Depth Bounded Explicit-State Model Checking

Abhishek Udupa[1], Ankush Desai[2], and Sriram Rajamani[2]

[1] University of Pennsylvania
[2] Microsoft Research India

Abstract. We present algorithms to efficiently bound the depth of the state spaces explored by explicit-state model checkers. Given a parameter k, our algorithms guarantee finding any violation of an invariant that is witnessed using a counterexample of length k or less from the initial state. Though depth bounding is natural with breadth-first search, explicit-state model checkers are unable to use breadth first search due to prohibitive space requirements, and use depth-first search to explore large state spaces. Thus, we explore efficient ways to perform depth bounding with depth-first search. We prove our algorithms sound (in the sense that they explore exactly all the states reachable within a depth bound), and show their effectiveness on large real-life models from Microsoft's product groups.

1 Introduction

Though several strategies to mitigation the state explosion problem in model checking have been studied extensively (such as symbolic model checking [4],partial order reduction [8,15], symmetry reduction [18], automated abstraction-refinement [6,3,11]), model checkers are still unable to cope with the state spaces of very large models. To cope with very large state spaces, bounding techniques have been proposed to systematically explore a part of the state space. The key idea with bounding based approaches is given by the "small-scope hypothesis"(see, for instance [14]), which states that if a model is buggy then the bug will most likely manifest by exploring all states systematically after bounding a parameter of the model(such as input size, number of processors, number of context switches).

In this paper, we discuss a new algorithm for depth bounding —that is, exploring all states of a model that are reachable within a given depth from the initial states— with iterative deepening of the depth bound. Depth bounding is trivial in symbolic model checking [4], since symbolic model checking is naturally done in a breadth-first manner. If a depth bound d is fixed *a-priori*, then symbolic exploration of all states which are reachable within d steps from the initial states can be reduced to SAT directly, thereby avoiding expensive existential quantification operations during symbolic model checking. This technique is called Bounded Model Checking(BMC) [5], and has been studied widely. In contrast, even though depth bounding has been implemented in popular explicit-state model checkers such as SPIN [12], we find that algorithmic improvements

A. Groce and M. Musuvathi (Eds.): SPIN 2011, LNCS 6823, pp. 57–74, 2011.

are still possible to improve efficiency. This paper contains two new techniques: 1) depth threshold, and 2) frontier tree, to improve the efficiency of depth bounding in explicit-state model checking. Before we describe these techniques, we first motivate why it is nontrivial to bound depth in explicit-state model checking.

Depth bounding in explicit-state model checking. The obvious way to bound depth in explicit-state model checking is to use breadth-first search (BFS). However, as we explain below, breadth-first search is infeasible in explicit-state model checking due to excessive memory consumption. Suppose we have a single initial state, and each state consumes M bytes of memory. Let F_k be the set of states whose shortest path from the initial state is of length k. If we perform breadth-first search, then we first explore all the states in F_1, then all the states in F_2, F_3, etc in stages. At stage k, storage for the set of states F_k (which are called the "frontier states") consumes $\mid F_k \mid \times M$ bytes. Since $\mid F_k \mid$ is exponential in k, and M is of the order of hundreds of kilobytes, $\mid F_k \mid \times M$ explodes for large k. In contrast with BFS, for depth-first search(DFS), only the states on the DFS stack need to be stored in full. For visited states that are not on the DFS stack, only fingerprints or bit-state hashes [13,19] need to be stored to avoid revisiting states. Even within the DFS stack, only the top most state needs to be stored in full —each of the remaining states s can be stored in terms of incremental difference or undo log from the state above s in the DFS stack. Thus, most explicit-state model checkers use DFS instead of BFS in order to scale to large models.

In such a setting (DFS based explicit-state model checking), implementing depth bounding efficiently and correctly is non-trivial. The obvious way to bound depth is to simply stop exploring a state either if the current depth exceeds the depth bound, or if the state has been visited earlier. However, as we show in Section 3, this algorithm is incorrect, since the same state can be visited at different depths, and can lead to missing states that can be explored within the given depth bound. Alternatively, we can record the depth of each state in the state table, and re-explore a state if the current exploration depth is lesser than the previous exploration depth. This is a correct algorithm, and is indeed guaranteed to explore all states that are reachable within a depth bound. However, as our empirical data shows, this results in the same state being explored several times with different depths and the algorithm is very inefficient. In this paper, we describe a new algorithm that maintains a *threshold* value for each state. Intuitively, the threshold value for a state is the maximum depth at which the state needs to be revisited so that there is a possibility of exploring a previously unexplored state within the current depth bound. We show how we can compute thresholds and use thresholds to avoid revisiting states.

Iterative Depth Bounding. Since it is hard to pick a good depth bound *a-priori*, depth bounding works best if we can iteratively increment the bound and explore as much depth of the state space as we can, within our time and space budget. That is, we start with a depth bound d, and first explore all the states that are reachable using paths of length d or less from the initial state. Then, we increment the depth bound to $2d$, $3d$, ..., and keep exploring states that are reachable at these larger depths as much as our time and space budgets permit.

Such an iterative depth bounded search combines elements of both DFS and BFS —within each depth bound, we do DFS, and increasing the depth bound essentially amounts to doing BFS at the granularity of the depth increment d. In order to save space for storing the frontier at each of the depth bounds $d, 2d, 3d, \ldots$ in iterative depth bounded search, we represent frontier states using traces (a trace of a state S, is sequence of edge indices along a path from the initial state to S) and a full state is produced on demand by replaying the trace representing the state. Given a trace of length nd producing a full state by replaying takes time $O(nd)$, and the replay overhead becomes large for large values of nd. We propose a data structure called *frontier tree* to reduce the replay overhead to $O(2d)$ during iterative depth bounding.

The main motivation for our work was demand from our users (Microsoft product groups) to explore state spaces of very large models to as large depths as possible. In particular, the Universal Serial Bus (USB) team in the Windows product group found and fixed over 300 design bugs using our model checker, including some bugs that manifest only at very large depths in the state space. Consequently, they wanted to cover every state within as large a depth bound as possible, within a fixed time and memory budget, to get confidence in the correctness of their design. Our algorithms have helped them improve the depth up to which they can cover all states up to 86.8%, and improve the number of states explored up to 1246.8% on one of their large models for a time budget of 5 hours and 30 minutes, and a memory budget of 1200MB. Our efforts have resulted in our model checker being used day-to-day in a production setting. The Windows group uses our depth bounded model checker as key component in design validation.

2 Background

In this section, we give some background about how explicit-state model checkers work.

We assume the existence of the following data-types. A *State* data-type is used to represent states of the system we want to explore. It has the following members: (1) the property fp returns the finger print of the state, (2) the property Depth returns the depth at which the state has been encountered. *Set* is a generic data-type, which supports three methods: (1) the Add method takes an object and adds it to the set, and (2) the $\mathsf{Contains}$ method returns true if the object passed as parameter is present in the set, and false otherwise, and (3) the Remove method removes the object passed as parameter if that object is present in the set. We instantiate *Set* with fingerprints of states in this section. In later sections we also instantiate *Set* with states to represent frontier sets.

The fingerprints of states have the property that identical states are guaranteed to have identical fingerprints. That is:

$$\forall \mathsf{S}_1, \mathsf{S}_2 \in \mathit{State}. \mathsf{S}_1 = \mathsf{S}_2 \Rightarrow \mathsf{S}_1.\mathsf{fp} = \mathsf{S}_2.\mathsf{fp}$$

```
Set⟨Fingerprint⟩ DoneStates

void doDfs(State currentState){
if not DoneStates.Contains(currentState.fp) then
    DoneStates.Add(currentState.fp)
    for all successors S of currentState do
        doDfs(S)
    end for
end if
}

void DFS(State initialState) {
DoneStates= {}
doDfs(initialState);
}
```

Fig. 1. Simple Explicit-State DFS algorithm

Though the converse of the above implication does not hold, the probability of two different states having the same fingerprint can be made extremely low (see [13,19]).

Figure 1 shows the simple DFS algorithm implemented by most explicit-state model checkers. The fingerprints of all explored states is stored in the set DoneStates. The core of the algorithm is the recursive method doDfs, which is called with the initial state. It works by checking if the current state is already in the set DoneStates, and if not, adds it to DoneStates, and invokes itself on all its successors.

DFS is a space efficient algorithm for explicit-state model checking, since we need to store only fingerprints for explored states. Only states that are on the DFS stack need to be represented in memory as full states. A further optimization is possible— we only need to store the top of the DFS stack as a full state. For every state S that is inside the DFS stack, we can represent S using its difference from the state T that is above S in the DFS stack. This technique is called "state delta" and is routinely used in explicit-state model checkers (see, for instance [2]).

In the next section, we use another generic data-type *Hashtable*, and instantiate it with fingerprints as keys and integers as values. *Hashtable* supports the following methods: (1) the Add method, which adds a new key-value pair to the table. (2) the Contains method, which returns true if the specified key is in the hash table. (3) the Update method, which takes as input a key-value pair and updates the table with the new value if the key is already present and adds the key-value pair to the table otherwise.

In later sections, we show how to systematically bound the depth of explored states in DFS, without missing any states. For the purposes of soundness proofs of our algorithms, we assume that fingerprints are not lossy. That is,

$$\forall \mathsf{S}_1, \mathsf{S}_2 \in State. \mathsf{S}_1 = \mathsf{S}_2 \Leftrightarrow \mathsf{S}_1.\mathsf{fp} = \mathsf{S}_2.\mathsf{fp}$$

This assumption allows us to separate soundness concerns about our algorithms from soundness concerns about fingerprints.

```
1: bool IterBoundedDfs(State initialState, int depthCutoff, int inc) {
2: initialState.Depth= 0
3: Set⟨State⟩ frontier = {initialState}
4: Set⟨State⟩ newFrontier= {}
5: int currentBound = inc
6: while currentBound ≤ depthCutoff do
7:    newFrontier = BoundedDfsFromFrontier(frontier, currentBound)
8:    if newFrontier = {} then
9:       return(true)
10:   else
11:      currentBound = currentBound + inc
12:      frontier = newFrontier
13:   end if
14: end while
15: return(false)
16: }
17:
18: Set⟨State⟩ outFrontier
19: /* outFrontier is a global variable which gets updated inside BoundedDfs*/
20: BoundedDfsFromFrontier(Set⟨State⟩ frontier, int currentBound) {
21: outFrontier= { }
22: for all F ∈ frontier do
23:    BoundedDfs(F, currentBound)
24: end for
25: return(outFrontier)
26:
```

Fig. 2. Iterative Depth Bounded Search Algorithm

3 Depth Bounding: Warmup

The state spaces of real-world systems are too large to be completely explored, and in such circumstances, it is desirable to systematically explore all states within a given depth bound under the small scope hypothesis [14]. To achieve this goal, we modify the simple DFS algorithm given in Section 2 to visit a state *if and only if* it is reachable within d steps from the initial state, and iteratively increasing d. In particular, we perform two attempts —the first one is unsound, and the second one is sound but inefficient. These are intended as warm-up exercises before we present our efficient depth bounding techniques in the next section.

Figure 2 gives the outer loop for iterative depth bounded search. The IterBoundedDfs method takes three arguments: (1) initialState, which is the initial state of the model, (2) depthCutoff, which is the depth cutoff bound for the search and (3) inc, which is the amount by which the depth bound is increased in each iteration. We assume that depthCutoff > 0, inc > 0, and that depthCutoff is divisible by inc.

The method IterBoundedDfs repeatedly calls the BoundedDfsFromFrontier method (line 7) in the while loop from lines 6–14. If all states in the model are reachable within depthCutoff, then IterBoundedDfs returns true, otherwise, it returns false.

The BoundedDfsFromFrontier method takes the current frontier set frontier and a depth bound currentBound as parameters, explores all the states starting from the current frontier set frontier that are reachable within currentBound more steps, and returns a new set of frontiers newFrontier.

```
1: Set⟨Fingerprint⟩ DoneStates
2: /* initialized to null-set once in the beginning */
3:
4: Set⟨State⟩ outFrontier
5: /* initialized to null-set in BoundedDfsFromFrontier*/
6:
7: void BoundedDfs(State currentState, int depthBound) {
8: if ¬ DoneStates.Contains(currentState.fp) then
9:    if currentState.Depth < depthBound then
10:        DoneStates.Add(currentState.fp)
11:        for all successors S of currentState do
12:           S.Depth = currentState.Depth + 1
13:           BoundedDfs(S, depthBound)
14:        end for
15:    else
16:        outFrontier.Add(currentState)
17:    end if
18: end if
19: }
```

Fig. 3. Naïve Unsound Depth Bounded DFS algorithm

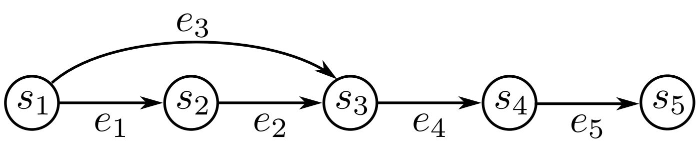

Fig. 4. Example where the algorithm shown in Figure 3 does not cover all reachable states

The implementation of BoundedDfsFromFrontier is shown in lines 20–25. It calls the BoundedDfs function for each state in the frontier set. The BoundedDfs function thus takes a single state and a depth bound and explores all the states that are reachable within the depth bound. States that are reached exactly at the depth bound are added by BoundedDfs to the global set outFrontier to be explored further in the next call to BoundedDfsFromFrontier. The design of BoundedDfs is a deceptively simple problem at the outset, but one that is tricky, if our goal is to be both efficient and correct.

Naïve Unsound Depth Bounded DFS. To give the reader an appreciation for the difficulty in designing BoundedDfs efficiently and correctly, we present our first attempt in Figure 3. We refer to this approach as *Naïve Unsound DBDFS.* Recall that the goal of the BoundedDfs() algorithm is to explore all the states that are reachable within the bound depthBound starting from the input parameter currentState. This algorithm is similar to Figure 1, except that a state is explored only if it is encountered at a depth *less than* the current depth bound (see the conditional at line 9 of Figure 3). If not, then it is added to outFrontier (line 16) to be explored in the next depth bounded iteration.

The algorithm in Figure 3 is incorrect in the sense that it may not explore all the states that are reachable within the given bound depthBound. For instance if a state S is reached initially with a depth of d and later with a depth $d' < d$, the algorithm does not explore the state S, the second time around, which could

```
Hashtable⟨Fingerprint, int⟩ DoneStates
/* initialized to null-set once in the beginning */

Set⟨State⟩ outFrontier
/* initialized to null-set in BoundedDfsFromFrontier*/

bool mustExplore(State S) {
if DoneStates.Contains(S.fp) then
   if DoneStates.Lookup(S.fp) ≤ S.Depth then
      return(false)
   end if
end if
return(true)
}

void BoundedDfs(State currentState, int depthBound) {
if mustExplore(currentState) then
   if currentState.Depth < depthBound then
      DoneStates.Update(currentState.fp, currentState.Depth)
      outFrontier.Remove(currentState)
      for all successors S of currentState do
         S.Depth = currentState.Depth + 1
         BoundedDfs(S, depthBound)
      end for
   else
      outFrontier.Add(currentState)
   end if
end if
return
```

Fig. 5. Naïve Sound Depth Bounded DFS

lead to not exploring some states, although these states are reachable within the given depth bound. For instance, consider the state space shown in Figure 4. If the algorithm is run with a depth-bound of 3, with s_1 as the initial state, and if e_1, e_2 and e_4 are traversed, adding s_1.fp, s_2.fp and s_3.fp to DoneStates. At this point, the algorithm determines that s_4 is at the depth cut-off and adds it to the frontier set. When the recursion unwinds to the state s_1, it does not explore the state s_3, since $s_3.\mathsf{fp} \in$ DoneStates already. Thus, the algorithm misses the state s_5, even though s_5 is reachable within three steps (recall that our depth bound is 3) from s_1 via $e_3 - e_4 - e_5$.

Naïve Sound Depth Bounded DFS. Figure 5 shows our second attempt, which we refer to as *Naïve Sound DBDFS*, where we fix the issue of missing states, by tracking the minimum depth at which a state has been reached so far. That is, we use a hash table DoneStates (rather than a set) to store fingerprints of visited states. For each visited state S, the hash table DoneStates maps the fingerprint of S to the minimum depth the state has been reached so far. When a state S is re-visited, the mustExplore method compares the current depth S.Depth with the smallest depth at which S has been encountered so far (which is stored in DoneStates). If the current depth is smaller, then the state is re-explored with the (smaller) depth and the DoneStates hash table is updated to reflect this. All the states that are precisely at the depth bound are added to outFrontier. The declaration of outFrontier, and the body of the BoundedDfsFromFrontier functions are the same as before.

Note that a state that is added to outFrontier at line 26 may indeed later be found to have a shorter path to it. Consequently, in line 20, we invoke outFrontier.Remove for currentState since currentState is currently visited with depth less than the given depth bound, and may have been added to outFrontier earlier.

Below, we state lemmas and a theorem to prove that the algorithm in Figure 5 is correct in the sense that it explores exactly all the states that are reachable from the input frontier set within the depth bound, and that all the states whose shortest distances equal the depth bound are returned in the output frontier.

Lemma 1. *Consider the invocation of the method* BoundedDfs *from the initial state with a depth bound* d*. Consider any state* S *whose shortest path from the initial state is* $\ell < d$*, where* d *is the depth bound. Then, the method* BoundedDfs *in Figure 5 eventually explores* S *through a path of length* ℓ *from the initial state, and updates the value for key* S.fp *to* ℓ *in the hash table* DoneStates*.*

Proof. By induction on ℓ. For $\ell = 0$ the only state is the initial state, and it is easy to check that the fingerprint for the initial state is stored in DoneStates mapped to the value 0. Consider any state S with shortest path ℓ from the initial state. Consider any shortest path P from the initial state to S. Let A be the predecessor of S in P. By induction hypothesis, the algorithm eventually explores A at depth $\ell - 1$ (since P is a shortest path, the shortest distance from the initial state to A is $\ell - 1$). At that instant, either S will be revisited with a depth ℓ, or S has already been visited at depth ℓ through another shortest path P' from the initial state. In either case, the proof is complete.

Lemma 2. *Consider the invocation of the method* BoundedDfs *from the initial state with a depth bound* d*. For any state* S*, we have that* S $\in$ outFrontier *on completion of the call to* BoundedDfs *if and only if the shortest path from the initial state to* S *is* d*.*

The Proof of Lemma 2 follows from Lemma 1. Note that a state S with shortest path $\ell < d$ may be initially added to outFrontier if it is first visited through a path of length d. However, when it is later revisited through a path of length $\ell < d$, it will be removed from outFrontier.

Theorem 1. *The algorithm shown in Figure 5, in conjunction with the algorithm in Figure 2, run with a depth increment of* i *and a depth bound* d*, explores a state if and only if it is reachable from the initial state via at least one path of length less than or equal to the depth bound* d*.*

Proof. As mentioned earlier, we assume that $i > 0$, $d > 0$ and d divides i. The Theorem follows by repeated application of Lemma 1 and Lemma 2 for each level of the iterated depth bounded DFS.

4 Efficient Depth Bounding

Though the algorithm in Figure 5 is correct, it has two main inefficiencies. First, it ends up revisiting the same state several times (see Section 5 for empirical

```
Hashtable⟨Fingerprint, int⟩ Threshold

⟨bool, int⟩ mustExplore(State S) {
if Threshold.Contains(S.fp) then
   if S.Depth < Threshold.Lookup(S.fp) then
      return(⟨true, Threshold.Lookup(S.fp)⟩)
   else
      return(⟨false, Threshold.Lookup(S.fp)⟩)
   end if
else
   return(⟨true, S.Depth⟩)
end if
}

int BoundedDfs(State currentState, int depthBound) {
int currThreshold = ⊥
int myThreshold = −1
bool needsexploration = false
⟨needsexploration, currThreshold⟩ = mustExplore(currentState)
if ¬needsexploration then
   return(currThreshold)
end if
if currentState.Depth < depthBound then
   Threshold.Update(currentState, currentState.Depth)
   outFrontier.Remove(currentState)
   for all Successors S of currentState do
      S.Depth = currentState.Depth + 1
      currThreshold= BoundedDfs(S, depthBound)
      myThreshold = max(myThreshold, currThreshold − 1)
   end for
   Threshold.Update(currentState, myThreshold)
else
   outFrontier.Add(currentState)
   myThreshold = currentState.Depth
end if
return(myThreshold)
}
```

Fig. 6. Efficient Depth Bounded DFS with Thresholding

validation). Second, the storage requirement for frontier states for large depths is prohibitive. In this section, we propose optimizations for both these problems. The first optimization is a thresholding technique to reduce the number of revisits, and the second is a technique to represent frontier states efficiently using traces rather than full states and exploiting the tree-structure among these traces to reduce replay overhead.

4.1 Efficient Depth Bounded DFS with Thresholding

Figure 6 presents an improved algorithm for depth bounded DFS. The key idea in this algorithm is to *propagate* the reason why a state need not be explored upwards in the call stack (which represents the depth bounded DFS stack) by maintaining a threshold value for each visited state. Intuitively, the threshold value for a state S corresponds to the maximum depth at which the S needs to be revisited so that there is a possibility of exploring a previously unexplored state within the current depth bound.

The example in Figure 7 motivates the use of threshold. In the example, we suppose that the state S is explored at a depth of 50. Further, suppose (for

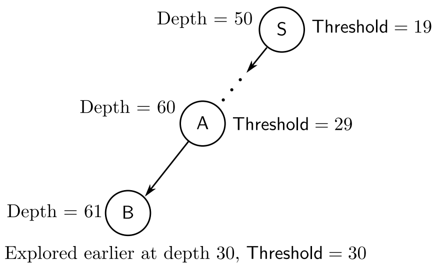

Fig. 7. Example to motivate use of threshold

simplicity) S has only one successor, and each transitive successor of S also has only one successor. After 10 more steps, suppose state A is explored, at a depth of 60. Finally, suppose the successor B of A has been explored before at a lower depth 30. In this case, the algorithm in Figure 5 merely stops exploring state B, since it has been encountered at a lower depth 30 before. However, in the hash table DoneStates, the minimum depth at which A has been encountered is still set to 60, and the minimum depth at which S has been encountered is still set to 50. Suppose S is now revisited with a depth of 45. Then, the algorithm in Figure 5 revisits all the states from S to A, since they are now revisited at lower depths. In particular, A is now revisited at a lower depth 55. However, all the revisits of the states along the path S to A are wasteful, since B is encountered at a depth 56, which is still higher than 30. The idea behind thresholds is to propagate the lower bound 30 for revisiting B back along the path from S to A. In particular, since B is the only successor of A, the threshold for revisiting A is 29, which is one less than 30. By repeating this propagation along the path from S to A, the threshold for S is calculated as 19. This means, that even though the minimum depth at which S has been encountered so far is 50, a revisit of S is needed only at depths lesser than 19, since revisits at depths larger than 19 will not lead to any new states being explored. In this example, S and its successors had only one successor. We can easily generalize to the case where S has multiple successors by calculating the threshold of S to be maximum among the thresholds propagated from all the successors of S.

The algorithm in Figure 6 maintains thresholds instead of minimum depths for each state S in the hash table Threshold. The threshold of a state represents the depth at which the state needs to be revisited. It is guaranteed that exploring the state at a depth greater than the threshold will never result in exploring new states within the current depth bound. Whenever the mustExplore function returns `false`, indicating that a state need not be explored, it also returns a threshold value for the state. The BoundedDfs function then calculates and updates the threshold value for a state S as the maximum of the thresholds of all its successors minus one, thus propagating the threshold values up the call stack.

We use the expression $\mathsf{Threshold(S)}$, where S is a state to represent the threshold value for S as stored in $\mathsf{Threshold}$. Also, we use the expression $\mathsf{minDepth(S)}$ to represent the length of the shortest path from the initial state to S. At any point in the execution of the algorithm $\mathsf{S.Depth}$ is the current depth at which S has been reached. One invariant (which holds at all times during the algorithm) is that $\mathsf{minDepth(S)} \leq \mathsf{S.Depth}$ for all states S. For any state $\mathsf{S} \in \mathsf{Threshold}$, we also maintain the invariant that $\mathsf{Threshold(S)} \leq \mathsf{S.Depth}$. Further, the threshold for a given state is non-increasing over the course of the algorithm execution.

We define a *frontier state* as a state which is reachable by a shortest path of length exactly d from the initial state, and an *internal state* as a state which is reachable by a shortest path of length strictly less than d, where d is the current depth bound.

The main technical difficulty in establishing the correctness of Algorithm 6 is that Lemma 1 does not hold. Specifically, during the execution of Algorithm 6, suppose for a state S, we have that $\mathsf{Threshold(S)} < \mathsf{minDepth(S)}$. Consequently, $\mathsf{mustExplore}$ returns `false` for any attempt to revisit S, and the shortest path to S is not explored by the algorithm. Thus, Lemma 1 does not hold, and the correctness of Algorithm 6 is nontrivial. Interestingly, when $\mathsf{Threshold(S)} < \mathsf{minDepth(S)}$, even though we may miss exploring the shortest path to S, this does not affect the frontier states that can be reached from S. Below, we formalize this intuition and establish the correctness of Algorithm 6.

Lemma 3. *For a given state* S *if* $\mathsf{Threshold(S)} < \mathsf{minDepth(S)}$ *at some point in the execution of the algorithm shown in Figure 6, then* S *is not along any shortest path from the initial state to some frontier state.*

Proof. Suppose that a S was along the shortest path from the initial state to a frontier state F and that $\mathsf{Threshold(S)} < \mathsf{minDepth(S)}$. Consider the point of time during the execution of the algorithm that the update to $\mathsf{Threshold(S)}$ making it less than $\mathsf{minDepth(S)}$ occurred. Since the updates occur *after* all the recursive calls have completed, it must be the case that S was explored during the call at which the update occurred. Since S is along the shortest path to some frontier state F either the frontier itself was reached and the recursive returns along this path effectively propagated the depth at which S was encountered back to S, in which case $\mathsf{Threshold(S)} = \mathsf{S.Depth}$, a contradiction! The other case is that the frontier was not explored along this path due to S not being encountered at its minimum depth. In this case as well, some other state F' will be added to the frontier and $\mathsf{Threshold(S)}$ will again be set to $\mathsf{S.Depth}$. But $\mathsf{S.Depth} \geq \mathsf{minDepth(S)} \implies \mathsf{Threshold(S)} \geq \mathsf{minDepth(S)}$, which is again a contradiction, completing the proof.

Lemma 4. *Exploring a state* S *when encountered at a depth greater than the* $\mathsf{Threshold(S)}$ *will not result in any new states being discovered in the current depth bounded iteration.*

Proof. For all the states S where $\mathsf{Threshold(S)} \geq \mathsf{minDepth(S)}$, the proof holds from Theorem 1, since in this case, the optimized algorithm is equivalent to the naïve algorithm. For the cases where Threshold(S) < minDepth(S), we have from Lemma 3 that these states are not along the shortest path to any frontier state. This implies that *all* states S′ that are reachable from S are also reachable at a lower depth via some other state. The threshold calculations in this case effectively propagate the depth at which this S must be revisited in order to have any possibility of exploring new states.

Theorem 2. *The algorithm shown in Figure 6 in conjunction with the algorithm shown in Figure 2, when run with a depth bound d, explores all states that are reachable within d states from the initial state.*

Proof. We can conclude this result from Lemma 4 and Theorem 1; Since the algorithm in Figure 6 is essentially the same as the algorithm in Figure 5, except for the decision to revisit or not which is based on the Threshold instead of the depth of the state.

Section 5 gives empirical data with shows that the Optimized Depth Bounded DFS algorithm greatly reduces the number of revisits to states without compromising on correctness.

4.2 Traces and Frontier Trees

Though the optimized depth bounded DFS algorithm in Figure 6 greatly reduces the number of revisits for a state, we still have the issue that the space required to store the frontier states at each iteration of the depth bounded search explodes with increasing depth. In particular, the amount of storage required to store the set outFrontier in Figure 2 becomes prohibitively expensive for large depths. Thus, we end up storing in lieu of each state s in outFrontier a *trace* t, which is a path from the initial state to s. If the length of the path is d, the storage requirement for t is $O(kd)$ bits for some small k, since at each level we only need to store a unique identifier for each outgoing edge from each state. In contrast, the storage requirement for a state s is on the order of hundreds of kilobytes for the large models we have. However, the price paid for storing t instead of s is that we finally need s in order to explore successors of s, and generating s from t takes time $O(d)$, which becomes expensive for large d.

```
class FrontierNode {
FrontierNode p
Trace t
}

State getState( FrontierNode f, State sf, FrontierNode g) {
FrontierNode a = LowestCommonAncestor(f,g)
t = TraceFrom(g, a)
State sa= sf.UnwindTo(a)
State sg= sa.ExecuteTrace(t)
return (sg)
}
```

Fig. 8. Using frontier trees to optimize replay overhead

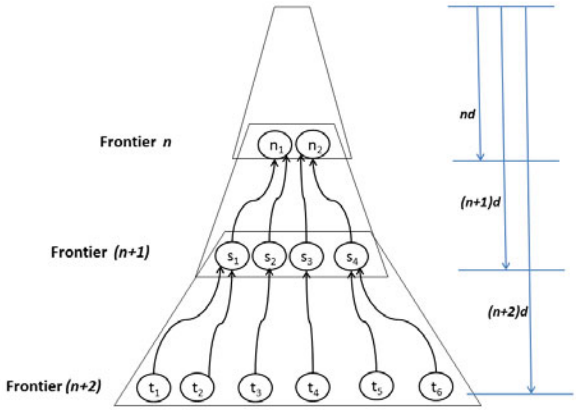

Fig. 9. Frontier tree

To optimize the trade-off between space and time, we introduce a data structure called frontier tree. Instead of storing states at the frontier, we store a FrontierNode for each state (see Figure 8) with two fields: (1) a pointer p to the parent node, and (2) a trace t from the parent node p to this node. Figure 9 shows a pictorial description of the frontier tree that is formed using the frontier nodes at levels nd, $(n+1)d$, and $(n+2)d$. Suppose we have just finished exploring all the successors of frontier node t_1 in Figure 9. Next, we need to explore the successors of t_2. To get the state corresponding to t_2, if we replay the trace associated with t_2 all the way from the initial state, the replay would take $O((n+2)d)$ time. Instead, we can find the least common ancestor of t_1 and t_2 in the frontier tree, which is s_1, and do the following: (1) first construct the state corresponding to s_1 by executing the undo logs from s_1 to t_1 using "state-delta" (See Section 2), and (2) replay only the trace from s_1 to t_2 to get the state corresponding to t_2. This can be done in $O(2d)$ time, since it takes $O(d)$ time to execute undo logs from t_1 to s_1 and another $O(d)$ time to execute the trace from s_1 to t_2. Procedure getState in Figure 8 shows that given a frontier node f with corresponding state sf, we can construct the state corresponding to frontier node g by unwinding to the least common ancestor a of f and g, and replaying only the trace from a to g.

As shown by our empirical results in Section 5, this greatly reduces the overhead of replay, and hence the overall execution time of the iterative depth bounded search.

5 Empirical Results

We have implemented both the optimized iterative depth bounding DFS algorithm (Figure 6, Section 4) as well as the frontier tree optimization (Section 4) in the Zing model checker [1,2]. We evaluate the effectiveness of these optimizations below.

Table 1. States explored and peak memory usage for a fixed time budget (depth increment = 1000)

Model	Time budget (hh:mm)	Distinct States Explored			Maximum Depth Explored			Peak Memory Usage	
		Unopt	Opt	Increase (%)	Unopt	Opt	Increase (%)	Unopt (MB)	Opt (MB)
ISM	2:30	5.933M	7.499M	26.4%	975	1025	5.1%	1644	1712
PSM30	3:30	499.1K	1.462M	192.9%	2750	3400	23.6%	1201	1341
PSM20	5:30	859.0K	2.233M	159.9%	2500	2800	12%	767	872
DSM	5:30	92.31K	1.243M	1246.8%	2650	4950	86.8%	1108	1127

The primary motivation for the algorithms described in this paper was to help with the design of the USB stack in the Windows operating system. Our models of the USB stack have very large state spaces with very large depths, and we have not been able to explore all the states of these models. Thus, we do not even know the total number of states or depth of the state space of these models. However, even by exploring all states within fixed depth bounds, we have been able to find and fix over 300 bugs in the design of the USB stack. Several of the bugs were only discovered at depths greater than 1000. Every time the USB designers run the model checker, they fix a time budget(say a few hours or a few days), and they want to explore all states of these models for as large a depth bound as possible within this time budget, to get high confidence in their design.

We evaluate the efficacy of our optimizations on models from the USB team. Table 1 shows this data for 4 different models: ISM, PSM30, PSM20 and DSM. In this table "Unopt"refers to the unoptimized algorithm in Figure 5, but with traces used to represent frontier states (if we store frontier states in full, we run out of memory for these models), and "Opt" refers to the optimized algorithm in Figure 6, with frontier trees. Thus, the data in Table 1 measures the combined gains due to the depth thresholding and frontier tree optimizations.

The second column in Table 1 shows the amount of time budget given to the optimized and unoptimized algorithms, and the remaining columns compare the number of distinct states and the maximum depth that was completely explored within that time budget. We note that the optimizations enable the model checker to explore more states, and also enable the model checker to explore all the states up to larger depths. Most notably, in the DSM model the optimizations enable the model checker to explore all states up to a depth of 4950, which is an 86.8% improvement in the depth of states explored, and 1246.8% improvement in the number of states explored. We also note that the optimizations add only a very small memory overhead, as evidenced by the data in the last two columns.

We note that we know of no other method that can explore all reachable states of these models for as large a depth bound as possible within a fixed time and memory budget. Breadth-first search simply runs out of memory for these models since each state in the frontier occupies hundreds of kilobytes.

Table 2. Reduction in number of revisits due to thresholding

Model	Depth	Distinct States Explored	Depth Increment = 1000			Depth Increment = 50		
			Revisits Without Threshold	Revisits With Threshold	Reduction in revisits	Revisits Without Threshold	Revisits With Threshold	Reduction in revisits
TMCompletionEventFixed	1000	231.1K	52.32K	24.49K	53.2%	15.6K	15.3K	1.9%
TMHashTableFixed	1000	3.000M	1.902M	113.2K	1581%	272.47K	268.25K	1.52%
ISM	1000	4.924M	2.404M	2.317M	3.6%	1.41M	1.39M	1.4%
PSM20	2700	649.9K	3.834M	2.205M	42.5%	785.5K	690.49K	12.09%
PSM30	3000	145.3K	3.233M	1.968M	39.1%	284.69K	249.9K	12.22%
DSM	6000	423.3K	3.133M	1.822M	42%	1.36M	1.34M	1.47%
HSM	16000	186.9K	438.9K	287.8K	34.3%	50.54K	48.49K	4.05%

Table 3. Time to explore a fixed depth with and without frontier tree

Depth Increment = 50						
Model	Depth	Without Frontier-Tree		With Frontier-Tree		Reduction in
		Execution Time(sec.)	getState() Time(sec.)	Execution Time(sec.)	getState() Time(sec.)	execution time (%)
ISM	1000	3006	621	2430	64	19.1%
PSM20	2700	7411	4183	3033	284	59.1%
PSM30	2700	1674	951	704	115	57.9%
DSM	6000	15023	9695	4391	536	70.8%
HSM	16000	6530	4579	1114	365	82.9%

Next, we measure the effect of the thresholding and frontier tree optimizations separately. First, we measure the reduction in the number of revisits of states due to thresholding. Table 2 compares the number of revisits of states with and without the use of thresholds. The first two models, TMCompletionEventFixed and TMHashTableFixed are models of a distributed transaction manager. The remaining models ISM, PSM20, PSM30, DSM and HSM are all various state machine components of the USB stack. As the results show, thresholding reduces revisits, without compromising on the soundness. The reduction in the number of revisits is dependent on both the model and the depth increment. With a large depth increment (such as 1000), each state is reachable through a large number of paths within each iteration of BoundedDfs, and thresholding is able to greatly reduce the number of revisits. With a small depth increment (such as 50), the number of revisits is relatively small even without the use of thresholding, and thresholding is relatively less effective.

Finally, we measure the gains due to frontier trees. We fix a large depth bound for these models and measure how long it takes to explore all the states within the depth bound with and without frontier trees. We do these measurements

Table 4. Time to explore a fixed depth with and without frontier tree

Depth Increment = 1000						
Model	Depth	Without Frontier-Tree		With Frontier-Tree		Reduction in
		Execution Time(sec.)	getState() Time(sec.)	Execution Time(sec.)	getState() Time(sec.)	execution time (%)
ISM	1000	9198	3	9197	4	0.01%
PSM20	2700	24420	15621	18123	11284	25.78%
PSM30	2700	11523	7819	8136	1693	29.39%
DSM	6000	38623	28133	24929	14455	35.45%
HSM	16000	16860	16080	3355	2498	80.1%

both for a depth increment of 50 (see Table 3) and a depth increment of 1000 (see Table 4). The results show substantial reduction in execution times due to the frontier trees. The reduction is larger with depth increment 50 than depth increment 1000, since at depth increment 50, there are several more frontiers, and several more replays done, and the scope for savings in replay is more. The results both establish that (1) the time required to replay traces to generate full states for the frontier is a significant fraction of the total execution time, and (2) the frontier tree optimization greatly reduces the replay overhead.

The results show that thresholding is more effective at large depth increments, and frontier trees are more effective at small depth increments. The combination of the two optimizations improves the effectiveness of the model checker for all depth increments.

6 Related Work

The use of fingerprints to save storage in model checkers was first introduced by Holzmann who called it "bit-state hashing" [13]. Holzmann's SPIN model checker also supports bounded depth first search, but it does not attempt to optimize the number of revisits or the replay overhead, which are the main contributions of our work.

The use of traces instead of states to space has been observed before in software model checking. In particular, Verisoft [9] is a stateless model checker, which only remembers traces of states to save space, and works essentially by replaying traces from the initial state. The use of "state delta" or undo logs to store only differences between states on the DFS stack has been explored before in several model checkers such as CMC [17], JPF [10] and Zing [2]. Frontier trees combine the use of traces and undo logs to greatly reduce the replay overhead during iterative depth bounded DFS.

While at first glance, our approach to depth-bounding looks similar to the iterative deepening algorithms such as IDA* [16], there are significant differences. The approach presented in [16] and other related work primarily aims to reduce the number of states visited while arriving at an optimal solution. In contrast, the work presented in this paper aims to reduced the number of *revisits* to a given state, while ensuring that *every* state which is reachable, given the depth bound, is indeed explored. Also, the algorithms along the lines of the algorithm presented in [16] require the use of a heuristic *cost-function* f with some characteristics: specifically, that f never overestimate the true cost of exploring a given path and that f have some monotonicity properties. In our context, since a bug can manifest anywhere, the use of such monotonic cost metrics is not feasible.

7 Conclusion

We presented algorithms to systematically bound the depth of the state spaces explored by explicit-state model checkers. Since explicit-state space model checkers use DFS for space efficiency, depth bounding is non-trivial to do correctly and efficiently. In particular, we presented a bounding algorithm to greatly avoid the number of revisits of states, and a new data structure called Frontier Tree to optimize the replay overhead during iterative depth bounding. Our depth-bounded model checker has been used by product groups inside Microsoft to successfully find several hundred bugs in large real-world models, and the use of depth bounding was crucial in these applications.

Though we focus on checking safety properties, our techniques can be adapted to check liveness properties as well. Let Θ be the set of all states of a model that are reachable from the initial state. Let $\Theta_d \subseteq \Theta$ be the set of all states that can be reached from the initial state at a depth of d or less. Let $\Gamma \subseteq \Theta$ be a set of Büchi states. Our algorithms can be adapted to look for all lassos which consist of a "stem" from an initial state to a state $\mathsf{S} \in \Gamma$ and a cycle back to S such that all states in the lasso are reachable within a distance d from the initial state. In particular, consider the nested depth-first algorithm of Corcoubetis, Vardi, Wolper and Yannakakis [7]. Given a depth bound d, we can first compute Θ_d using the techniques given in Section 3 and Section 4. Then, we can restrict the search in both phases of the nested DFS algorithm to remain within Θ_d. This can be proved to search for all lassos such that all states in the lasso are reachable within a distance d from the initial state.

Currently, we are working on parallelizing the depth bounded model checker in both multi-cores and clusters of workstations. We plan to present these results in a future paper.

Acknowledgment. We thank Randy Aull, Tom Ball, Vivek Gupta, Vlad Levin, Madan Musuvathi, Aditya Nori and Shaz Qadeer for helpful discussions.

References

1. Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: Zing: A model checker for concurrent software. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 484–487. Springer, Heidelberg (2004)
2. Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: Zing: Exploiting program structure for model checking concurrent software. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 1–15. Springer, Heidelberg (2004)

3. Ball, T., Rajamani, S.K.: The SLAM Project: Debugging system software via static analysis. In: POPL 2002: Principles of Programming Languages, pp. 1–3. ACM, New York (2002)
4. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: LICS 1990: Logic in Computer Science, pp. 428–439 (1990)
5. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design 19(1), 7–34 (2001)
6. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
7. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory efficient algorithms for the verification of temporal properties. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 58–328. Springer, Heidelberg (1991)
8. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer, Heidelberg (1996)
9. Godefroid, P.: Software Model Checking: The Verisoft Approach. Formal Methods in System Design 26, 77–101 (2005)
10. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer (STTT) 2, 366–381 (2000)
11. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL 2002: Principles of Programming Languages, pp. 58–70. ACM, New York (2002)
12. Holzmann, G.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (May 1997)
13. Holzmann, G.J.: An analysis of bitstate hashing. Form. Methods Syst. Des. 13, 289–307 (1998)
14. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
15. Katz, S., Peled, D.: Verification of distributed programs using representative interleaving sequences, vol. 6, pp. 107–120 (1992)
16. Korf, R.E.: Depth-first Iterative-deepening: An Optimal Admissible Tree Search. In the Journal of Artificial Intelligence 27, 97–109 (1985)
17. Musuvathi, M., Park, D.Y.W., Chou, A., Engler, D.R., Dill, D.L.: CMC: a pragmatic approach to model checking real code. SIGOPS Oper. Syst. Rev. 36, 75–88 (2002)
18. Sistla, A.P., Godefroid, P.: Symmetry and reduced symmetry in model checking. ACM Trans. Program. Lang. Syst. 26, 702–734 (2004)
19. Stern, U., Dill, D.L.: Improved probabilistic verification by hash compaction. In: Camurati, P.E., Eveking, H. (eds.) CHARME 1995. LNCS, vol. 987, pp. 206–224. Springer, Heidelberg (1995)