EXPERIENCE FROM VERIFYING IN TVLA

Ing. Pavel ERLEBACH, Doctoral Degree Programme (2)
Dept. of Intelligent Systems, FIT, BUT
E-mail: erlebach@fit.vutbr.cz

Supervised by: Prof. Milan Češka, Ing. Tomáš Vojnar, Ph.D.

ABSTRACT

This paper investigates capabilities of the TVLA, which is one of the most promising tools for automated formal verification of programs manipulating unbounded dynamic data structures. We chose two areas to be verified with this tool and we hope that we will learn as much as possible from problems arising in the verification process. In the future we would like to eliminate some of these problems.

The first set of examples contains the operations with binary sorted trees. After executing these procedures we verify if the tree is sorted after the insertion and if all the pointer manipulations were correct (no memory leakage, null pointer dereferences, etc.). The second example is the Bakery algorithm which is one of the most famous mutual exclusion algorithms. We want to verify safety of this algorithm.

1 INTRODUCTION

We address the problem of *automated formal verification* of programs handling dynamic recursive data structures via pointers (or references). Formal verification of such programs is a very difficult task—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. We show only a short digest. One of the least automatic is the use of Separation Logic [1] which is an extension of Hoare logic. PALE [2] works on a similar basis but the verification is more automatic. Even more automatic is the approach of the parametric shape analysis implemented in the TVLA tool [3]. That is the method we aim to at this paper. 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.

The paper is organized as follows. In Section 2, we describe the examples which we have chosen for analysis of TVLA. In Section 3 and 4 we discuss verification of operations of the binary sorted tree and the Bakery algorithm. We discuss the results and our future plans in Section 5.

2 EXAMINED EXAMPLES

For effective analysis of capabilities of TVLA we aim at nontrivial algorithms which may cause some problems when verified. From these problems in the verification process of such algorithms we will learn much more than from verifying of already solved problems. In TVLA exist many enclosed examples which verify procedures working with linear linked lists, so we choose from algorithms working with nonlinear data structure.

As the representatives of the algorithms working with complex data structure we chose the operations with binary sorted trees, i.e. the `InsertBST`, `SearchBST` and `DeleteBST` procedures. After executing each procedure we want to verify that the tree remains sorted, that all pointer manipulations were valid and that the procedure works as expected (e.g. in `InsertBST` the new node was inserted).

One enclosed example does the same thing as `InsertBST`, but it checks only *partial* sortedness, i.e. that left (or right) child of the node n is smaller (or bigger) than n respectivelly. We propose checking of the *full* sortedness, i.e. all nodes in the left (or right) **subtree** of the node n are smaller (or bigger) than n.

The algorithm mentioned above works with data structures that have predefined format (binary sorted trees). For further analysis we choose the Bakery algorithm [5] which is one of the most famous mutual exclusion algorithms. It works for unbounded number of processes without any special atomic operations. It consists in taking the *ticket* by processes which want to enter critical section and then resolving which process may enter. This will lead to complex memory states where generally graphs without any predefined structure may appear. Moreover, we will test the ability of TVLA (or more precisely TVMC—extension of TVLA for concurrent systems) to verify parallel programs.

To recall the way the `InsertBST` (which is analysed in this paper more in detail) may be implemented and the core of the Bakery algorithm, here are the codes, whereas $(a,b) < (c,d)$ is an abbreviation for the condition $(a < c) \,||\, (a == c \;\&\&\; b < d)$.

```
InsertBST(Tree *x, Tree *y) {     | /* Bakery algorithm */
  Tree *z, *t;                    | choosing[i] = true;
  z = x;                          | num[i] = max(num[0]..num[n-1])+1;
  while (z != NULL) {             | choosing[i] = false;
    t = z;                        | for (j=0; j<n; j++) {
    if (z->d == y->d) exit;       |   while (choosing[j]) {}
    if (z->d > y->d) z = z->left; |   while ((num[j] != 0) &&
    else z = z->right;            |    (num[j],j) < (num[i],i)) {}
  }                               | }
  if (t->d > y->d) t->left = y;   | /* Critical section */
  else t->right = y;              | num[i] = 0;
}                                 | /* Noncritical section */
```

3 VERIFICATION OF THE OPERATIONS OF BINARY SORTED TREES

In case of the `SearchBST` procedure the verification is trivial since the tree is not modified within the procedure. In case of the `InsertBST` procedure we found out that the supplied information is not sufficient which means that the verification always results in $1/2$ (unknown value). It is caused by the fact that as the pointer *z* descends from the root to the leaves, then the information about the path of this pointer is lost. It is the effect of the abstraction to the three valued logic.

To solve this problem we have to add one new instumentation predicate $i[x,y](v)$. This predicate can keep the information about the path of the pointer *z*. Its definition is:

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

Its meaning can be described as: the $i[x,y](v)$ holds if "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*. Thanks to this the information about the path remain achieved and after adding a new node we can say for certain, whether the modified tree is sorted or unsorted.

In case of `DeleteBST` a complicated situation arose. It was caused by the fact that deleting a non-leaf node results in nontrivial rotations of the tree. It was difficult to verify because in the program there were 8 active pointers at the same time which led to the state explosion. The number of pointers is partally caused by the inability of TVLA to perform statements like `if (x->next == NULL)`. More precisely, it is able to perform such statements but with zero effect. Being short, when executing this statements, some memory states satisfy this condition and some not. But then comes to place the blur phase which joins some of satisfying states with the other ones and the resulting state of memory is the same as at the beginning. So when one wants to perform `x = x->next` after the condition mentioned above it may result in false alarms. This has to be solved by adding auxiliary pointer.

4 ANALYSIS OF BAKERY ALGORITHM

There have already been some successful attempts to verify Bakery algorithm [4] but in that case it was verified manually. We hope that in TVLA the verification will be more automated.

Bakery algorithm uses three data types: bool (*choosing*), int (*num*) and constant (process ID). To make verification possible we must represent these types suitable. One of the possible representations is as follows: Bool needs no special representation, it is represented by a unary predicate. For representing IDs of the processes we can line up the processes into single linked list and define that the position of the process in the list mirrors the value of the ID, e.g. the first process has the smallest ID, the last process the greatest ID, etc. For representing local integers we need the auxiliary single linked list to which the processes point. Process which points to the first node has the smallest *num*, etc.

Unfortunately, after creating the code the verification resulted in some false alarms. They are caused by the fact that the information about the order of both processes and

nodes in the list is completely lost. In the program we have unbounded count of the processes pointing completely randomly to the unbounded count of the nodes (and processes). For precise representation of the relations between processes and nodes one would need unbounded memory and time. The abstraction degrades this configuration to one summary node which is pointed to by all processes. Moreover, we did not succeed to find the instrumentation predicate which would solve this property. For make it clear, we desribe the differences from the verification of the `InsertBST` procedure. Even there it was possible that in some states the whole tree (except the root) was reduced to one summary node but as the pointer descended down to the leaves, we exactly knew the structure of the tree, i.e. every node has up to two children, left child is smaller than its parent, etc. In the case of Bakery algorithm we know almost nothing when this summary node appears, i.e. which process was at the beginning of the cycle of tests, which was just before the entering to the critical section, etc.

In the verification of the original Bakery algorithm we did not succeed, so we come up to some simplifications. Instead of pointers we will have only two unary predicates $min(t)$ and $min\_p(t)$. The $min(t)$ predicate holds if the process t has the smallest ticket *num*. The $min\_p(t)$ predicate holds if $min(p)$ holds and the process t has the smallest ID of all the processes u which satisfy $min(u)$. The motivation is that we are not interested in the information about the process with the smallest ID (e.g. this process may never ask for entering the critical section) but we want to know which process of the ones with minimal ticket has the smallest ID—it is the promising adept for entering the critical section.

Only the information about the minimal ticket and minimal ID remains, so it is necessary to simplify the verified algorithm. It is not possible to simulate cycles with such predicates so we have to create a simpler version of the algorithm where the tests in cycles are atomic, i.e. testing of *choosing* variable and the testing of *num* variable and ID of the process. It is quite radical modification and the eventual verification would not be so valuable. The Bakery algorithm is distinctive thanks to the fact that it works without any special atomic operation. On the other hand non-atomicity of assigning the maximum (at the second line of the code) was preserved after adding next predicate *notmin*.

After this simplification the verification results in correct answers but the computation time lasts roughly days. Even the code has only 8 lines and 3 predicates, it is enough to the state explosion. After some minor optimization the verification ends in minutes. However, it means additional work required from the user.

The variant of the Bakery algorithm, where the seemingly unnecessary variable *choosing* is missing, is frequently mentioned as the “false Bakery algorithm”. This variant does not ensure the mutual exclusion—for 3 or more processes it can happen that two processes enter the critical section. We tried to verify this variant also and the result was correct, so the verification showed that the algorithm is not safe.

5 CONCLUSION

We succeeded in the verification of the `SearchBST` and `InsertBST` procedures with the full sortedness property. However, we had to supply relatively big amount of additional data despite the fact that TVLA is classified as one of the most automatic tools for formal verification of programs which perform destructive updating. Aside from the initial

state and predicate update formulae (which define changes in predicates while executing statements of the program) we had to supply the new instrumentation predicate. It needs quite insight to create a helpful one. For this predicate we had to create nontrivial predicate update formulae and finally the verification resulted in the correct answers.

In the case of the `DeleteBST` procedure the state explosion arose which was caused by a great number of active pointers at the same time in the procedure. The count of pointers had to be increased due to the inability of TVLA to perform statements like `if (x->next == NULL)` which led to need of auxiliary ponters.

In the case of the Bakery algorithm the verification resulted in false alarms due to the loss of information which we were not able to reduce. However, we were able to verify the simplified version where the cycles with tests were replaced by atomic operations. First, the computation took about days, but after some optimizations it was reduced to minutes. Nonetheless, even the verification of the simpler version of the Bakery algorithm needed quite big amount of information manually supplied. We were also able to detect violation of safety while verifying the "false Bakery algorithm" where *choosing* variables are missing.

These problems in the verification gave us valuable clues and in the future we plan to aim at following tasks namelly. In the verification of both the `DeleteBST` procedure and Bakery algorithm we had problems with the state explosion, so it would be very interesting to examine possibilities of reduction of the state space, e.g. the partial order method and symmetry reduction method seem to be very promising. These methods were developed in the context of finite state space, so we will have to adjust them to work in the context of unbounded state space.

Other task, we are interested in, is the inability of TVLA to verify the original Bakery algorithm. The reason is that the memory states contain very complicated net of pointers which are pointing to each other entirelly randomly and the abstraction causes a drastic loss of information. By the verification of the `DeleteBST` procedure we learned an interesting property of TVLA — inability to perform statements like `if (x->next == NULL)`. Last but not least, it would be appealing to examine the possibility of the parallel or distributed computation which would also make the computation time shorter.

We would like to aim at improving all of these properties of the verification tool preferably in self made tool where it would be easy to practise the experiments.

REFERENCES

[1] Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In Proc. of LICS'02. IEEE Computer Society, 2002.

[2] Møller, A., Schwartzbach, M.I.: The Pointer Assertion Logic Engine. In Proc. of PLDI'01, 2001. Also in SIGPLAN Notices 36(5), 2001.

[3] Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric Shape Analysis via 3-Valued Logic. TOPLAS, 24(3), 2002.

[4] McMillan, K., Qadeer, S., Saxe, J.: Induction in Compositional Model Checking. In CAV 2000, Eds. LNCS 1855.

[5] Lamport, L.: A New Solution of Dijkstra's Concurrent Programming Problem. Commun. ACM, 17(8):453–455, 1974.