Distributed Explicit Bounded LTL Model Checking

Pavel Krčál, 1,2

Faculty of Informatics, Masaryk University Brno, Czech Republic

Abstract
Automated formal verification becomes a significant part of an industrial design process. Favourite formal verification method – model checking – is strongly limited by the size of the model of the verified system. It suffers from the so called state explosion problem. We propose to fight this problem by applying the idea of bounding the examined state space in explicit model checking. Moreover, we combine this approach with the distribution of the computation among the network of workstations. We consider several distributed bounded LTL model checking algorithms and carry out a series of experiments to evaluate them and to compare their behaviour.

1 Introduction
The need for higher reliability of computer systems leads to the development of automated verification methods. Model checking has become a very useful and successful method because of its push-button character.

The main challenge in model checking is the so called space explosion problem – combinatorial growth of the space of all possible system states with respect to the design size. This problem occurs in systems with many independent interacting components or systems with large data structures. As a consequence, only systems much smaller than would be desirable can be verified.

Considerable application of model checking in the industrial area of hardware designs is due to symbolic representation of the state space [7]. Other systems (like software communication protocols, etc.) lack the regularity of hardware designs exploited by the symbolic methods and thus this technique

1 This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/03/0509
2 Email: xkrcal@fi.muni.cz

©2003 Published by Elsevier Science B. V. Open access under CC BY-NC-ND license.
is not efficient for such systems [14]. Consequently, the state space has to be represented explicitly – all inspected states of the system are stored separately (explicit model checking).

A recent technique used to cope with the state explosion is the bounded model checking [3]. This technique was developed in connection with symbolic representation of the state space. The main idea is to restrict the verification just to the system runs shorter than a given bound and to reduce the whole problem to SAT. The reason for considering only bounded runs is that unbounded model checking problem cannot be translated into (single) satisfiability problem of a Boolean formula.

However, the idea of bounding the state space (which would exceed the memory capacity) can help to extend the applicability of the explicit algorithms as well. Even an “on-the-fly” algorithm (e.g. NDFS used in SPIN) can fail to find an existing short counterexample and run out of memory. We propose to solve this particular problem by bounding the examined state space in such a way that it fits into the computer memory. We suggest two bounded model checking problems: we bound the length of the runs of the examined system and we bound their depth (distance of their constituent states from the initial state of the system).

Another method used in model checking is the distribution of the computation among several computers. Cheap and common architecture, a network of workstations, is powerful enough to push the verification by extending the available computational capacity. Several attempts to distribute the explicit LTL model checking algorithms were undertaken in recent years.

A distributed version of the LTL model checker SPIN [11] based on nested depth first search approach has been explored in [1,2]. Other attempts to distribute LTL model checking are based on negative cycle detection [5] and on simulation of a symbolic algorithm [8]. Distributed explicit algorithms for branching time logics were also proposed. Paper [4] deals with alternation free µ-calculus and [6] distributes model checking of CTL.

In this paper, we study the possibilities of performing explicit bounded LTL model checking in the distributed environment in order to save the time. By employing a network of workstations we aggregate CPU power. Typical purpose of the distribution in model checking – extending the global memory – is not our primary objective because we bound the state space such that it does not exceed the memory capacity.

We propose several distributed bounded model checking algorithms working on explicitly represented state space. Main contribution of this work is experimental evaluation of these algorithms. It turned out, that the best strategy is to generate bounded state space in parallel and then run unbounded (efficient) model checking algorithm on it.
2 Bounded Semantics of Linear Temporal Logic

The set of LTL formulae is defined inductively starting from a countable set $AP$ of atomic propositions, Boolean operators, and the temporal operators $X$ (Next) and $U$ (Until). We consider formulae in negation normal form (negations only occur in front of the atomic propositions). Therefore, we add operator $R$ (Release) and for clarity we introduce also operators $F$ (Eventually) and $G$ (Globally):

$$\Psi := a | -a | \Psi \lor \Psi | \Psi \land \Psi | X \Psi | \Psi U \Psi | \Psi R \Psi | F \Psi | G \Psi$$

LTL formulae are interpreted over infinite words $w = w(0)w(1)\ldots$ over the alphabet $\Sigma = 2^{AP}$. Let $w(i)$ denote the $i$-th letter and $w_i$ the suffix of $w$ starting from the $i$-th letter.

**Definition 2.1** [Semantics] Let $w$ be an infinite word and $\alpha, \beta$ be LTL formulae. Then (unbounded) semantics of LTL is defined as follows:

- $w \models a$ iff $a \in w(0)$, for $a \in AP$,
- $w \models -a$ iff $a \not\in w(0)$, for $a \in AP$,
- $w \models \alpha \lor \beta$ iff $w \models \alpha$ or $w \models \beta$,
- $w \models \alpha \land \beta$ iff $w \models \alpha$ and $w \models \beta$,
- $w \models X \alpha$ iff $w_1 \models \alpha$,
- $w \models F \alpha$ iff $\exists i \geq 0 : w_i \models \alpha$.
- $w \models G \alpha$ iff $\forall i \geq 0 : w_i \models \alpha$.
- $w \models \alpha U \beta$ iff $\exists i \geq 0 : w_i \models \beta \land \forall 0 \leq j < i : w^j \models \alpha$.
- $w \models \alpha R \beta$ iff $\forall i \geq 0 : w_i \models \beta \lor \exists 0 \leq j < i : w^j \models \alpha$.

The basic idea of bounded model checking is to consider a finite prefix (bounded by $k$) $w = w(0)w(1)\ldots w(k)$ of an infinite word $w = w(0)w(1)\ldots$. Even a finite prefix $w = w(0)w(1)\ldots w(l)\ldots w(k)$ can represent an infinite word $w = w(0)w(1)\ldots w(l)\ldots w(k)w(l)\ldots w(k)\ldots w(k)\ldots$ if we repeat periodically a sequence of letters. We call such prefix a word with the loop from $k$ to $l$.

This observation leads to the definition of bounded semantics of LTL [3]. LTL formulae are now interpreted over finite words $w = w(0)w(1)\ldots w(k)$ over the alphabet $\Sigma = 2^{AP}$. We present the semantics that captures only finite properties of a bounded word that represents only finite behaviour. The aim is to reformulate unbounded LTL semantics such that a finite word satisfies the formula if and only if this word with any infinite suffix satisfies the formula in unbounded semantics.

We give also a definition of bounded semantics for the case where finite word represents infinite behaviour. It corresponds to the unbounded semantics of LTL.
Definition 2.2 [Bounded semantics without a loop] Let $w$ be a finite word $w = w(0)w(1)\ldots w(k)$ and $\alpha, \beta$ be LTL formulae. Then bounded semantics without a loop of LTL is defined as follows:
- $w \models_B \alpha$ iff $a \in w(0)$, for $a \in AP$,
- $w \models_B \neg \alpha$ iff $a \notin w(0)$, for $a \in AP$,
- $w \models_B \alpha \lor \beta$ iff $w \models_B \alpha$ or $w \models_B \beta$,
- $w \models_B \alpha \land \beta$ iff $w \models_B \alpha$ and $w \models_B \beta$,
- $w \models_B \mathbf{X} \alpha$ iff $k > 0 \land w_i \models_B \alpha$,
- $w \models_B \mathbf{F} \alpha$ iff $\exists 0 \leq i \leq k : w_i \models_B \alpha$.

Definition 2.3 [Bounded semantics with a loop]
Let $w$ be a finite word $w = w(0)w(1)\ldots w(l)\ldots w(k)$ with the loop from $k$ to $l$ and $\alpha$ be LTL formula. Then $w$ satisfies $\alpha$ ($w \models_L \alpha$) iff $w' \models \alpha$, where
$$w' = w(0)w(1)\ldots w(l)\ldots w(k)w(l)\ldots w(k)w(l)\ldots w(k)\ldots$$ is an infinite unwinding of $w$.

3 Bounded Model Checking Problems

We use a transition system called Kripke structure to capture the behaviour of a reactive system. A Kripke structure is a tuple $\langle S, s_0, R, L \rangle$, where $S$ is a finite set of states, $s_0 \in S$ is an initial state, $R \subseteq S \times S$ is a total transition relation, and $L : S \to 2^{AP}$ is a labeling function assigning each state a set of atomic propositions (that hold in this state).

A run of a Kripke structure $K$ is an infinite word $w$ such that there exists an infinite sequence $s_0, s_1, s_2, \ldots, (s_i, s_{i+1}) \in R$, $w(i) = L(s_i)$. The model checking problem is to determine for a Kripke structure $K$ and a temporal formula $\alpha$ whether the system satisfies the formula, i.e. whether for every run $w$ of $K$ holds $w \models \alpha$.

A length-bounded run (by $k$) of a Kripke structure $K$ is a finite word $w$ such that there exists a sequence $s = s_0, s_1, \ldots, s_k$, $(s_i, s_{i+1}) \in R$, $w(i) = L(s_i)$. For all transitions $(s_k, s_l) \in R$ such that $s_l$ occurs on $s$ we have different runs with loop from $k$ to $l$ (denoted by $w_{k,l}^l$). If there is no such transition the run is without a loop (denoted by $w_k$).

The length-bounded model checking problem is to determine for a Kripke structure $K$, a temporal formula $\alpha$, and a bound $k$ whether the bounded system satisfies the formula, i.e. whether for every run of $K$ length-bounded by $k$ holds $w_k \models_B \alpha$ if $w_k$ is without a loop or $w_k^l \models_B \alpha$ if $w_k^l$ is with a loop.
Moreover, we introduce depth-bounded model checking. The restriction
is given on the depth of the searched state space, in contrast with length-
bounded model checking where we restrict the length of the runs.

The depth of a state $s$ is the length of a shortest path from $s_0$ to $s$. Length
of a path is the number of its constituent edges.

A depth-bounded run (by $k$) is an infinite word $w$ such that there exists
a sequence $s = s_0, s_1, s_2, \ldots, (s_i, s_{i+1}) \in R, w(i) = L(s_i)$ such that the depth
of all states in $s$ is less than or equal to $k$. The depth-bounded model checking
problem is to determine for a Kripke structure $K$, a temporal formula $\alpha$, and
a bound $k$ whether the bounded system satisfies the formula, i.e. whether for
every run depth-bounded by $k$ of $K$ holds $w_k \models \alpha$.

The model checking problem can be reduced to the non-emptiness problem
of a Büchi automaton [18]. We construct this automaton (product automaton)
as the (synchronous) product of a Kripke structure and a formula automaton
(never claim in SPIN).

In addition, we can view a Büchi automaton as an oriented graph with
one source node corresponding to the initial state. The nodes corresponding
to the accepting states are called accepting nodes.

Then solving the non-emptiness of an automaton is equivalent to searching
for a cycle reachable from the source node containing an accepting node
(accepting cycle) in the graph corresponding to the automaton. This cycle
together with a path from the source node to this cycle is called a counterex-
ample.

The bounded model checking problem is equivalent to checking whether
there exists a counterexample of the length less than or equal to the bound.
The restricted model checking problem (with restriction $k$) corresponds to
searching for a counterexample that consists only of the states with the depth
less than or equal to $k$. In other words, restricted model checking is equivalent
to checking whether there exists a counterexample in the graph induced by
the states with the depth less than or equal to $k$.

4 Model Checking Algorithms

We propose four distributed algorithms for bounded LTL model checking. The
algorithms are to be performed on a network of workstations where no global
information is directly accessible. Each computer has its name and it can
communicate directly with any other computer via message passing.

One of these algorithms (DEBMC) does not use the reduction to the non-
emptiness of a Büchi automaton. Therefore, we use the word state in the
meaning of a state of the Kripke structure induced by a verified system. In all
other algorithms state denotes a state of the product automaton of a system
and an LTL formula automaton.

DEBMC algorithm is distributed in a different way. For the other three
algorithms, we suppose that the set of states of the inspected product au-
automaton is divided into disjoint subsets. The distribution is determined by the globally known static function \textit{(owner)}, which assigns every state to a computer (each state is \textit{local} for its owner). Edges between states belonging to the different computers are called \textit{cross-edges}. Distributed termination detection or distributed synchronization is handled by a computer called the \textit{Manager}. It sends termination messages only if all computers are idle and there are no pending messages. Our termination detection is based on Mattern’s ring-based termination detection [17,15].

When we check whether there is a state in a nonempty queue, the head of the queue is read and deleted from the queue at the same time. Therefore, a state is never deleted from the queue by a separate command.

4.1 \textit{DEBMC (Distributed Explicit Bounded Model Checking)}

DEBMC algorithm for length-bounded model checking problem simulates explicitly symbolic (length-)bounded model checking algorithm [3]. The symbolic algorithm reduces the bounded model checking problem to the satisfiability of a Boolean formula. It translates the system and the LTL formula into the Boolean formula. This formula poses some constrains on the solution – counterexample. The solution must correspond to a legal run (it must obey the transitions of the system) and this run must satisfy the LTL formula. The SAT solver searches for a satisfying valuation in the space of all possible valuations of the Boolean variables.

Our algorithm searches through the space of all system runs with length equal to the bound and checks whether they satisfy the formula (according to the bounded LTL semantics). Note, that it can visit a state more than once – in the different runs. Bounded runs are examined in the depth first order using backtracking.

Distribution of DEBMC algorithm is straightforward. Each computer searches through and checks only a part of the space of all length-bounded runs. We partition all length-bounded runs into several disjunct slices and then assign these slices to the computers. Slices are determined in the following way. Runs belong to the same slice if and only if they meet a specific state in a given depth. Thus, a slice is completely determined by a state (we call it \textit{distributing state}) and a depth (we call it \textit{distributing depth}).

One distinguished computer (\textit{Manager}) generates the \textit{distributing queue} first. This queue contains states that determine slices of the state space as described in previous paragraph. In our pseudo-code all states have the same depth (\textit{distr_depth}) and the queue contains all states with this depth. Then the Manager sends the distributing states to the computers that are idle. This distribution does not require any further communication except for control messages. Moreover, no bounded run is checked more than once.

Each computer runs procedure \textit{Worker}. This procedure handles the variable \textit{satisfied}, which is set to true by the procedure \textit{Check_formula}. 
if and only if path satisfies the formula. This procedure simply follows the definition of bounded LTL semantics (see Section 2).

**Distributed DEBMC algorithm**

**proc** Manager

```
end := false;
genenerate distributing queue;
while not end or formula not satisfied do
    wait_for_message;
    if message = i is idle then
        if distr_state in distributing_queue then
            send(i, “check, distr_state, distr_depth”);
            if message = satisfied then report path; send(all, “terminate”); fi
        if distributing_queue is empty and there are no pending messages and all processes are idle
            report “no counterexample detected”;
            end := true; fi
    od
end
```

**proc** Worker

```
end := false;
send(Manager, “id is idle”);
while not end do
    wait_for_message;
    if message = check then
        filter := distr_state; filter_depth := distr_depth;
        path := Check(init, 0);
        if satisfied then send(Manager, “path satisfies formula”);
        else send(Manager, “id is idle”);
        fi
    fi
    if message = terminate then end := true; fi
od
end
```

**proc** Check(s, depth)

```
if depth = filter_depth and s = filter then return; fi
foreach successor t of s do
    add t to path;
    if path does not satisfy the formula then return; fi
    if depth = bound then Check_FORMULA(path);
    else Check(t, depth + 1);
    fi
    if satisfied then return path; fi
    remove t from path;
od
end
```
4.2 BNDFS (Bounded NDFS)

BNDFS consists of two parts – generation of the bounded state space and the accepting cycle detection within this bounded space.

The accepting cycle detection performed by BNDFS is achieved by efficient nested depth first search (NDFS) algorithm (for detailed description of this algorithm see [9]). Unfortunately, as depth first search is inherently sequential [16], this algorithm is believed to be difficult to distribute and to the best of our knowledge there is no suitable distribution of NDFS algorithm. Therefore, we run it on a single machine (Manager).

All our attempts to modify NDFS algorithm to compute length-bounded model checking problem in the linear time failed. Therefore, we propose an algorithm for depth-bounded model checking problem that uses unmodified NDFS.

We generate the bounded state space first and then run NDFS on it. For the state space generation we use bounded version of the algorithm described in [10]. This bounded state space construction uses distributed breadth first search (BFS). Each computer performs its own BFS procedure on its local states. Whenever a successor of a processed local state does not belong to this computer, the successor’s owner is requested to continue the search by a message.

Moreover, we assign each state a unique number and we compose a new graph for NDFS, whose nodes are state numbers, hashing thus the state space compactly. To avoid the name collisions, each computer assigns only numbers congruent with its id (which is a number) modulo number of the computers to the generated local states. Each computer sends its renamed nodes and their successors to the Manager.

For renaming of the nodes, we remember a fresh number that has not been assigned to any state yet in the variable counter. After a number has been assigned to a state, a new one is generated by adding number of all computers (n) to counter.

Distributed BNDFS algorithm

```plaintext
proc Visit_State(name[s], t)
  if t not visited in smaller or equal depth then
    if name[t] does not exist then name[t] := counter;
      counter := counter + n;
  fi
  if depth[t] < bound then add t to queue fi
  if Manager then add edge (name[s], name[t]) into graph;
    else send(Manager, "ADD_EDGE_INTO_GRAPH(name[s], name[t])");
  fi
end
```

8
**proc** BNDFS

if init is local then add init into queue; fi

counter := id;

while not finished do

while s in queue do

process messages;

foreach successor t of s do

if t is local then VISIT_STATE(name[s], t);

else send(Owner(t), "VISIT_STATE(name[s], t"));

fi

od;

od:

CHECK_END;

if Manager then NDFS(name[init]); fi

end

4.3 BNBFS (Bounded Nested BFS)

BNBFS solves the length-bounded model checking problem. This algorithm searches through the bounded state space using BFS. Every time it meets an accepting state it runs nested BFS. Nested BFS procedure searches for the accepting state from which it was initiated. If nested BFS succeeds then there is a reachable cycle containing an accepting state. Nested search is bounded as well. This bound depends on the depth in which the accepting state was found. The sum of this (nested) bound and the depth of an accepting state equals to the (global) bound.

The first BFS is distributed, but each computer performs nested BFS for each local accepting state on the whole state space sequentially.

To generate the bounded state space correctly we have to know the depth of each state we visit. Unlike the sequential BFS algorithm, distributed BFS can visit (and process) a state more than once. This is due to the fact that we can visit a state in a depth greater than the actual depth sooner than it is visited in its actual depth. It is impossible to determine the actual depth of a state before the computation is finished. Therefore, we remember for each state the minimal depth in which it was visited. When an already visited state is visited in a smaller depth, it is processed again.

Nested BFS procedure runs sequentially and thus it knows exactly the depth of each state in the queue. Therefore, we visit each state only once and no extra memory is needed to remember the depth.

Termination detection is handled by the Manager. It sends termination messages only if BFS queues on all computers are empty (all computers are idle) and there are no pending messages or if a computer detects an accepting cycle.
Distributed BNBFS algorithm

```plaintext
proc BNBFS
    if init is local then add init into queue; fi
    while not finished do
        while s in queue do
            process messages;
            foreach successor t of s do
                if t is local then VISIT_STATE(t); else send(Owner(t), "VISIT_STATE(t)"); fi
            od:
        od:
        CHECK_END;
    od
    report "no counterexample detected";
end

proc VISIT_STATE(t)
    if t not visited in smaller or equal depth then
        if accepting(t) then NESTED_BFS(t) fi
        if depth[t] < bound then add t to queue fi
    fi
end

proc NESTED_BFS(g)
    goal := g;
    steps := bound - depth of g;
    add g to queue;
    while s in queue do
        foreach successor t of s do
            if t not visited in nested_BFS then
                if t = goal then report counterexample;
                send(all, "terminate") fi
                if local depth of t < steps then add t to queue fi
            fi
        od od
end
```

4.4 BSIMSYM (Bounded Simulation of Symbolic Algorithm)

BSIMSYM is a modification of the algorithm simulating the fixpoint computation of the set of reachable accepting states which can be reached from themselves. This set-based algorithm was proposed in [12] and it was used for distributed explicit model checking in [8]. The algorithm repeatedly performs two phases – reachability (generation of all reachable states from a given set of states), and elimination (removing of the states that are not on a cycle from the set of states). Unlike general algorithm we perform only bounded reachability. Therefore, our algorithm solves depth-bounded model checking problem. Computation during both phases is distributed. However, computers have to synchronize after each phase.
In order to bound the reachability we keep the information about the depth of each state. This value is computed during the first run of the reachability procedure. After that, reachability procedure makes use of this depth values.

Moreover, we remember a number of ingoing edges for each state. This value is utilized during the elimination phase. We reiteratively eliminate states without ingoing edges and decrease the number of ingoing edges of their successors.

A state can be visited and processed more than once during the reachability phase (in the different depths). Thus, we have to remember whether we have already counted in its outgoing edges. It is necessary for the correctness of the elimination to count in each edge only once. We remember this information in the variable \texttt{done}.

Counterexamples have to be extracted by a special double BFS procedure. Firstly, it searches for a path from the initial state to an accepting state contained in the computed set. Then the procedure searches for a cycle from this accepting state to itself.

Modifications making general algorithm [8] bounded are marked by comments in the pseudo-code.

**Distributed BSIMSYM algorithm**

```plaintext
proc BSIMSYM
    if init is local then add init into queue;
        add init into S; fi
    Reachability;
    while continue do
        Reset;
        Reachability;
        Elimination;
        Count_size; od
    if global_Ssize > 0 then generate and report counterexample;
        else report "no counterexample detected"; fi
end

proc Reset
    local_Ssize := 0;
    foreach s in S do
        if accepting(s) then local_Ssize := local_Ssize + 1;
            add s into queue;
            add s into L;
        else remove s from S; fi
    od
    foreach s do in_edges[s] := 0; done[s] := false; od
end
```

// BOUNDED
**proc** Reachability

```plaintext
while not finished do
  while s in queue do
    process_messages;
    foreach successor t of s do
      if t is local then VISIT_STATE(t, done[t]);
        else send(Owner(t), "VISIT_STATE(t, done[t])");
    fi
    od
  done[s] := true; // BOUNDED
od

Synchronization; od
```

**proc** Elimination

```plaintext
while not finished do
  while s in L do
    process_messages;
    remove s from S; local_Ssize := local_Ssize - 1;
    foreach successor t of s do
      if t is local then ELIMINATE_STATE(t);
        else send(Owner(t), "ELIMINATE_STATE(t)"); fi
    od
  od
Synchronization; od
```

**proc** Count_Size

```plaintext
if Manager then sum up local_Ssize from all workstations;
  if global_Ssize = old_global_Ssize then send(all, "terminate");
    else send(all, "continue"); fi
  else send(Manager, local_Ssize);
        wait_for_message;
  fi
end
```

**proc** VISIT_STATE(t, done)

```plaintext
if t not visited in smaller or equal depth then // BOUNDED
  if t not in S then add t into S;
    local_Ssize := local_Ssize + 1; fi
  if depth[t] < bound then add t to queue fi fi // BOUNDED
  if ¬done then if in_edges[t] = 0 then remove t from L; fi
    in_edges[t] := in_edges[t] + 1; fi // BOUNDED
end
```

**proc** ELIMINATE_STATE(t)

```plaintext
in_edges[t] := in_edges[t] - 1;
if in_edges[t] = 0 then add t to L; fi
end
```
5 Comparison of the Algorithms

We have implemented the algorithms proposed in Section 4. The implementation has been done in C++ using STL and the experiments have been performed on a cluster of ten 366 MHz Pentium (750 AMD Duron) PC Linux workstations with 256+128 Mbytes of RAM interconnected with 100Mbps Ethernet and using Message Passing Interface (MPI) library. The main objective was to compare our algorithms according to their execution time with respect to the size of the input. Measured time does not include I/O operations. The examples either do not contain an error or the algorithm has the same time complexity for examples with and without an error (not “on-the-fly” algorithm). [13] contains more detailed experimental evaluation of the algorithms as well as arguments for their theoretical time and space complexities.

5.1 DEBMC (Distributed Explicit Bounded Model Checking)

DEBMC algorithm has exponential theoretical time complexity. Experimental results confirmed this estimation (Figure 1). This algorithm computes in the “on-the-fly” fashion and there are many possibilities of improving this algorithm, such as extracting more information from the system or employing stuttering in order to cut off non-perspective branches of the searched space. However, even in spite of these facts, DEBMC cannot compete with the other algorithms.

DEBMC beats all the other algorithms in the space complexity, which depends only on the bound and thus it is logarithmic with respect to the size of the searched state space.

Distribution of this algorithm is straightforward and needs no communication (except for control communication with the Manager). The whole computation is distributed over the cluster and scales well (Figure 2). Computers send only a small, in practice constant, number of control messages.

DEBMC algorithm finds a shortest counterexample. It returns the path which satisfies the (negated) formula.

5.2 BNDFS (Bounded NDFS)

BNDFS algorithm has the best time complexity. As the only one, BNDFS algorithm runs in the time linear to the size of the state space – number of the product automaton states (Figure 1). Its main drawback is that it is necessary to compute the whole bounded state space to find a counterexample.

Another drawback of this algorithm is that only the state space generation phase is distributed and we have to remember the whole renumbered state space in the memory of a single computer. But in spite of these shortcomings we propose BNDFS as the best distributed explicit bounded model checking algorithm.
Fig. 1. Time Complexity of the Algorithms (10 Computers)

Fig. 2. Scalability of the Algorithms
Counterexamples found by BNDFS algorithm are usually smaller than those found by unbounded NDFS. However, they are mostly much longer than the bound. It shows that this way of bounding the NDFS algorithm does not provide us with reasonably small counterexamples. Therefore, we implemented a supplementary algorithm for counterexample generation. This algorithm performs two BFS searches. The first one starts in the initial state and it searches for the accepting state in which successful nested DFS procedure was initialized. The second one starts in this accepting state and searches for a loop back into this state. We call this procedure GEN. For the comparison of the counterexample lengths see Table 2.

5.3 BNBFS (Bounded Nested BFS)

BNBFS gives the counterexamples of an optimal length. Moreover, this algorithm runs “on-the-fly”. Unfortunately, quadratic time complexity of BNBFS rules it out of the practical use for larger bounds and product automata without an accepting cycle. In these systems, the algorithm proved its quadratic behaviour even in spite of the “on-the-fly” computation.

Distribution does not help to decrease the memory requirements posed on the individual machines. Distribution helps to save the time. We search through the state space in parallel and each computer runs nested BFS only for its own accepting states. Indeed, BNBFS scaled well in our experiments (Figure 2).

5.4 BSIMSYM (Bounded Simulation of Symbolic Algorithm)

Theoretical worst case time complexity of BSIMSYM algorithm is quadratic. There are the systems for which the algorithm runs in quadratic time, but in the most cases, it has almost linear time complexity (Figure 1). However, BNDFS algorithm outperforms BSIMSYM algorithm on all examples. The main advantage of BSIMSYM is its complete distribution of the state space and work. Therefore, BSIMSYM is the most memory efficient algorithm.

Number of sent messages corresponds to the time complexity. If fixpoint is not reached, we perform distributed reachability and elimination again. During this new iteration the algorithm sends again a message for each cross-edge it meets.

5.5 Comparison

An overview of the algorithms is given in the Table 1. Moreover, we compare the following characteristics of the algorithms:

**Space** — size of the data structures employed by the algorithm. For the analysis we use the following notation: $n$ — size of the bounded state space, $k$ — number of the computers. The space analysis is not based on experiments, comparison is made on basis of the theoretical consideration.
**Messages** — measured communication complexity (number of sent messages) with respect to the size of the bounded state space.

**Incrementality** — we can reuse already generated state space in BNDFS algorithm when we run the algorithm with a greater bound. Other algorithms have to recompute everything from the scratch.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DEBMC</td>
<td>Length Exp</td>
<td>Yes</td>
<td>bound</td>
<td>Time</td>
</tr>
<tr>
<td>BNDFS</td>
<td>Depth</td>
<td>Lin</td>
<td>No</td>
<td>n Gen</td>
</tr>
<tr>
<td>BNBFS</td>
<td>Length</td>
<td>Quad</td>
<td>Yes</td>
<td>n Time</td>
</tr>
<tr>
<td>BSIMSYM</td>
<td>Depth</td>
<td>Mostly Lin</td>
<td>No</td>
<td>n/k All</td>
</tr>
</tbody>
</table>

Table 1
Comparison of the Algorithms.

<table>
<thead>
<tr>
<th>Example</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not &quot;fair&quot; MC</td>
<td></td>
</tr>
<tr>
<td>Ring 5, Invariance</td>
<td>NDFS GEN</td>
</tr>
<tr>
<td></td>
<td>14423   50</td>
</tr>
<tr>
<td></td>
<td>BNDFS GEN</td>
</tr>
<tr>
<td></td>
<td>BSIMSYM</td>
</tr>
<tr>
<td></td>
<td>BNBFS</td>
</tr>
<tr>
<td>Ring 5, Justice</td>
<td>NDFS GEN</td>
</tr>
<tr>
<td></td>
<td>895      8</td>
</tr>
<tr>
<td></td>
<td>BNDFS GEN</td>
</tr>
<tr>
<td></td>
<td>BSIMSYM</td>
</tr>
<tr>
<td>Com. protocol, Reply</td>
<td>NDFS GEN</td>
</tr>
<tr>
<td></td>
<td>73       20</td>
</tr>
<tr>
<td></td>
<td>BNDFS GEN</td>
</tr>
<tr>
<td></td>
<td>BSIMSYM</td>
</tr>
<tr>
<td>&quot;Fair&quot; MC</td>
<td></td>
</tr>
<tr>
<td>Ring 5, Invariance</td>
<td>NDFS GEN</td>
</tr>
<tr>
<td></td>
<td>23 567  63</td>
</tr>
<tr>
<td></td>
<td>BNDFS GEN</td>
</tr>
<tr>
<td></td>
<td>BSIMSYM</td>
</tr>
<tr>
<td></td>
<td>BNBFS</td>
</tr>
<tr>
<td>Ring 5, Liveness</td>
<td>NDFS GEN</td>
</tr>
<tr>
<td></td>
<td>3 116    72</td>
</tr>
<tr>
<td></td>
<td>BNDFS GEN</td>
</tr>
<tr>
<td>Elevator, Response 1</td>
<td>NDFS GEN</td>
</tr>
<tr>
<td></td>
<td>379      42</td>
</tr>
<tr>
<td></td>
<td>BNDFS GEN</td>
</tr>
<tr>
<td></td>
<td>BSIMSYM</td>
</tr>
<tr>
<td>Com. protocol, Reply</td>
<td>NDFS GEN</td>
</tr>
<tr>
<td></td>
<td>149      19</td>
</tr>
<tr>
<td></td>
<td>BNDFS GEN</td>
</tr>
<tr>
<td></td>
<td>BSIMSYM</td>
</tr>
<tr>
<td></td>
<td>BNBFS</td>
</tr>
</tbody>
</table>

Table 2
Lengths of the Counterexamples Generated by the Algorithms

The length of the counterexample generated by a model checker has a great impact on the debugging process. The shorter the counterexample is, the easier can the designer comprehend the error. In our experiments, counterexamples generated by all algorithms (or by an extra counterexample generation
procedure GEN in case of NDFS, BNDFS, and BSIMSYM) had comparable lengths (see Table 2). However, this result is rather surprising and we stress that more experiments on various systems should be carried out to corroborate or refute it.

6 Conclusions and Future Work

We have proposed several approaches related to the ideas of bounded model checking for verification of the systems with explicitly represented state space. These approaches fight the state space explosion by posing some limits on the searched state space. This led to a proposal of algorithms for explicit bounded LTL model checking. In order to speed up the verification, all algorithms make use of the distributed environment – network of workstations.

Prototype implementation of these algorithms allowed to compare their practical behaviour. None of them proved to beat all the others in every situation. BNDFS algorithm with extra counterexample generation behaved best in most characteristics and therefore we propose it as the best explicit bounded model checking algorithm. Unfortunately, we were not able to propose any linear time “on-the-fly” length-bounded MC algorithm.

The results suggest following directions for the future work. Embedding of our algorithms into existing verification tool, (e.g. SPIN), could enable to test them more thoroughly in real life situations. Another way is to develop brand new tool supporting distributed verification and to incorporate our algorithms into it. Here, the comparison of different algorithms would be fair in the sense that differences in the implementation would be eliminated.

References


