Depth Bounded Explicit State Model Checking

Abhishek Udupa[‡] Ankush Desai[†] Sriram Rajamani[†]

[‡] University of Pennsylvania
[†] 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

Model Checking is an algorithmic technique to systematically explore the reachable states of a system's model, and verify if the system satisfies specified properties [6, 23, 9]. Though model checking has been successful in several academic and industrial efforts to find bugs and prove properties of various systems ranging from coherence protocols [7], telephone software [12], OS components [20] and device drivers [3] two main obstacles remain: (1) usually models of the system need to be written at a level of abstraction that is amenable for model checking, and (2) for large systems, the number of states is too large to be explored with a reasonable amount of space and time.

The second problem is referred to as "state explosion" in the model checking community and several optimizations have been studied extensively, such as such as symbolic model checking [4],partial order reduction [11, 17], symmetry reduction [24], automated abstraction-refinement [8, 3, 14]. Even with all of the above optimizations, model checkers are still unable to cope with the state spaces of very large models. To cope with such large state spaces, bounding techniques have been proposed to systematically explore a part of the state space. Notable examples of bounding are Bounded Model Checking(usually abbreviated as BMC) [5], context bounding [22, 21, 19], and domain size bounding [16]. The key idea with bounding based approaches is given by the "small-scope hypothesis" [16], which states that if a model is buggy then the bug will most likely manifest by exploring all states systematically after bounding a parameter (such as depth, input size, number of processors, and 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— with iterative deepening of the depth bound. While BMC techniques solve this problem using symbolic model checking, the corresponding analog for explicit state model checking has not been studied carefully before. Large software models have features such as unbounded call stacks and dynamically allocated memory which make it impossible to even build a symbolic transition relation for the model, and explicit state model checking is currently the only viable alternative for checking such models. Thus, it is important to be able to do systematic depth bounding for explicit state model checking.

The obvious way to bound depth in explicit state model checking is to use breadth first search. However, using breadth-first search in explicit state model checking is very expensive in terms of space, particularly if each state consumes a lot of storage. The common way to avoid memory explosion in explicit state model checking is to use depth first search, and store only fingerprints or bit-state hashes [15, 25] for each visited state. For states on the DFS stack, only the top most state needs to be stored in full —the remaining states can be stored in terms of incremental differences or undo logs from the states above them 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 quite non-trivial. The obvious way to bound depth is to record the depth of each state, and simply stop exploring a state if has been either visited earlier, or if the current depth exceeds the depth bound. However, as we show in Section 2, 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 additional information for each visited state and greatly reduces the number of times a state is revisited, while ensuring that all states that can be reachable within the depth bound are indeed explored.

Since it is hard to pick a good depth bound apriori, 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. Such an iterative depth bounded search combines elements of both depth first search and breadth first search. In order to save space in iterative depth bounded search, frontier states at each depth bound are represented 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. However, for a trace of length d producing a full state by replaying takes time $O(d)$, and the replay overhead becomes large for large values of d. We propose a data structure called frontier tree to greatly reduce the replay overhead during iterative depth bounding.

The main motivation for our work came from a product group's desire to use our model checker to explore state spaces of very large models from Microsoft product groups, including models of distributed transaction managers, and models from components of the Windows Operating System used to manage devices connected to the USB (Universal Serial Bus). We show empirical results showing effectiveness of our algorithms on these models.

To summarize, this paper makes the following contributions:

- First, it presents a new algorithm to efficiently and correctly perform iterative depth bounded search with explicit state model checkers.
- Second, it presents a data structure called frontier tree to greatly reduce the replay overhead during iterative depth bounding.
- Third, it presents empirical performance results obtained by applying our approach on several large models from Microsoft's product groups.

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.

Though we focus on checking safety properties, our techniques can be adapted to check liveness properties as well. In particular, we describe how our techniques can be adapted to be used as a pre-processing technique for the nested depth first search algorithm of Corcoubetis, Vardi, Wolper and Yannakakis [10] to search systematically for all lassos within a given depth bound.

The remainder of the paper is organized as follows. Section 2 gives background on explicit state model checkers. Section 3 describes the Iterative Depth Bounded Search problem and develops our algorithms for this problem. Section 5 describes how our algorithms can be adapted to check liveness properties. Section 6 presents empirical results. Section 7 describes related work, and Section 8 concludes the paper.

2 Background

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

We assume the existence of the following datatypes. A *State* datatype is used to represents 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 datatype, which supports three methods: (1) the Add method takes an object and adds it to the set, and (2) the 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

```
$Set\langle Fingerprint\rangle$ 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

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:

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

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 [15, 25]).

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 has is already in the set DoneStates, and if not, adds it to DoneStates, and invokes itself on all its successors.

DFS is a very 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 several explicit state model checkers (see, for instance [2]).

In the next section, we use another generic datatype *Hashtable*. We instantiate *Hashtable* with fingerprints as keys and integer values in the next section. It 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.

```
1: $bool$ IterBoundedDfs($State$ initialState, $int$ depthCutoff, $int$ inc) {
2: initialState.Depth= 0
3: $Set\langle State\rangle$ frontier = {initialState}
4: $Set\langle State\rangle$ newFrontier= {}
5: $int$ currentBound = inc
6: while currentBound $\leq$ 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: $Set\langle State\rangle$ outFrontier
19: /* outFrontier is a global variable which gets updated inside BoundedDfs*/
20: BoundedDfsFromFrontier($Set\langle State\rangle$ frontier, $int$ currentBound) {
21: outFrontier= { }
22: for all F$\in$ frontier do
23:    BoundedDfs( F, currentBound)
24: end for
25: return(outFrontier)
26:
```

Fig. 2. Iterative Depth Bounded Search Algorithm

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.

3 Depth Bounded Search

In this section, we modify the simple explicit state 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. As mentioned in Section 1, our motivation is that the state spaces of several 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 [16].

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 works by repeatedly calling the BoundedDfsFromFrontier method (line 7) in the while loop from lines 6–14. If

```
 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

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.

The implementaton of BoundedDfsFromFrontier is shown in lines 20–25. It calls the BoundedDfs function for each state in the froniter 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 quite 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 8 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 Sis 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 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

$$\begin{tikzcd} s_1 \arrow[r, "e_1"'] \arrow[rr, bend left, "e_3"] & s_2 \arrow[r, "e_2"'] & s_3 \arrow[r, "e_4"'] & s_4 \arrow[r, "e_5"'] & s_5 \end{tikzcd}$$

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

```
$Hashtable\langle Fingerprint, int\rangle$ DoneStates
/* initialized to null-set once in the beginning */

$Set\langle State\rangle$ outFrontier
/* initialized to null-set in BoundedDfsFromFrontier*/

bool mustExplore(State S) {
if DoneStates.Contains(S.fp) then
    if DoneStates.Lookup(S.fp) $\leq$ 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

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 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 hashtable DoneStates (rather than a set) to store fingerprints of visited states. For each visited state S, the hashtable 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 hashtable is updated 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* S *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 hashtable* 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 *iff 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 conjuction 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.

Though the algorithm in Figure 2 is correct, it is very inefficient in practice, and it ends up revisiting the same state several times (see Section 6). Next, we present an improvement to reduce the number of revisits.

```
$Hashtable\langle Fingerprint, int\rangle$ DoneStates

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

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

Fig. 6. Optimized Depth Bounded DFS algorithm

Optimized Sound Depth Bounded DFS. Figure 6 presents our improved algorithm for depth bounded DFS. We will refer to this as *Optimized Sound DBDFS*. The main idea in the improved algorithm is to propagate some additional information up the call stack whenever a successor need not be explored to avoid unnecessary revisits to states.

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 state S $\in$ DoneStates. Intuitively this corresponds to the depth at which the state needs to be re-explored so that there is a possibility of exploring a previously unexplored state.

The algorithm works as follows: Associated with each state S in $\mathsf{DoneStates}$, we maintain a threshold (instead of the least depth at which S has been encountered as before). The threshold of a state represents the depth at which the state needs to be re-explored. It is guaranteed that exploring the state at a depth greater than the threshold will never result in exploring new states. Obviously, for a given state $\mathsf{S} \in \mathsf{DoneStates}$, we have that $\mathsf{threshold(S)} \leq \mathsf{S.Depth}$ and the threshold for a given state is non-increasing over the course of the algorithm execution. Whenever the $\mathsf{mustExplore}$ function returns `false`, indicating that a state need not be explored, it also returns a threshold value for the state. The $\mathsf{BoundedDfs}$ function then calculates and updates the $\mathsf{threshold}$ 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 $\mathsf{LiveStates}$. Also, we use the expression $\mathsf{minDepth(S)}$ to represent the length of the shortest path from the initial state to S. Also, 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 $l < d$, for a given depth-bounded iteration. Below, we state lemmas and a theorem that establishes correctness of the optimized depth bounded DFS algorithm.

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 the 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 $\mathsf{F'}$ 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

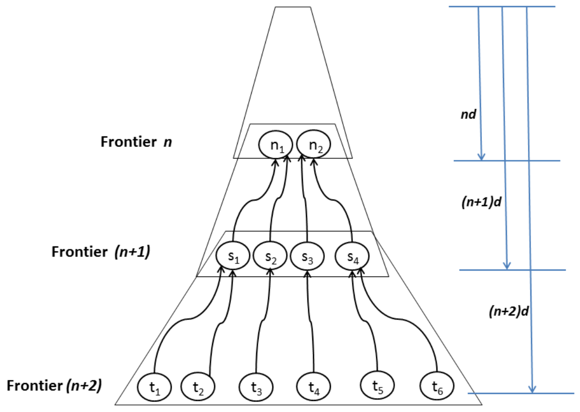

Fig. 7. Frontier tree

the naïve algorithm. For the cases where $\mathsf{threshold}(\mathsf{S}) < \mathsf{minDepth}(\mathsf{S})$, 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 Smust be re-explored 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 re-explore or not which is based on the threshold instead of the depth of the state.

Section 6 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 Frontier Trees to Optimize Replay Overhead

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($g$, $a$)
$sa$ = $sf$.UnwindTo($a$)
$sg$ = $sa$.ExecuteTrace($t$)
return ($sg$)
}
```

Fig. 8. Using frontier trees to optimize replay overhead

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 7 shows a pictorial description of the frontier tree that is formed using the frontier nodes at various levels. Suppose we have just finished exploring all the successors of frontier node t_1 in Figure 7. 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 6, this greatly reduces the overhead of replay and hence the overall execution time of the iterative depth bounded search.

5 Liveness

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 [10]. 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.

6 Empirical Results

We have implemented both the optimized iterative depth bouding DFS algorithm (Figure 6, Section 3) as well as the frontier tree optimization (Section 4) in the Zing model checker [1, 2].

The Zing model checker has two components: (1) a compiler for translating a Zing model into an executable representation of its transition relation, and (2) a model checker for exploring the state space of the Zing model.

The Zing modeling language has several features that capture the essence of modern concurrent object oriented languages, including procedure calls with a call-stack, objects with dynamic allocation, processes with dynamic creation, and facilities for using both shared memory and message passing for inter-process communication. A "choose" construct is provided that can be used to non-deterministically pick an element out of a finite set of integers, enumeration values, array members or set elements.

We were able to implement both the optimized iterative depth bounding DFS algorithm, and the frontier tree optimization fully inside the model checker, without making any changes to the Zing compiler.

Table 1 compares the number of revisits of states for the optimized depth bounded DFS algorithm(Figure 6) and the naive depth bounded DFS algorithm(Figure 5 for fixed depth cutoffs. The models used for the comparison are all large real-life models used by product groups inside Microsoft. 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 inside the Windows operating system. As the results show, the optimized algorithm greatly reduces the number of states that are revisited, without compromising on the soundness. The reduction in the number of revisits is model dependent. In

Model	Depth-Cutoff	Distinct States Explored	Revisits Naive (Figure 5)	Revisits Optimized (Figure 6)	Reduction in number of revisits (%)
TMCompletionEventFixed	1000	231056	52322	24489	53.2%
TMHashTableFixed	1000	3000230	1902332	113163	99.4%
ISM	1000	4924340	24038436	23168493	3.6%
PSM20	2700	649886	3834495	2205435	42.5%
PSM30	3000	145361	3233021	1967829	39.1%
DSM	6000	423348	3133430	1822287	42%
HSM	16000	186899	438923	287846	34.3%

Table 1. Number of revisits for a fixed depth bound

most models, the optimized algorithm reduces the number of revisits by 35-42%. We found two extreme cases —in one model (ISM) the reduction in the number of revisits is only 3% and in another model (TMHashTableFixed) the reduction is 99.4%. We have empirically verified that the number of distinct states explored by the optimized algorithm in Figure 6 is exactly the same as the number of states explored by the naive algorithm in Figure 5, thereby providing empirical confirmation of Theorem 2.

Model	Depth	Without Frontier-Tree		With Frontier-Tree		Reduction in execution time (%)
		Execution Time(sec.)	getState() Time(sec.)	Execution Time(sec.)	getState() Time(sec.)	
TMCompletionEventFixed	1000	89.9	15.8	130.32	04.22	-44.9%
TMHashTableFixed	1000	1281.3	253.98	1233.75	47.262	3.7%
ISM	1000	3006.021	621.5066	2430.9	64.135	19.1%
PSM20	2700	7411.6	4183.8	3033.727	284.093	59.1%
PSM30	2700	1674.7	951.36	704.96	115.36	57.9%
DSM	6000	15023.77	9695.36	4391.64	535.506	70.8%
HSM	16000	6529.723	4579.131	1113.81	364.66	82.9%

Table 2. Time to explore a fixed depth with and without frontier tree

Table 2 shows the execution times for these models with and without the frontier tree optimization. We also show the time spent by the model checker in the getState method. 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 reply overhead. For the first two models (TMCompletionEventFixed and TMHashTableFixed), the total depth of the state space is low and it is more efficient to replay from the initial state rather than compute the least common ancestor using frontier trees. For the remaining models with larger depths, the performance gains from using frontier trees is significant.

Suppose we have a fixed time budget (say a few hours or a few days). Then, it is useful to investigate how many more distinct states can be explored using our optimizations. Table 3 shows this data for 4 different models. The fourth column is the number of distinct states we were able to explore with both the

Model	Time budget (hh:mm:ss)	Distinct States Explored			Peak Memory Usage	
		Naive (Figure 5) No Frontier Tree	Optimized (Figure 6) Frontier Tree	Improvement in number of states(%)	No Frontier Tree	Frontier Tree
ISM	2:30:00	5933009	7499284	26.4%	1644 MB	1712 MB
PSM30	3:30:00	499074	1461939	192.9%	1201 MB	1341 MB
PSM20	5:30:00	859004	2232549	159.9%	767 MB	872 MB
DSM	5:30:00	92305	1243204	1246.8%	1108 MB	1127 MB

Table 3. States explored and peak memory usage for a fixed time budget

optimized search (Figure 6) and frontier tree enabled. The third column shows the number of distinct states we could explore with the naive algorithm (Figure 5) with frontier tree disabled (that is, without the optimizations). We note that in all models, the fourth column is significantly bigger than the third column, showing that under a fixed time budget, our algorithms allow exploring orders of magnitude more states. We also note that the frontier tree optimization adds only a very small memory overhead, as evidenced by the data in the last two columns.

Together, the optimized depth bounded algorithm and the frontier tree optimization has enabled our model checker to scale and handle several large models constructed by Microsoft's product groups. One product group uses the checker as part of its design validation and it has found several hundred design bugs due to concurrency and asynchrony. The details of the bugs found are not relevant to the focus of this paper, which is on the optimization algorithms, so we do not present them.

7 Related Work

The use of fingerprints to save storage in model checkers was first introduced by Holzmann who called it "bit-state hashing" [15]. 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 [12] 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 [20], JPF [13] 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* [18], there are significant differences. The approach presented in [18] 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 [18] 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.

8 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-life models, and the use of depth bounding was crucial in these applications.

Currently, we are working on parallelizing the depth-bounded model checker in both multicores 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, Aditya Nori and Shaz Qadeer for helpful discussions.

References

1. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *CAV 2004: Computer Aided Verification*, LNCS 3114, pages 484–487. Springer-Verlag, 2004.
2. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *CONCUR 2004: Concurrency Theory*, LNCS 3170, pages 1–15. Springer-Verlag, 2004.
3. T. Ball and S. K. Rajamani. The SLAM Project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, January 2002.
4. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS 90: Logic in Computer Science*, pages 428–439, 1990.
5. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
6. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs*, pages 52–71, 1981.
7. E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the futurebus+ cache coherence protocol. In *CHDL: Computer Hardware Description Languages and their Applications*, pages 15–30, 1993.

8. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.
9. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.
10. C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *CAV 90: Computer Aided Verification*, LNCS 531, pages 233–242. Springer-Verlag, 1990.
11. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag, 1996.
12. P. Godefroid. Software Model Checking: The Verisoft Approach. *Formal Methods in System Design*, 26:77–101, 2005.
13. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2:366–381, 2000.
14. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02: Principles of Programming Languages*, pages 58–70. ACM, January 2002.
15. G. J. Holzmann. An analysis of bitstate hashing. *Form. Methods Syst. Des.*, 13:289–307, 1998.
16. D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.
17. S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. volume 6, pages 107–120. 1992.
18. R. E. Korf. Depth-first Iterative-deepening: An Optimal Admissible Tree Search. *In the Journal of Artificial Intelligence*, 27:97–109, September 1985.
19. A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Form. Methods Syst. Des.*, 35:73–97, 2009.
20. M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36:75–88, 2002.
21. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI 07: Programming Languages Design and Implementation*, pages 446–455. ACM, 2007.
22. S. Qadeer and D. Wu. KISS: Keep it simple and seqeuential. In *PLDI 04: Programming Language Design and Implementation*, pages 14–24. ACM, 2004.
23. J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, volume 137, pages 337–351. 1982.
24. A. P. Sistla and P. Godefroid. Symmetry and reduced symmetry in model checking. *ACM Trans. Program. Lang. Syst.*, 26:702–734, 2004.
25. U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In *CHARME 95: Correct Hardware Design and Verification Methods*, volume LNCS 987, pages 206–224. Springer-Verlag, 1995.