Automated Formal Verification of Programs with Dynamic Data Structures Using State-of-the-Art Tools

Pavel Erlebach [⋆]
erlebach@fit.vutbr.cz

Tomáš Vojnar[⋆]
vojnar@fit.vutbr.cz

Abstract: This paper investigates capabilities of two advanced state-of-the-art tools—namely Pale and TVLA—for automated formal verification of programs manipulating *unbounded* dynamic data structures. We consider verification of operations dealing with binary search trees over which we want to verify the basic correctness of pointer manipulations (no null pointer dereferences, etc.) as well as the sortedness requirement of binary search trees. Unlike in other works, we want to verify the *full* sortedness requirement, i.e. that all nodes in the left subtree of a node n are smaller than n, and all nodes in the right subtree of n are bigger than n (and not just the relation between a node and its direct subnodes). For the needs of verifying this property in the TVLA tool, we provide a new kind of instrumentation predicate that allows one to handle such a property.

1 Introduction

We address the problem of *automated formal verification* of programs handling dynamic recursive data structures via pointers (or references). Such programs are generally difficult to write and understand, and so the possibility of their formal verification is highly desirable. Formal verification of such programs is, however, a very difficult task too—it is easy to show it is undecidable. This is caused by the fact that such programs work with unbounded structures, which leads to a necessity of dealing with infinite state spaces. To be able to cope with them, we need a suitable finite representation of infinite sets of states, which in the worst case can have a complex graph structure, and (semi-)algorithms working over these structures and capable of returning as precise as possible results and terminating as often as possible.

There have been proposed several (though still not yet fully satisfactory) approaches to formal verification of programs manipulating pointers. One of the least automatic is the use of Separation Logic [7] which is an extension of Hoare logic. PALE [6] works on a similar basis but the verification is more automatic. Only pre- and post-conditions and loop invariants need to be supplied and the rest is done automatically using the MONA decision procedures. There is a more detailed description of PALE in Section 3.

Even more automatic is the approach of the parametric shape analysis implemented in the TVLA tool [8]. The approach is based on an automatic abstraction of memory configurations (denoted as stores) using 3-valued logic wrt. a given set of instrumentation predicates and on an automatic computation of sets of abstract stores reachable at particular program points. We discuss this approach in more detail in Section 4.

An alternative approach [1] is based on abstract regular model checking in which finite automata encode sets of configurations of the given program viewed as words over

[⋆] FIT, Brno University of Technology, Božetěchova 2, CZ-61266, Brno, Czech Republic a suitable alphabet and finite transducers are used to implement the particular program statements. Via a repeated application of the transducers accelerated by abstraction over automata, overapproximations of sets of configurations reachable at particular program points are obtained fully automatically. Unlike TVLA and PALE, the approach is currently usable for dealing with linear (or linearisable) data structures only though its generalizations are in development.

Representations of linked memory structures based (at least partially) on automata are used in [4, 3, 9, 2] too. In [3, 9], the special problem of may-alias analysis is primarily considered, and automata are combined with various classes of constraint systems. In [2], an alias logic with a Hoare-like proof systems is introduced. In [10], the automata part of the encoding is dropped and a semantic model of the execution of a program is used in which every allocated object is identified by a timestamp representing the state of the program at the instant of the object creation. Relations of the timestamps and various program variables are then encoded via automatically derived constraints.

In this paper, we concentrate on the approaches of PALE and TVLA for which a solid tool support is available, which handle even non-linear structures, and are capable of verifying both basic pointer-related properties (no memory leakage, no work with uninitialized pointers, etc.) as well as more complex properties of the considered data structures (e.g., sortedness). To compare these approaches, we choose working with the non-linear structure of binary search trees and we want to verify the basic consistency of pointer manipulation over them as well as—unlike in other works—their *full* sortedness requirement, i.e. that all nodes in the left subtree of a node n are smaller than n, and all nodes in the right subtree of n are bigger than n (and not just the relation between a node and its direct subnodes). To be able to verify the full sortedness property in the TVLA tool, we then provide a new kind of instrumentation predicate that allows one to handle such a property.

The paper is organized as follows. In Section 2, we briefly recall some of the code for dealing with binary search trees. In Sections 3 and 4, we discuss verification of the code in PALE and TVLA. We discuss the results and our future plans in Section 5.

2 Binary search trees

Binary search tree is a binary tree in which each node has a *key*, which is unique. For every node n, all keys in the left (or right) subtree are smaller (or greater) than key of n, respectively. The basic operations include `SearchBST`, `InsertBST`, and `DeleteBST`. The sortedness of the tree is supposed to hold before and after executing each of these procedures. `InsertBST` inserts a new node to a tree or does nothing when a node with the same key is found. `DeleteBST` deletes a given node from a tree. This operation is the most complicated since deleting a non-leaf node causes *rotations* of adjacent nodes. `SearchBST` searches a given node in the tree in a logarithmic time (wrt. the number of nodes).

In this paper, we concentrate mostly on the verification of the `InsertBST` procedure. To recall the way it may be implemented, here is the code of this procedure:

```
Tree { Tree *left;
       Tree *right;
       data d;
};
```

```
InsertBST(Tree *x, Tree *y) {
    Tree *z, *t;

    z = x;
    while (z != NULL) {
        t = z;
        if (z->d == y->d) exit;
        if (z->d > y->d) z = z->left;
        else z = z->right;
    }
    if (t->d > y->d) t->left = y;
    else t->right = y;
}
```

3 Verification in PALE

Pointer Assertion Logic is a notation for expressing pre- and post-conditions of procedures, loop invariants, and other assertions in Weak Monadic Second-order Logic of Graph Types [6]. These annotations are appended to the code of the verified program, and PALE (the Pointer Assertion Logic Engine) analyses it and reports null-pointer dereferences, memory leaks, and violations of assertions and graph type errors. Validity of conditions and other formulae is checked by the MONA tool which can provide explicit counterexamples to invalid formulae.

To make the verification decidable, the technique requires explicit loop and function call invariants to be provided manually, which is a significant disadvantage of PALE compared to TVLA. On the other hand, thanks to the provided information, every statement of the program is analyzed only once, so the verification time is relatively short.

We present here a more abstract and simpler version of the verification—we check whether the new node is inserted properly into the tree regardless of whether it is initially sorted or not. Then, in the PALE notation, the node of the tree may be defined as follows:

```
type Node = { data left,right:Node;
              bool le; }
```

It has a left and right child and an attribute describing the relation of its key to the key of the new node being inserted into the tree—if *le* is true for the node, then this node is less than or equal to the new node).

Now, we write a precondition that describes the initial state of the variables and the memory such that we have a tree pointed to by x and a new node not reachable from x that is pointed to by y. (Below, we denote the node pointed to by a pointer x as $\mathbf{x}$.)

```
[x!=null & y!=null & y.left=null & y.right=null & !x<(left+right)*>y]
```

We add a postcondition which describes the desired property that we want to verify, i.e. that **y** is reachable from the root and was inserted properly. The latter means that all its parents are greater (or less) than **y** if **y** is reachable from the left (or right) child of that parent, respectively.

```
[x<(left+right)*>y & allpos t of Node: t<(left+right)*>y =>
((t.left<(left+right)*>y => t.ge) & (t.right<(left+right)*>y => !t.ge))]
```

Both the precondition and postcondition are quite short and simple. However, since there is a loop within the `InsertBST` procedure, we must add a quite difficult loop invariant. The invariant we were able to come up is approximately five times longer than the pre- and post-conditions and twice longer than the code of the procedure. It contains the precondition and further a requirement of non-reachability of **y** from z and t, a parent-child relation between `t` and `z`, a condition about the end of the descent (if z is `null`, then the left or right descendant of `t` is `null`—according to its relation to **y**—that is the place where **y** comes to), and finally, two conditions that are similar to the postcondition—all parents are greater (or less) than **y** if `z` is reachable from the left (or right) child of that parent, respectively, and the same for `t`.

After supplying all these conditions, the verification passes correctly informing us that there can be no null pointer dereferences, there are no cycles in the tree, all cells are reachable from the data variables, and the postcondition is always satisfied.

4 Verification in TVLA

TVLA (Three-Valued Logic Analyzer) is based on a three-valued first-order predicate logic with transitive closure. It outputs the set of all possible memory states which can appear in every step of the program encoded in the three-valued logic wrt. a given set of predicates. The indefinite value of the logic (i.e. 1/2 compared to the definite values 0 and 1) allows us to represent an arbitrary number of nodes as a single so-called *summary node*. In this way, we may easily finitely encode infinite structures. However, we obtain an overapproximation, and so the output is sometimes not precise enough. For example, for a certain choice of predicates, all singly linked lists with two or more nodes and a variable pointing to the head of the list may be abstracted to one structure with one head node and the rest compressed into one summary node. This problem can be solved by supplying additional predicates.

4.1 The Process of Verification in TVLA

In a simplified view, the analysis can be described as follows. We code the initial memory configuration(s) into values of the chosen predicates and then simulate execution of the program statements by updating these values via predicate transformers (either supplied manually or derived). At the end we have all possible memory configurations at every line of the program.

Before we start the above computation, we must create the predicates which will hold information about the memory. The set of the predicates is partitioned into two disjoint sets. For describing atomic properties of a memory configuration, we use *core predicates* and for describing derived properties, we use *instrumentation predicates*. The sets of these predicates remain nearly the same for procedures working with the same or similar structures. So there is no need to completely re-create them every time. For example, when we have already verified some property for unsorted binary trees and we want to verify a procedure working with binary search trees, we only add predicates that hold information about the sortedness.

Next, the initial memory configuration(s) must be specified. Then, we must specify how this configuration and other configurations change depending on the statements of the program. These rules are called *predicate update formulae*. These formulae must be

Predicate	Intended meaning
$x(v)$	Is variable x pointing to the node v?
$right(v_1, v_2)$	Is the node v_2 right child of the node v_1?
$left(v_1, v_2)$	Is the node v_2 left child of the node v_1?
$dle(v_1, v_2)$	Is the node v_1 less-than-or-equal-to the node v_2?
$downStar(v_1, v_2)$	Is the node v_2 child of the node v_1? $downStar(v_1, v_2) = (left + right)^*(v_1, v_2)$
$r[x](v)$	Is node v reachable from variable x? $r[x](v) = \exists v_1 : x(v_1) \wedge downStar(v_1, v)$
$sorted(v)$	Are all the left and right children of node v less and greater than v, respectively? $sorted(v) = \forall v_1, v_2 : (left(v, v_1) \wedge downStar(v_1, v_2) \rightarrow dle(v_2, v)) \wedge (right(v, v_1) \wedge downStar(v_1, v_2) \rightarrow dle(v, v_2))$
$allSorted$	Are all nodes sorted? (this is the desired property) $allSorted = \forall v_1 : sorted(v_1)$

Tab. 1: Simplified predicates used in verification of search trees.

written manually for core predicates while for instrumentation predicates, they can be generated automatically from the base definition of these predicates.

4.2 Verification of Binary Search Trees in TVLA

For our procedure working with search trees, we may use the instrumentation predicates devoted to working with trees published in the literature on TVLA—they are shown in Table 4.2 together with the predicates *sorted* and *allSorted* that we introduce for specifying the full sortedness property. The first four predicates are core predicates and the other four are instrumentation predicates.

However, these formulae are not enough for verifying the full sortedness property for `InsertBST`. The result is always $allSorted = 1/2$ (i.e. the unknown value). This is caused by the fact that as the pointer z descends from the root to the leaves, we forget which nodes it has gone through and which not. One of the possible abstract memory configurations at the end of this descent is depicted in Figure 1. The notation is as follows: Nodes are ellipses, summary nodes are double ellipses, unary predicates are satisfied for a node if the name of the predicate is written in it, otherwise they evaluate to 0. Binary predicates are depicted as full line arrows when they evaluate to *one*, dotted arrows when they evaluate to 1/2, and finally when they evaluate to 0, no arrows are drawn. For simplicity and transparency, all the *dle* predicates are omitted.

After adding a new node, nothing is known about the sortedness of the tree. To solve this problem, we have to create an instrumentation predicate which will keep the information about the descent. The possible need to devise such predicates is a disadvantage of using TVLA. It may be quite time-consuming to think out a helpful predicate and it may require a lot of insight. Very recently there appeared some new results [5] about the possibility of automatically generating some instrumentation predicates. In the future, it will be interesting to examine if these new results are able to generate the special new kind of predicate for verifying full sortedness that we describe below. Currently these methods are not implemented in the available version of TVLA yet.

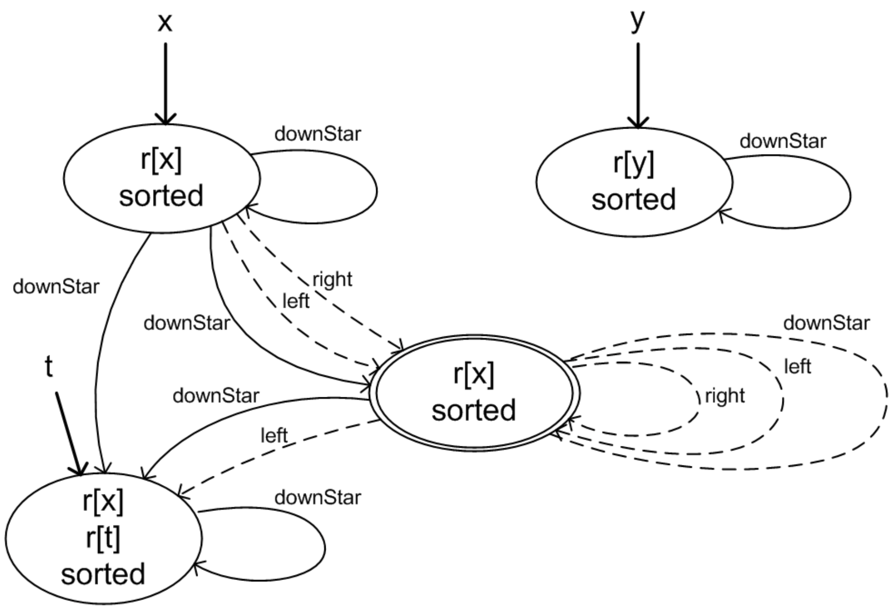

Fig. 1: An example of a memory configuration after executing the loop of `InsertBST`

4.3 A New Instrumentation Predicate

The new predicate we provide for solving the above problem is defined as follows:

$$i[x,y](v) = \exists v_1, v_2, v_3 : x(v_1) \land y(v_2) \land downStar(v_3, v_1) \land \\ (right(v, v_3) \land dle(v, v_2) \lor left(v, v_3) \land dle(v_2, v))$$

Its meaning can be described as *if i[x,y] evaluates to* 1 *for a node v then "adding y after x" does not harm sortedness of the node v* where adding y after x means adding the node which is pointed to by y after the node which is pointed to by x (i.e. executing `x->left = y` or `x->right = y`). The predicate can be described in another way too: *if x points to the node which is the left (right) descendant of v, then v must be greater (less) than the node pointed to by y, respectively.*

The next thing is to create predicate update formulae for the new predicate. The generated ones do not work in this case possibly due to the transitive operation hidden in *downStar*. So we have to create them manually which is relatively time-consuming. Then we have to modify (or create) predicate update formulae for the predicate *sorted* in statements `x->left|right = y` for our effort to take effect. It is enough to add "`|i[x,y]`" to the formulae.

Now we can check whether the new predicate solves our sortedness problem. Before the insertion of the new node into the tree, the situation is much better as depicted in Figure 2. The tree is divided into two disjoint parts: nodes which the pointers t and z went through (those with $i[t,y]$) and the untouched ones. As the pointer t points to a node whose all ancestors fulfil $i[t,y]$, the insertion passes without harming the sortedness.

For a better understanding see Figure 3 where a not abstracted tree is shown. The node pointed to by x (the root) and the node pointed to by t in Figure 3 are not abstracted in Figure 2. The path the pointers z and t went through is marked with `p` in Figure 3 and these nodes correspond to the summary node which satisfies $i[t,y]$ in Figure 2. All the other nodes in Figure 3 are compressed to the summary node in the middle of Figure 2.

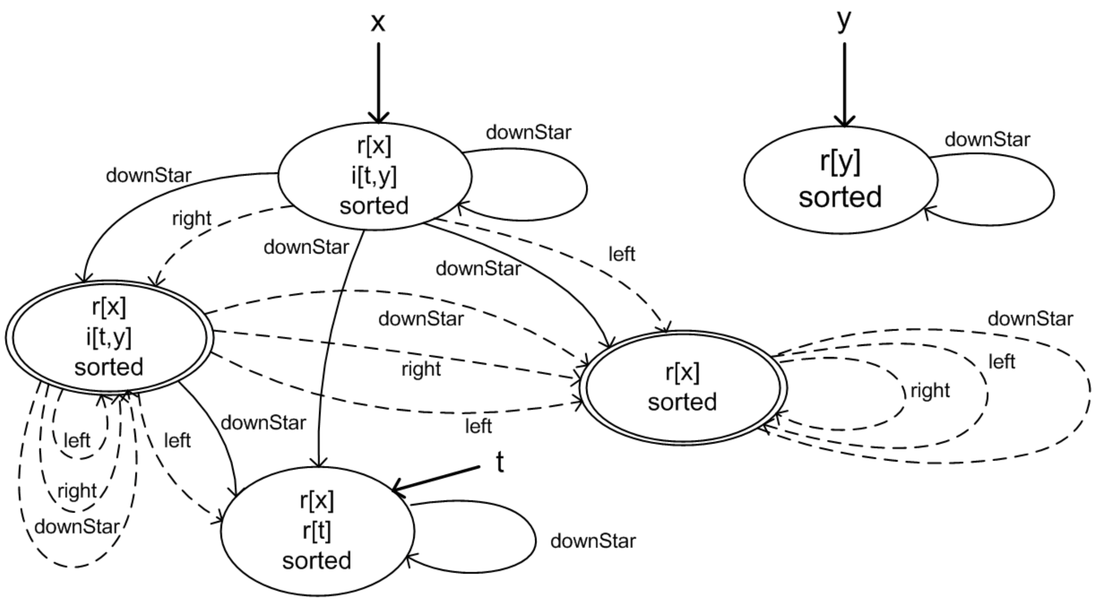

Fig. 2: One of the memory configurations with the new predicate after executing the loop

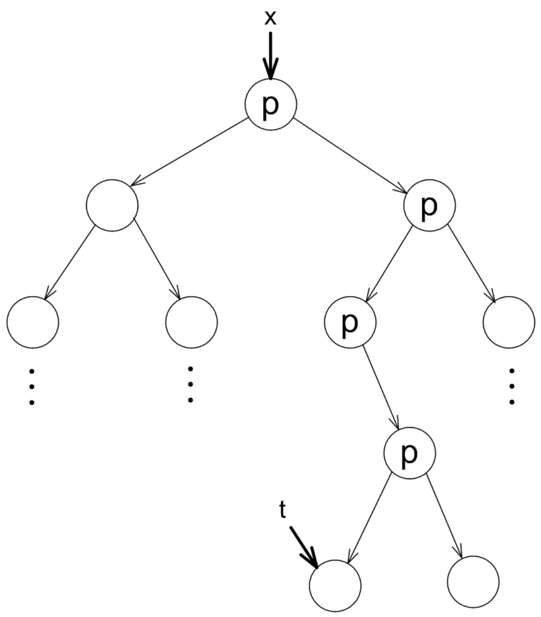

Fig. 3: A sample concrete tree covered by the abstract configuration of Fig. 2

5 Conclusion

We discussed formal verification of the unbounded dynamic data structure of binary search trees using two state-of-the-art tools, namely TVLA and PALE. Unlike in other works, we considered even verification of the full sortedness property of binary search trees. In the paper, we concentrated on the `InsertBST` procedure. The case of searching in binary search trees is simpler because it does not perform any destructive updating of the structure. On the other hand, the case of `DeleteBST` is much more difficult—e.g., as discussed below, our verification of `DeleteBST` was too long to complete. In order to successfully verify the full sortedness property in the TVLA tool, we have provided a new specialised kind of instrumentation predicate $i[x,y](v)$.

The biggest differences between PALE and TVLA witnessed in our experiments are in the amount of information to be provided manually and in the verification times. In PALE, one must almost always manually provide a significant amount of help to the tool, but the verification times may be relatively low. In TVLA, the amount of help to the tool is usually much lower, but the running times may be higher—the verification of `InsertBST` takes about 2 seconds in PALE and 2 hours in TVLA on Athlon 1.66GHz. Sometimes, even for TVLA, one may have to provide manually some help in the form of special instrumentation predicates such as our $i[x,y](v)$ predicate. The first steps towards relieving this problem are presented in [5].

As we have already mentioned, we were not able to successfully complete verification of the `DeleteBST` procedure in TVLA, which may be due to a very high number of structures (there are 8 pointers and the number of their possible relations in the structure is reflected in the computing time). For the future, we would like to more carefully examine sources of this problem and its possible solutions including a possibility of a parallelization of the method. Further, it is interesting to try to verify procedures working with even more complex structures like, e.g., B-trees. Finally, the question of verifying liveness properties over procedures manipulating dynamic data structures is another challenging issue.

Acknowledgement. *The work was supported by the Grant Agency of the Czech Republic under the contracts 102/04/0780 and 102/03/D211.*

Bibliography

1. A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In *Proc. of TACAS'05*, LNCS. Springer, 2005. To Appear.
2. M. Bozga, R. Iosif, and Y. Laknech. Storeless Semantics and Alias Logic. In *Proc. of PEPM'03*. ACM Press, 2003.
3. A. Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-Limiting. In *Proc. of PLDI'94*. ACM Press, 1994.
4. H.B.M. Jonkers. Abstract Storage Structures. In *Algorithmic Languages*. IFIP, 1981.
5. A. Loginov, T. Reps, and M. Sagiv. Learning Abstractions for Verifying Data-Structure Properties. Technical Report TR-1519, Computer Sciences Department, University of Wisconsin, Madison, WI, USA, 2005.
6. A. Møller and M.I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI'01*, 2001. Also in SIGPLAN Notices 36(5), 2001.
7. J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. IEEE Computer Society, 2002.
8. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *TOPLAS*, 24(3), 2002.
9. A. Venet. Automatic Analysis of Pointer Aliasing for Untyped Programs. *Science of Computer Programming*, 35(2), 1999.
10. A. Venet. Nonuniform Alias Analysis of Recursive Data Structures and Arrays. In *Proc. of SAS'02*, LNCS. Springer, 2002.