Lindstrom tree traversal which uses a counter (or marking) on every node to keep track of the visited nodes. Similarly to the Lindstrom algorithm, the branches of the tree are often reversed so that the entire tree is always reachable from at least one pointer.

A possible implementation of this algorithm is shown in Algorithm 5. The main idea is that every node will be visited three times : once to explore its right branch, once to explore its left branch (while backtracking from the right branch), and finally once while backtracking from the left branch. Knowing this, we add a *visits* counter to every node, and branch according to this counter : if its value is strictly lower than two, then we explore the right branch while reversing the right pointers so that we can backtrack later. If its value is two, then we do the same with the left branch. Finally, if its value is three, then it means we need to backtrack until we find an node that has not been completely explored. Moreover, while backtracking, we make sure to restore the branches so that the resulting and the initial tree are identical.

The implementation of the function *explore_right* that explores the right branch is also given in Algorithm 4. This function moves the pointers forward while reversing the *right* edge of the current node so that backtracking is possible. The function *explore_left* can be obtained from this one by changing the *right* fields into *left* fields, and by setting the *visits* counter to 2. Moreover, the functions *restore_right* and *restore_left* can also be obtained from those two functions by just changing the order of the parameters : $t$ is the next position, and $y$ the previous one.

The DSW tree traversal algorithm is definitely the most difficult algorithm in this study. It really challenges the abstraction capabilities of the different tools, particularly because it does not preserve the tree structure (there are some states where the algorithm manipulates two disjoint trees). Moreover, the counters (or markings) are essential and must be preserved throughout the verification by the abstraction.

Verifying the DSW algorithm with TVLA is probably easier than with the other tools. This is mainly because the rules from the Lindstrom algorithm can be reused, and adding a *visits* counter is as simple as adding three unary predicates. From there, TVLA automatically abstracts two structures only if they are indistinguishable by those three predicates, which ensures that there will be no loss of information. All the properties (structural correctness, algorithm correctness, memory preservation and absence of null dereferences) can be verified with TVLA. With Groove, we have only been able to verify the the absence of null dereferences. Verifying the three other properties should be possible, but requires so many abstraction rules that proving the soundness of those rules becomes harder than proving the algorithm by hand. Finally, proving this algorithm

```
Data: A pointer x to a tree T
Result: All the nodes of T have been traversed
Tree* y, t ← null;
x.visits ← 1;
while 1 do
    if x.visits ≤ 1 then
        // Explore the right part of x
        y ← x.right;
        if y == null or y.visits > 0 then
            // The right part has already been explored
            x.visits ← 2;
        end
        else
            explore_right(x, y, t);
        end
    end
    else if x.visits == 2 then
        explore_left(x, y, t);
    end
    else
        // This part of the tree is done :  restore the pointers and backtrack
        y ← x; x ← t;
        if x == null then
            // All the tree is done :  return
            return
        end
        else
            // Restore x.right or x.left depending on the value of x.visits and
               continue
            if x.visits ≤ 1 then
                restore_right(x,y,t);
            end
            else
                restore_left(x,y,t);
            end
        end
    end
end
```

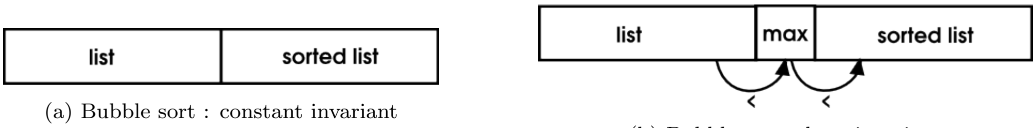
**Algorithm 5:** DSW tree traversal algorithm

with Juggrnaut is still a work in progress.

## 3.2 Particular Case : Bubble Sort

In this section, we will give more details about one of the four case studies, namely bubble sort, as it gave the most interesting results. We will present the difficulties encountered as well as the solutions and abstractions we developed for every approach. Unfortunately, due to some problems with jStar, there will be only few results about separation logic.

**Generalities.** The general idea of the bubble sort is to start at the head of the list and move




(a) Bubble sort : constant invariant

(b) Bubble sort : loop invariant

Figure 11: Bubble sort loop invariants

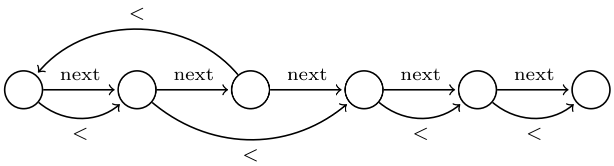



Figure 12: A structure combining a list and an ordering defined by a permutation

forward while swapping every pair of elements in the wrong order, and repeat this as long as the list is not sorted. This is similar to taking the maximal element of the list, putting it at the end, and repeating this operation on the remaining elements. Thus, we have one *constant* invariant (true at any step of the execution), and one *loop* invariant (true after every outer loop iteration), shown in Figure 11. Basically, the list can always be decomposed into a list followed by a sorted list, and any iteration brings the maximal element of the list just in front of the sorted list.

We want to prove the following properties on this algorithm :

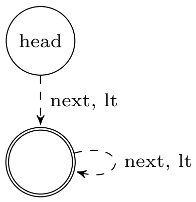
1. the list structure is preserved during the execution (structural correctness)
2. the resulting list is sorted (algorithm correctness)
3. absence of null dereferences
4. the resulting list is a permutation of the list given as input (memory preservation)

The main difficulty of bubble sort is that it manipulates data (integers) and often compare pairs of elements, and that none of the tools are able to handle data natively. Therefore, we need to find a way to represent the ordering without using data. The following paragraphs will present a structure (and its properties) that can be used to prove this algorithm, and the properties 2 and 4 in particular.

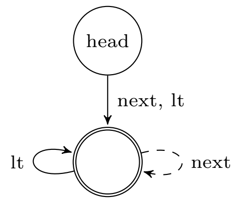
First of all, the structure should correctly represent a list, and should retain enough information about the ordering of the elements when abstracted (for example, we should be able to know if the abstracted elements are sorted or not). For this, one can consider two abstract structures, a list and a sorted list, having basically the same abstraction and concretization rules.

Secondly, the structure should correctly define an ordering on the elements, so that it is possible to compare two elements and branch accordingly. For this, one can either directly define a total ordering on the elements (via oriented edges between every pair of elements for example), or (more easily) a permutation of the elements. Such a permutation on a list structure is shown in Figure 12. Note that the corresponding total ordering can be easily deduced by taking the transitive closure of this permutation, so the two approaches are equivalent.

**Verification with TVLA.** It is possible to implement directly this structure in three-valued logic. Basically, we take the list structure shown before (with a *next* value initially set to 0.5),

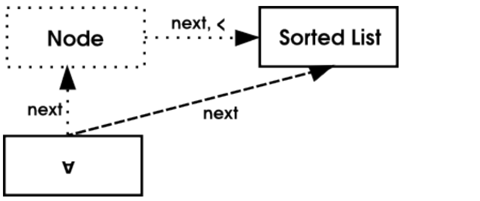



(a) Partially abstracted, non-empty and unsorted list

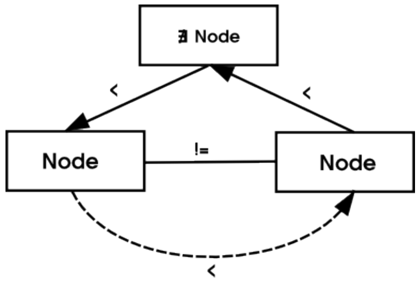



(b) Partially abstracted sorted list with at least two elements

Figure 13: TVLA structures arising during the verification of the bubble sort




(a) GGT abstraction rule for a sorted list. The < edges pointing to the abstracted node are lost.




(b) GGT rule to create a total ordering. Notice the unoriented edge labeled "!=" to ensure that the two nodes are different, and the quantifier to prevent cycle creation.

Figure 14: Some rules used to verify bubble sort with Groove. As before, dotted (resp. dashed) edges stand for deleted (resp. created) edges. Restriction edges are not shown but are implied by the ∄ quantifier.

and add an *lt* binary predicate between every pair of nodes also initialized to 0.5. This value will force the evaluation of both cases when a comparison is encountered during the execution.

Moreover, it is possible to force the *lt* predicate to be transitive and antisymmetric with simple keywords (or via additional rules), so that the ordering is always consistent. In this case, the transitivity guarantees that a sorted list will be abstracted as such, as no nodes will be distinguishable by the *lt* predicate in a sorted list. Some examples of structures arising during the execution are shown in Figure 13.

This is actually enough to verify the bubble sort in TVLA. But because of the complexity of this approach (the graph representation of this structure is basically a complete graph), and the fact that TVLA applies a lossless abstraction on this graph, there are a lot of possible configurations (a few hundreds of thousands), and verifying the algorithm in TVLA can take up to one hour. This shows that developing a sound and effective abstraction for this problem can be difficult, as we will see in the following implementations.

**Verification with Groove.** The implementation of this structure in Groove is really similar to TVLA's. However, as abstraction is not computed automatically, we have to provide consistent abstraction rules for every structures, and call them as needed. Moreover, to simplify this process, we assume that there is a sorted list at the end of the initial list (it is, in fact, empty), and we always abstract the sorted list from the left, according to the rule given in Figure 14a.

Note that this method of abstracting the sorted list can lead to some unexpected behaviour : it is possible to concretize all the elements of the sorted list, then abstract them into an unsorted

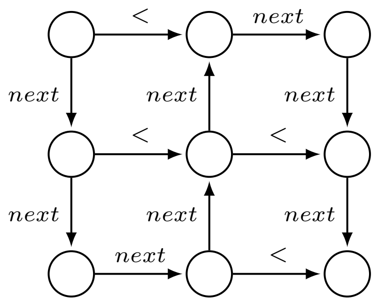



Figure 15: Writing a permutation of a list of length 9 as a 3-grid.

list, thus leading the verification back to the initial state. This is, in fact, not a problem : the abstract state space is still finite, we just need to make sure to only consider the finite paths when verifying an LTL formula (to achieve this, Groove automatically marks all terminal configurations).

Concerning the ordering, thanks to restriction edges and quantifiers, it is also possible to define a total ordering with generalized graph transformations. For example, each time a node is concretized, one can iterate over all existing nodes and order them, while making sure that no cycle is created during the process. This can be done by using the rule given in Figure 14b and applying it "as long as possible" every time a node is created.

Finally, it is realistically impossible to provide a lossless abstraction like the one used by TVLA. Therefore, the abstraction we used did not retain any information about the ordering (except for the sorted list), and the ordering is re-created every time a node is concretized. This is again not a problem, as the case we are trying to verify is a subcase of this one (it is the only case where the created ordering is exactly the same as the previous one).

**Verification with Juggrnaut.** Due to intrinsic limitations of hyperedge replacement grammars, implementing this structure in Juggrnaut is impossible. Indeed, we have the following theorem about expressiveness of HRGs :

**Theorem 1.** *(Bounded treewidth of HRLs)*
*Let $\mathcal{L}$ be an Hyperedge Replacement Language (HRL). There exists an integer $N$ such that, for every graph $G \in \mathcal{L}$, $treewidth(G) \leq N$.*

The treewidth of a graph can be defined in several ways (for example as the largest clique in a chordal completion of the graph), and can be seen informally as a parameter indicating "how close" the graph is from a tree. The treewidth of a graph with $n$ vertices is at most $n-1$ (obtained for a complete graph or a grid), and at least 0 (obtained for a tree). Note that the treewidth of an oriented graph is defined by being equal to the treewidth of the undirected underlying graph.

The proof of this theorem is given in [1]. It stems from the fact that applying a rewriting rule to a graph cannot increase the treewidth of this graph past the treewidth of the rule. Formally, if $G$ rewrites to $G'$ using a rule $R \to R'$, then $treewidth(G') \leq max(treewidth(G), treewidth(R'))$.

Using this theorem, it is easy to show that producing totally ordered lists is out of scope, as it is equivalent to producing n-cliques (which set, as we said before, does not have a bounded treewidth). Moreover, simply producing permutations as shown in Figure 12 is also out of scope, because $n$-grids are a subset of those permutations (as shown in Figure 15), and the set of all $n$-grids does not have a bounded treewidth either.

However, it is still possible to verify the bubble sort algorithm with HRGs by remarking that :

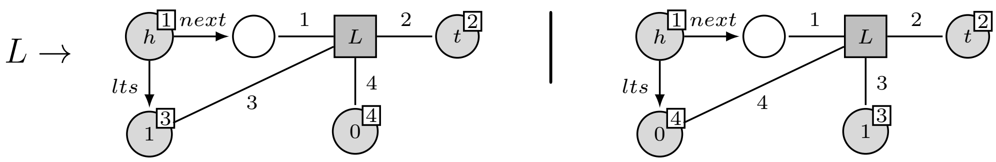



Figure 16: The two hyperedge replacement rules used to concretize the non-terminal L. The indices of the parameters are given in the white squares. By applying those nondeterministically, we test every possible case.

1. we only care about comparing successive elements
2. we do not necessarily need the ordering to be consistent, as long as we test all mathematically possible cases. For example, the case where the list has three elements $x, y, z$ where $x < y$, $y < z$ and $z < x$ is impossible, but will still be considered.

Therefore, we modified the Java program to add a new field to the $Node$ class, so that it now has an $lts$ flag indicating if the node is lower that its successor. Formally, $node.lts = 1 \iff node.value < node.next.value$. We then use this $lts$ flag to compare the nodes in the algorithm : the condition $if(y.value > y.next.value)$ becomes $if(y.lts == 0)$.

We now need to make sure in the HRG that both cases ($lts = 0$ and $lts = 1$) are tested. To achieve this, our non-terminal $L$ representing a list now takes four parameters : the head and the tail of the list as well as two distinguished nodes representing 0 and 1. The rules used to concretize this non-terminal are shown in Figure 16. Then, we create a non-terminal $L_{sort}$ representing a sorted list and taking only three parameters, as only the value 1 is needed (indeed, in a sorted list, all the elements have their $lts$ flag set to 1).

Interestingly enough, to make this grammar backwards confluent, one has to create another non-terminal $L_{revsort}$ standing for a list sorted in reverse order (i.e. such that all of its elements have their $lts$ flag set to 0). This will be quite important in the next section as it will allow Juggrnaut to detect some errors. With the three non-terminals, the full grammar contains sixteen rules and is shown in Appendix A.

**Conclusion.** In conclusion to this case study, we have been able to verify that the list is correctly sorted, the absence of null dereferences and that the resulting structure is still a list on each of the three tools. Moreover, Juggrnaut and TVLA were also able to verify that they were no memory leaks and no added elements to the resulting list (i.e. the resulting list is exactly a permutation of the initial one).
With jStar, we have been able to prove the absence of null dereferences as well as the fact that most operations preserve the list structure, but finding a complete proof of the algorithm is still a work in progress.

### 3.3 Robustness Study

A verification tool should not only be able to verify an algorithm, it should also be able to detect any possible error in a program. It is possible that the abstractions we implemented are too strong and can "hide" a programming mistake. Moreover, verification tools often rely on *partial correctness*, which means that one has to make sure that a program is terminating before verifying it.

Thus, it seems necessary to also study the robustness of each tool. For this study, we added some errors in the algorithms before verifying them *with the exact same rules as before*. We consider an error to be detected by a tool if and only if the tool terminates and is not able to prove the properties of the correct algorithm (given in the previous section).

In this section, we will present the robustness study based on the bubble sort algorithm.

**Error 1 : inversing the condition.** The first error consists in inversing the comparison in the bubble sort algorithm. Basically, the test $if(y.data > yn.data)$ becomes $if(y.data < yn.data)$. This error is interesting because the resulting algorithm sorts the list in decreasing order (rather than increasing), and it shows in what extent the abstractions, grammars and rules can be applied to other valid algorithms.

In TVLA, the error is detected : it is not able to prove that the list is still sorted in increasing order at the end. Moreover, by adding only one rule, it is possible to verify that the list is sorted in decreasing order. All the other properties are still verifiable.

In Juggrnaut, the error is also detected, and it is possible to verify that in any final configuration, the list is sorted in decreasing order. Indeed, to make the grammar backward confluent, we needed to add a non-terminal $L_{revsort}$ standing for a list sorted in decreasing order, which makes it possible to verify this property.

In Groove, this is more difficult. The grammar we wrote is really tailored to the initial algorithm. In particular, it is not capable of abstracting a list sorted in decreasing order, which causes a state space explosion as the number of concrete elements grows indefinitely. However, it is still possible to detect the error and verify that the list is sorted in decreasing order by adding only three rules (to concretize and abstract such a list). Nevertheless, the error cannot be considered "detected" as it requires changing the grammar.

**Error 2 : removing the first branch.** The second error consists in simply removing the first branch. This means that, instead of conditionally branching according to the test $y.data > yn.data$, the algorithm will just go into the second branch. The algorithm becomes equivalent to a simple list traversal, but with dead code. This error shows how the different tools behave when confronted to dead code and unused pointers. Moreover, one can remark that such an algorithm will still sort the list in some cases (exactly the cases where the list is sorted in the beginning), and the tools should be able to handle this.

In TVLA, the error is detected as it reports that the list is unsorted in some terminal configurations. Moreover, it is also able to prove that the list is sorted in two out of five final states (those two states correspond to a list with one element, and a list with at least two elements). TVLA also performs well when confronted to dead code as the number of configurations for this algorithm and for a simple list traversal are similar, but the memory consumption is a bit higher (probably due to the fact that TVLA has to store an additional node for the unused pointer in every configuration).

In Juggrnaut, the error is also detected. Because of some intrinsic properties of Juggrnaut, one has to provide three starting configurations to verify the bubble sort algorithm exhaustively (i.e. for any possible list). Those three configurations correspond to the cases of a list sorted in increasing order, a list sorted in decreasing order, and a (strictly) unsorted list. Therefore, the verification has to be run three times but produces the expected results on each starting configuration (for example, for the sorted configuration, the list is still sorted in the end). However, because the grammar is more complicated (and the verification has to be run several times), it takes longer and creates a larger state space than verifying a simple list traversal algorithm.

Groove also detects the error, as the list is sorted in only one out of three final configurations. Those configurations correspond to an unsorted list, a sorted list, and an unsorted list followed by a sorted list. Moreover, as Groove directly simulates the execution of the program, it is the

| | 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 | - |

(a) Number of verified properties per tool and algorithm.

| | 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 |

(b) Number of errors detected per tool and algorithm.

Figure 17: Expressive power and robustness

most efficient tool when confronted to dead code. Indeed, the number of configurations and the execution time are exactly the same as for a simple list traversal algorithm.

**Error 3 : infinite loop in the first branch.** The third error consists in keeping both branches, but doing nothing in the first one. Therefore if the list is initially sorted then the program will always execute the second branch and will terminate, whereas if two elements are not sorted then the algorithm will always execute the first branch and will not terminate. This error shows how the different tools behave when the verified algorithm does not always terminate.

In TVLA, the error is not detected : there are two terminal configurations, and the list is sorted in both of them. Therefore, when verifying the *sorted* property, TVLA does not detect that the algorithm does not terminate on some initial configurations, and is able to verify the property. Interestingly, this error also showed that TVLA can be a bit unstable when dealing with non-terminating algorithm. Therefore, one has to be careful with termination when implementing an algorithm in TVLA.

In Juggrnaut, the error is actually detected. We have seen before that, because of some particularities of Juggrnaut, verifying the algorithm requires three starting configurations. We also saw that the algorithm terminates if and only if the list is sorted. Therefore, Juggrnaut is able to tell that the algorithm does not terminate on two out of three starting configurations (there are no terminal states), and that the algorithm terminates (and the list is sorted) on the third starting configuration. This would not be possible if we were able to run Juggrnaut only once on a more general starting configuration. However, this detection can be seen as lucky, as it would probably not happen in other cases.

Groove does not detect the error. Moreover, as Groove directly simulates the execution of the program and lacks automatic abstraction, the state space generation does not terminate : the length of the list grows indefinitely. If one halts the generation, it is possible to show that the list is sorted in every terminal configuration. It is possible to add some abstraction rules to make the state space generation terminating, but it is not sufficient to detect the error.

# 4 Results and Comparison

In this section we will give the main results of this tool comparison according to several criteria such as expressive power, performance or ease of use. We will also give an overview of the strengths and weaknesses of each tool, as well as some ideas on how to improve them.

Note that some data may be missing in the provided tables, this can either be because the data is irrelevant to this study or because we have not been able to verify an algorithm with the corresponding tool yet.

[1]Groove can detect the second error by adding two rules
[2]TVLA does not terminate in two cases

| | TVLA | Groove | Juggrnaut | jStar |
|---|---|---|---|---|
| Reversal | 0.66 | 0.5 | 0.6 | 1.5 |
| Bubble sort | 1189 | 1.5 | 1.1 | >3.0 |
| DSW | 5.6 | - | - | - |
| Lindstrom | 18.4 | 16.3 | 0.9 | - |

(a) Verification time per tool and algorithm (in seconds)

| | TVLA | Groove | Juggrnaut |
|---|---|---|---|
| Reversal | 57 | 47 | 912 |
| Bubble sort | 300k | 328 | 40.7k |
| DSW | 11.5k | - | - |
| Lindstrom | 32.9k | 6.4k | 160k |

(b) State space size per tool and algorithm (in number of states)

Figure 18: Performance comparison of the tools

## 4.1 Expressive Power, Robustness

Firstly, we compared the expressive power of the tools according to the number of properties they are able to verify on each algorithm. The Figure 17a shows a summary of this comparison. It may be interesting to note that Groove is not realistically able to verify the memory preservation property (i.e. that no elements are deleted nor created) as it requires adding a lot of rules. TVLA is the tool that performs the better, closely followed by Juggrnaut. Both tools are able to verify any property, but implementing the DSW algorithm on Juggrnaut is complicated and is still a work in progress. Moreover, jStar must be able to verify any property on those algorithms (given that such a property can be verified in separation logic), but the lack of support for the tool really hinders its performance.

Secondly, we compared the robustness of the tools according to the number and the kind of errors they are able to detect. Such errors can simply add a null dereference, or make an algorithm non-terminating. The Figure 17b shows the results of this comparison. Of the three compared tools, Juggrnaut is the most robust as it detected every error. This, however, comes at a price : one may have to run the verification multiple times on several starting configurations to verify the algorithm exhaustively.

TVLA is also very robust as it abstracts structures with very little loss of information. However, the tool is unfortunately a bit unstable and struggles with non-terminating algorithms. Groove is probably not the best tool to automatically detect errors in an algorithm, but its GUI (and graph grammars in general) are intuitive enough so that one can detect most errors by simply looking at the generated configurations.

Overall, all the tools are able to verify properties and detect errors linked to null dereferences, sometimes even before the state space generation. The most difficult property is probably the memory preservation property, regardless of the algorithm. The tools that are able to deal with it (Juggrnaut and TVLA) both have a way to limit the abstraction (markings or snapshotting) so that they can keep some precise information about one element in particular.

## 4.2 Performance, Ease of Use

**Performance.** The second criterion of comparison was the relative performance of the tools. A summary of the results of this comparison is given in Figure 18. Overall, Juggrnaut is the fastest tool, with execution times around one second for every algorithm. It may also be interesting to note that it handles large state spaces pretty well, as seen with the Lindstrom algorithm that generates a large number of states but takes less than one second to verify. The overall greater number of states generated by Juggrnaut compared to the other tools is mainly due to the use of markings that prevent abstraction.

One can remark that Groove generates relatively small state spaces, this is because the abstraction rules we used (in the case of the list reversal and bubble sort) are really strong and tailored to the algorithms. Moreover, one should note that such a small state space comes at a

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

(a) Approximative amount of work per tool and algorithm (in weeks)

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

(b) Total number of lines given as input per tool and algorithm [3]

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

(c) Algorithm fidelity : approximative percentage of lines of code that need to be changed in order to verify the algorithm

Figure 19: Ease of use of the different tools

price : Groove is not able to verify the memory preservation property.

Finally, we can remark that TVLA is a bit longer than the other tools, especially on bubble sort where it takes almost twenty minutes (and creates hundreds of thousands of states). This is mostly due to the fact that it performs an almost lossless abstraction, and the structures manipulated by a sorting algorithm are really difficult to abstract without losing information. However, one should also consider the fact that, independently of the execution time, verifying the bubble sort with TVLA is probably easier than with the other tools as we will see in the following results.

**Ease of use.** The last criterion of comparison was the ease of use of every tool, for a user that has little to no experience with such a tool. To compare this, we used several criteria such as the number of lines a tool requires as input, or the time it takes to implement an algorithm.

The Figure 19a shows an approximation of the time (in weeks) it takes to implement each algorithm in each tool. The values are corrected to take into account the fact that the experience acquired during this study made each successive implementation easier. Moreover, some code can sometimes be reused for multiple algorithms (for example, most of the rules used to verify the DSW algorithm in TVLA can be reused to verify the Lindstrom algorithm). This also has to be considered when looking at the Figure 19b which gives an approximation of the number of lines of code required to verify each algorithm.

Overall, this shows that verifying an algorithm in TVLA or Juggrnaut is relatively easy most of the time, but implementing the bubble sort in Juggrnaut was harder than in the other tools (as shown in the previous section). Moreover, one should note that although implementing an algorithm in Groove seems to require a long time, this is mostly due to the lack of automatic abstraction, and Groove can (and should) totally be used for rapid prototyping and modeling. Indeed, for simple algorithms such as a list reversal, Groove is the tool that requires the fewest lines of codes.

The Figure 19c shows the amount of modifications required to implement an algorithm in a tool. This is based on an original pseudo-code algorithm : a modification is the addition or the deletion of a line in the original algorithm. For example, when implementing the list reversal algorithm in TVLA, approximately 10% of lines had to be added. This shows that most algorithms

[3] It can be important to note that, in the case of TVLA and Juggrnaut, the rules and grammar can often be reused for other algorithms

can be directly translated to be used by Juggrnaut or TVLA without modifying them too much.

However, most of the time an algorithm cannot be translated directly to be verified by Groove or jStar. Indeed, as Groove lacks automatic abstraction, one should call abstraction and concretization rules explicitly in the algorithm. In jStar, one has to make sure that the invariants are clear enough in the code so that jStar can automatically deduce them.

## 4.3 Summary

To conclude this comparison, we will now give a summary of the strengths of weaknesses of each tool, as well as the underlying approaches.

**Juggrnaut (HRGs).** Although verifying the DSW algorithm with Juggrnaut is still a work in progress, this study showed that Juggrnaut can verify all kinds of properties on various algorithms, and it is also one of the fastest tools. Moreover, hyperedge replacement grammars are intuitive and easy to write, even with limited knowledge (they are similar to graph grammars). Furthermore, abstraction is also really intuitive when using HRGs, as it is more or less equivalent to just applying the rules backwards.

However, HRGs are less expressive than the other approaches, and it can be difficult to find a suitable backward confluent grammar for an algorithm if it uses complicated structures. This problem is explored a bit more in the following section.

**TVLA (Three-Valued Logic).** Although less efficient when it comes to execution time, TVLA is probably more expressive than Juggrnaut. It is really easy to express most structures and rules in three-valued logic, without having to modify them. Moreover, abstraction in three-valued logic is really straightforward, and can be as precise as needed.

However, abstraction is also the main drawback of TVLA. Indeed, when the structures are too complicated, the verification of an algorithm can take a really long time. Moreover, TVLA seems to have some problems with non-terminating algorithms and some special cases that make it a bit unstable. It should also be noted that debugging in TVLA is difficult as it can generate hundreds of thousands of structures, and one sometimes has to go through them all to find a bug.

**Groove (Generalized Graph Transformations.)** Out of the four compared tools, Groove is definitely the most intuitive one. Indeed, its powerful GUI combined with the readability of graph transformations makes it possible to understand any program with limited knowledge. It is also easy to implement and debug most rules to execute an algorithm. Overall, GGTs are extremely powerful and are able to represent any structure and algorithm; but this power comes at a price as it makes abstraction really hard to automate.

Indeed, Groove's main drawback is the lack of automatic abstraction. As opposed to other rules (pointer manipulation, tests) abstraction rules can be complicated, and verifying an algorithm can get really difficult simply because of this. For example, in the case of the DSW algorithm, verifying it with Groove was harder than verifying it by hand.

**jStar (Separation Logic.)** Along with generalized graph transformations, separation logic is the most expressive approach, as it can represent any structure. Moreover, combined with sequent calculus as done in jStar, it is able to "reproduce" any proof that is feasible by hand.

However, jStar's lack of support and proper debugging features make it really difficult to verify an algorithm without being familiar with the tool. Moreover, as for GGTs, the expressive power of separation logic makes abstraction hard to automate.

### 4.4 Future Work

Other than comparing different approaches to the verification of pointer programs, this study also aimed at finding possible ideas to improve Juggrnaut, as it is the tool developed at the RWTH Aachen University. This section will explore some of the problems encountered with Juggrnaut and some possible solutions.

One may have remarked that although finding a HRG to verify a particular algorithm can be easy, finding an equivalent backward confluent grammar can be more difficult. Of course, removing the backward confluence requirement is impossible as the rules still need to be applied backwards for abstraction. However, one can wonder if it would be possible for the tool to automatically deduce a backward confluent grammar from an arbitrary one. Unfortunately, it is impossible in the general case : there is no backward confluent HRG that generates the string graphs representing the language $L = a^*(a + b)$, or even a "small" superset of $L$.

Indeed, if it were possible, then there would exist a derivation $S \to^* a$ and a derivation $S \to^* ab$. Therefore, it would be possible to abstract $aa \in L$ into $SS$ and produce $abab \notin L$. It is actually possible to prove that the smallest backward confluent language containing $L$ is $\Sigma^* = (a + b)^*$.

It would be possible to solve this problem by using a feature inspired by TVLA : being able to specify that a rule should only be used for abstraction or concretization. However, a grammar using such a feature would be more prone to produce infinite state spaces, and one would have to be careful with the concretization-only rules.

It may also be interesting to be able to force parallel application of certain rules. This means that, if a rule can be applied, then it is applied to every available non-terminal in parallel. It may be useful to combine this approach with the previous one, so that one is able to use abstraction-only rules in parallel for example (note that it would help solving the problem showed before). Such a feature would also make it possible to verify algorithms that use particular structures, such as the insertion in a binary tree.

Finally, we also thought about the possibility of running Juggrnaut on multiple starting configurations at once. This may, for example, be useful in the case of the bubble sort algorithm where there were three starting configurations. However, we have also seen in the corresponding case study that having to run the verification thrice made it possible to find an error, which would not have been possible otherwise. A good solution might be to run the verification only once, but add a special case (or a simple message) when the algorithm does not terminate on one of the starting configurations.

## 5 Conclusion

In this study, we exposed the strengths and weaknesses of four different tools and their underlying approaches. We showed that Groove and generalized graph transformations are well suited to rapid prototyping and modeling, but Groove is probably not the best choice as a verification tool as it lacks automatic abstraction.

We also showed that Juggrnaut and TVLA have similar performance and expressive power, and the underlying approaches (three-valued logic and hyperedge replacement grammars) have similar abstraction capabilities. Overall, except in some particular cases where HRGs are not powerful enough, the choice of one over the other is probably a matter of personal taste. Separation logic is probably the most powerful approach of the four, but jStar lacks intuitiveness and efficient abstraction.

Finally, we also provided some ideas about how to improve the expressive power and efficiency of Juggrnaut.

# References

[1] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer, 1992.

[2] University of Cambridge and University of London. jStar Verifier. `http://www.jstarverifier.org/`, 2008.

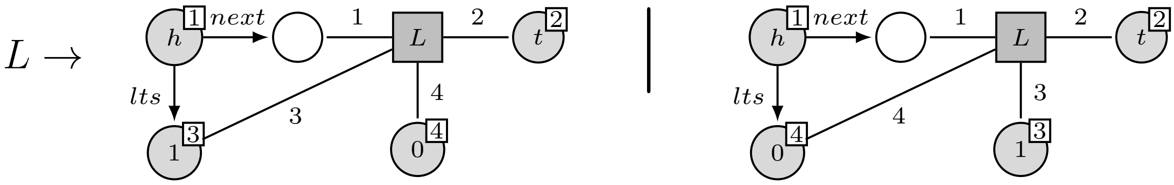
[3] University of Twente. Groove (Graphs for Object-Oriented Verification). `http://groove.cs.utwente.nl/`, 2004.

[4] RWTH Aachen University. Juggrnaut Project. `moves.rwth-aachen.de/research/projects/juggrnaut/`, 2011.

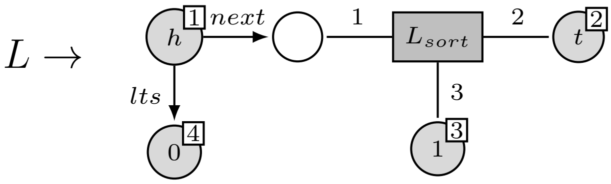
[5] Tel Aviv University. TVLA (Three Valued Logic Analysis) Home Page. `http://www.math.tau.ac.il/~tvla/`, 1999.

# A Full HRG for Bubble Sort

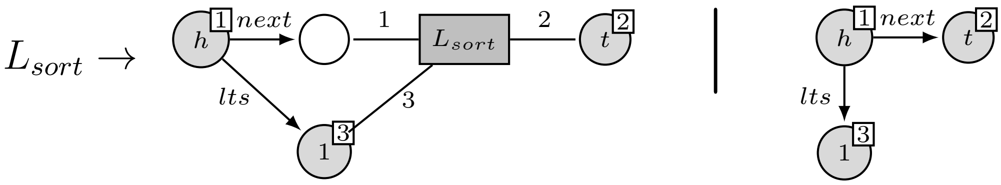
Adding a concrete element to $L$ :



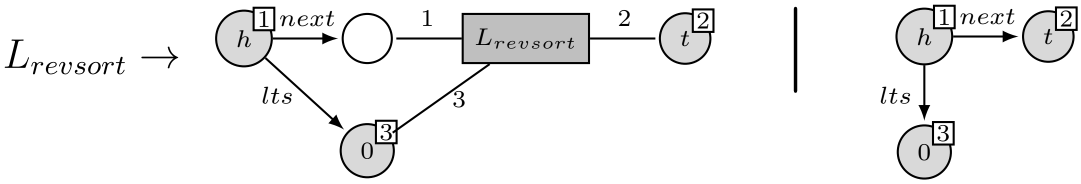

Finishing $L$ with a sorted list (possibly of length one) :



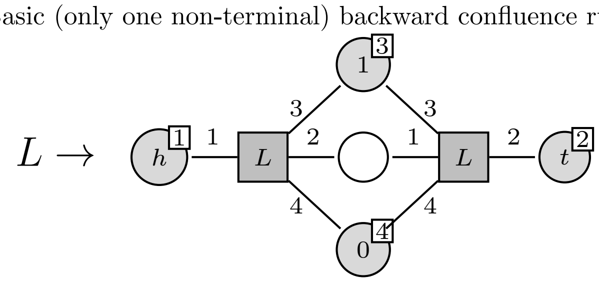

Concretization of a sorted list :

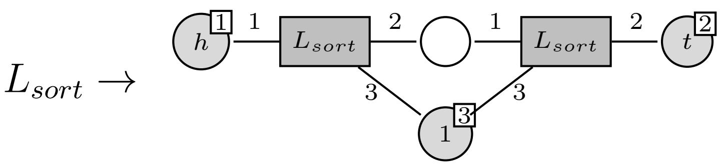



Concretization of a list sorted in decreasing order :

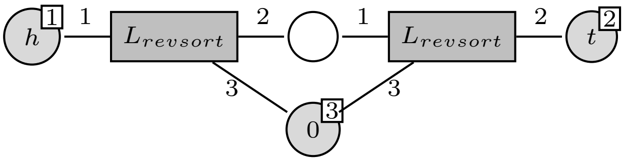



Basic (only one non-terminal) backward confluence rules :



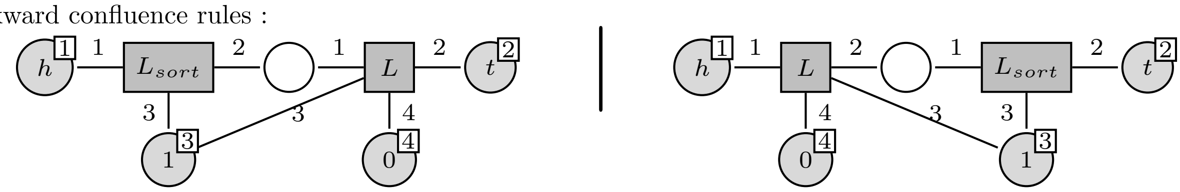



$L_{revsort} \rightarrow$

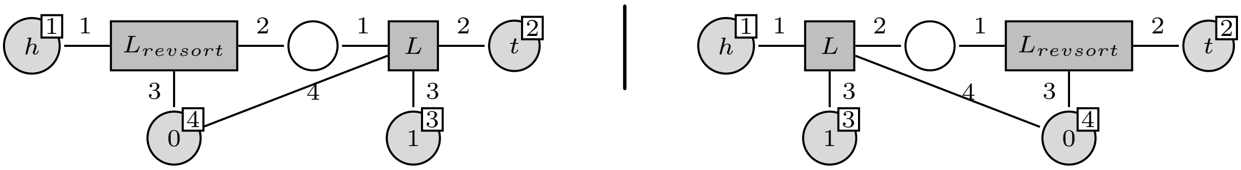



Mixed backward confluence rules :

$L \rightarrow$




$L \rightarrow$




$L \rightarrow$