Distributed Symbolic Bounded Property Checking

Pradeep K. Nalla, Roland J. Weiss, Prakash Peranandam, Jürgen Ruf, Thomas Kropf, Wolfgang Rosenstiel

Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
Sand 13, 72076 Tübingen, Germany

Abstract

In this paper we describe an algorithm for distributed, BDD-based bounded property checking and its implementation in the verification tool SymC. The distributed algorithm verifies larger models and returns results faster than the sequential version. The core algorithm distributes partitions of the state set to computation nodes after reaching a threshold size. The nodes proceed with image computation on the nodes asynchronously. The main scalability problem of this scheme is the overlap of state set partitions. We present static and dynamic overlap reduction techniques.

Keywords: Verification, bounded model checking, property checking, binary decision diagrams, parallelization.

1 Introduction

Although symbolic representations of state spaces [8] based on Binary Decision Diagrams (BDDs) [7] and bounded model checking (BMC) [3] have dramatically increased the design sizes that can be handled by verification tools, research in model checking techniques still concentrates on enabling faster verification of larger models. Large designs cause memory overflow during exploration of the state space, the dreaded state space explosion. There are

1 This work has been funded in part by the German Research Council (DFG) within projects GRASP and KOMFORT and by the BMBF and edacentrum within project FEST.
several proposed solutions to deal with the immense memory requirements of BDDs. One proposal is to partition BDDs [30] into two or more pieces and handle them separately during further traversal. The traversal of the partitions can be done sequentially [10] or in parallel [14].

In [28], a combination of on-the-fly [12] and bounded model checking is presented, which is implemented in the tool SymC. The checking algorithm traverses the product automaton of model and property until it either detects a validation or a violation of the property, or the explicit or implicit time bound is reached. Only the frontier set is kept in memory, i.e. no fix-point iterations are performed. This approach performs well for certain classes of models and properties, but the sequential version also faces memory exhaustion for large model, e.g. for some of the ISCAS89 examples. This fact motivated the parallelization of the proof algorithm which we present here.

The paper is organized as follows. The next section discusses related work and our contributions. Section 3 summarizes symbolic bounded property checking, followed by a description of the distributed algorithm. Then, we present our static and dynamic methods for overlap reduction. Section 6 gives experimental results. Finally, we conclude and mention future work.

2 Related Work

2.1 Partitioning

Many approaches for decomposing Boolean functions represented as BDDs exist in literature. For distributed verification [16,14] splitting algorithms aim at creating balanced partitions. However, similar approaches exist in sequential verification methodologies [11,10]. The main distinguishing feature of these algorithms is the employed cost function for selecting the splitting variable. The cost functions typically take into account the achieved memory reduction, the amount of sharing between the cofactors, and the memory balance of the cofactors. Also, the CUDD package [31] contains various decomposition algorithms, producing both balanced and unbalanced partitions. Furthermore, decomposition techniques allow representing the same function with multiple BDDs but requiring less memory [20,19]. The image computation algorithms have to be updated for these techniques. The more complex operations are set off by the reduced peak memory requirements of the BDDs [30]. As shown in [4], the reduction can even be exponential. Finally, dense under-approximations [26,25] try to reduce the memory requirements of the BDD but still capture a large percentage of the state space. These algorithms are of minor interest for state set distribution as they result in unbalanced subsets.
None of the proposed heuristics consider subsequent state overlap. However, similar efforts are undertaken for model checkers with an explicit state graph representation [21,5]. They apply graph algorithms that heuristically try to find partitions with few crossover transitions in order to reduce the communication effort between processes. In [15], the authors investigate state space distribution in the context of model checking Petri nets, also employing an explicit representation. These approaches cannot be directly applied to symbolic representations.

2.2 Distributed model checking

The state space explosion problem in model checking has raised interest in handling this problem by adjusting the algorithms for distributed environments recently. This includes both explicit [32,5,17,6] and symbolic [14,16,2,13,18] model checking methodologies.

The group at Haifa also works on the parallelization of BDD-based verification algorithms. At the core, they create $k$ slices of the current state set and distribute these slices to $k$ cluster machines. They use the slicing technology from [20], but with an enhanced cost function for selecting slicing variables [16]. States are classified as owned and non-owned. After every image computation step the non-owned states are distributed to the owning nodes. In [16] load balancing is achieved by adjusting the slices if the initial balance is lost. In [14] they try to keep only as many nodes busy as necessary by splitting and joining BDDs on demand. The exchange of non-owned states after every step makes their algorithm mainly synchronous. In [14,16] reachability is computed with fix-point iterations, in [2] regular expressions are used to indicate illegal behavior and $\mu$-calculus formulas are checked in [13]. Our approach checks time-bounded properties specified in PSL (Property Specification Language) [1] or FLTL (Finite Linear Time Temporal Logic) [27] without fixpoint iterations.

2.3 Contributions

Synchronous schemes for parallelizing BDD-based verification algorithms reduce the potential speedup because processes are kept waiting for others to complete. Up to now, no successful asynchronous BDD-based verification algorithms have been proposed.

The main contribution of our approach is such an asynchronous distributed algorithm. This algorithm becomes feasible only when the shared states due to crossover transitions are reduced to avoid duplicate work. We present algorithms for static and dynamic overlap reduction.
3 Sequential Symbolic Bounded Property Checking

The formal verification algorithms in [28,22] combine bounded property checking and symbolic traversal. The temporal logic formulas are converted to special finite state machines called Accept-Reject automata (AR-automata) [27]. AR-automata allow finding violations or validations of properties on finite sequences, thus they are well suited for bounded property checking. The checking algorithm manipulates both the system description and the AR-automata represented as BDDs. In order to avoid the construction of the complete transition relation, a set of conjunctively partitioned transition relations is built, which is used for early quantification [9]. The algorithms have been implemented in the tool SymC, whose general operation is shown in Fig. 1.

An iteration of the sequential verification algorithm works in two steps. First, the successor states of the AR-automata are computed and the termination condition is checked. If the termination condition is not satisfied, image computation is performed on the system in the second step. During image computation the conjunction of all partitions is built on-the-fly to obtain the successor state set. Like bounded model checking [3], this property checking algorithm does not traverse the state space exhaustively but examines all reachable states within a given time bound.

A central optimization technique for the algorithm is state set splitting. Whenever a threshold for the size of the BDD representing the current state set is reached, the set is split into disjoint parts and the algorithm continues working on these subsets in a divide-and-conquer manner.

The sequential verification algorithm continues with one of the subsets and stacks the others. This can happen recursively. Traversal proceeds on the current subset until the time bound is reached or the termination condition is satisfied. Termination stops the verification with finding either a validation or a violation of the property. Otherwise, the process is repeated for all stacked subsets. The termination condition differs if one checks the property on all paths, i.e. universal quantification, or on one path, i.e. existential quantification. Informally, the sequential termination condition is defined as follows:

**Universal** If one reject state is detected in the current state set, a violation
of the property is found. If all states in the current state set are accepting states, a validation of the property is found. Otherwise, the property is still pending.

**Existential** If one accept state is detected in the current state set, a validation of the property is found. If all states in the current state set are rejecting states, a violation of the property is found. Otherwise, the property is still pending.

### 4 Parallelization of Bounded Property Checking

The distributed checking algorithm is composed of an initial sequential stage and a subsequent parallel stage. First, the transition relation is created on all $k$ computation nodes and state space traversal proceeds sequentially on one node until a threshold limit on the BDD size triggers state set distribution. The splitting into $k$ subsets is already performed in parallel and every node is responsible for getting its own disjoint part of the whole state set. The nodes start state space traversal independently on these subsets. The termination condition stays the same, however the nodes have to communicate their local results in order to allow testing termination conditions that depend on all states.

This simple scheme fails to provide significant speedups on many models because of crossover transitions. These transitions start in a state of the current subset but lead to a state that is already present in one of the other state subsets. We call this phenomenon state set overlap, or just overlap. Of course, image computation for overlapping states is performed redundantly. As image computation is one of the key components of formal verification tools, redundancy of such a component badly affects the time and memory requirements of the whole verification process. Thus, optimizing the distributed algorithm concentrates on reducing the overlap (see section 5).

#### 4.1 State set distribution

Splitting the state set into $k$ parts for subsequent traversal in parallel is a costly operation. Therefore, we already perform it in parallel. For simplicity we assume that $k = 2^n, n \in \mathbb{N}$. Basically, once the first node dumps its state set to disk, all other nodes pick up the dumped set after notification. Then, each node splits the set into two parts and depending on its rank, a number identifying every node, it drops one part and continues splitting on the other part recursively until only its own subset remains. The algorithm is illustrated in Fig. 2.
// get subset i of k slices from state set S
getSubset(in: S, k, i; out: S_i)
S_i := S
for j := 1 .. log_2(k)
    split(S_i; g, h)
// skip h on odd bit, skip g on even bit
if i % 2 = 1 then S_i := g else S_i := h
i := i / 2 // get next bit
// split state set S into two slices g and h
split(in: S; out: g, h)

Fig. 2. Algorithm for state set distribution. The left hand side gives the distribution algorithms, an example application is shown on the right hand side.

4.2 State set overlap

After all nodes picked their state subsets, the nodes proceed with symbolic state space traversal. A very important observation is that after a few steps of traversal state overlap between network nodes may emerge.

Definition 1 Let S be a set represented using a BDD. Then \|S\| denotes the number of states in S, which is given by the number of maximal minterms of the BDD.

Definition 2 Let S be a nonempty set and S_1, \ldots, S_k \subseteq S with k \geq 2. Then we define the state overlap o_k \in [0, 1] of these partitions as:

\[
o_k = \frac{\sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \|S_i \cap S_j\|}{\|S\| \sum_{i=1}^{k-1} i}.
\] (1)

The overlap is thus the normalized average of states in the pairwise intersection of subset permutations. The sum in the denominator ranges from 1 to k – 1 because this yields the number of pairs S_i, S_j with i < j. An overlap of o_k = 0 corresponds to disjoint partitions and an overlap of o_k = 1 corresponds to partitions containing the same states.

5 Overlap Reduction

Boolean functions represent all the state sets and the transition relation in symbolic traversal. This representation can grow large if the sets to be represented are big, corresponding directly to more memory requirements. The Boolean functions are represented and manipulated using BDDs. The memory requirements |f| of a Boolean function f are defined as the number of its nodes. In order to reduce the memory requirements one can partition a Boolean function into smaller parts, whose union is the whole set.
Definition 3 Given a Boolean function $f : B^n \rightarrow B$, $f$ is partitioned into two functions $f_1$ and $f_2$ on a variable $v$ from the support set of $f$ with

$$f = f_1 \lor f_2 \text{ where } f_1 = v \land f_v, \quad f_2 = \bar{v} \land f_{ar{v}}.$$  \hspace{1cm} (2)

The splitting variable $v$ defines the partitioning of $f$ into $f_1$ and $f_2$. This splitting can be implemented easily with BDD operations. BDDs are compressed decision trees where common subtrees are joined. This causes significant sharing of nodes in a function’s representation. Thus, splitting a function $f$ into two functions $f_1$ and $f_2$ with a poor choice of $v$ may not necessarily reduce the memory requirements of the split functions and can result in $|f_1| \approx |f_2| \approx |f|$. In the following discourse, we identify a state set with its characteristic function represented as BDD.

5.1 Static overlap reduction

Overlap originates from states in different sets having transitions to the same next states. In order to minimize the overlap of splits, the selected splitting variable $v$ should not allow states that have common next states to be in different splits. In other words, $v$ should partition the states such that they have no common next states. However, in reality such a partitioning is not possible, but one can put some effort in selecting the splitting variable $v$ to minimize overlap. For finding a good splitting variable we statically analyze the design which is represented as finite state machine (FSM).

Definition 4 A FSM $\mathcal{A}$ is a 4-tuple $\mathcal{A} = (S, \Sigma, T, I)$, where $S = \{s_1, \ldots, s_n\}$ is a finite set of states encoded by state variables $e_1, \ldots, e_m$, $\Sigma$ is a finite input alphabet, $T \subseteq S \times \Sigma \times S$ is a transition relation represented with $T_1, \ldots, T_m$ partitions, and $I \subseteq S$ is the set of initial states.

The idea of selecting a good splitting variable $v$ relies on the conjunctively partitioned transition relation $T$ [9]. For every $i \in 1, \ldots, m$ a partition $T_i$ of
the transition relation corresponds to the truth value of next state variable \(e'_i\) such that \(\mathcal{T} = \bigwedge_{i=1}^{m} T_i\). We pick \(v\) from the set of state variables \(E = \{e_1, \ldots, e_m\}\). Fig. 3 (a) shows the best case where there is no overlap. This kind of situation is only possible if \(v\) will stick to its truth value in all further steps, i.e. partition \(T_v (T_i \text{ with } v = e_i)\) depends only on \(v\). Though this is the ideal case, we hardly have such situations in real designs. This means that \(v\) might change its truth value in future steps as its partition of the transition relation depends on more factors. The worst case of almost complete overlap can occur if \(T_v\) depends on input variables disjunctively only, as depicted by Fig. 3 (b). The common case lies in between these two extremes and happens when \(v\) depends on inputs conjunctively with other combinations of state variables, depicted in Fig. 3 (c). The algorithm \(\text{MinOverlap}\) pioneers in exploiting static information of the partitioned transition relation \(\mathcal{T}\) to find a good splitting variable \(v\).

In a pre-processing step, every state variable is assigned an influence and the variables are ordered decreasingly by their influence. The influence table maps state variables to their influence. Later, the splitting variable selection algorithm utilizes this information.

**Definition 5** Let \(l_1, l_2 \in \mathbb{N}\) be influence lookaheads. For a given FSM \(A\), the influence \(\Phi_{l_1, l_2}(e) \in [-1, 1]\) of a state variable \(e \in E\), with \(|E| = m\), is defined as

\[
\Phi_{l_1, l_2}(e) = \frac{|D^\uparrow(e, l_1)| - |D^\downarrow(e, l_2)|}{m}.
\]  

Set \(D^\uparrow(e, l_1)\) contains all state variables that get influenced by \(e\) in \(l_1\) steps, and set \(D^\downarrow(e, l_2)\) contains all state variables that influence \(e\) in \(l_2\) steps. These sets are determined iteratively starting with \(l_1 = 1\) and \(l_2 = 1\). Each \(T_i\) directly corresponds to the truth value of the next state variable \(e'_i\), so we compute these sets by walking all \(T_i\) and \(e_i\). For \(D^\uparrow(e, 1)\), we count the partitions \(T_i\) that contain \(e\), whereas for \(D^\downarrow(e, 1)\) we count the state variables in the support of \(T_i\).

The basic assumption of the \(\text{MinOverlap}\) algorithm is that splitting on a variable \(v\) with high influence will lead to fewer cross transitions between the resulting partitions, because the value of \(\Phi_{l_1, l_2}(v)\) next state variables depends on \(v\). Of course, there are other factors determining the values of these next state variables, weakening our assumption. Our algorithm works well if the partitioned transition relations \(T_i\) depend on conjunctively connected variables only. It degrades if the \(T_i\) depend on disjunctively connected variables where at least one disjunct contains only input variables. However, it is computationally expensive to analyze all Boolean connectives of the clauses of every \(T_i\).
The actual $MinOverlap$ algorithm picks a viable state variable for splitting. The state variables are categorized based on their influence and put into different sets. We start with the set containing variables with a high influence and check them against a balancing condition. Alongside, we compute the cost of these variables with the cost function from [16] that consists of a redundancy and a reduction factor. If none of the examined variables satisfied the balancing condition, the variable with minimal cost is selected. Fig. 4 gives the pseudo code for $MinOverlap$.

```c
// $S$ is the current state set
// $S_1$ and $S_2$ are the resulting partitions
// $\Phi$ is the influence table
// $\delta$ is the memory balance factor
// $\alpha$ is the weight for the cost function
split(in: $S$, $\Phi$, $\delta$, $\alpha$; out: $S_1$, $S_2$)

bestCost := $\Phi$.top()
minCost := cost($S$, bestCost, $\delta$, $\alpha$)
while $C$ = getCandidateSet($\Phi$) $\land$ $C \neq \emptyset$
for all $w \in C$
if max($|S_w|$, $|S_w'|$) $\leq \delta |S|$ then
$v := w; \text{goto do_split}$
else
thisCost := cost($S$, $v$, $\alpha$)
if thisCost $< \text{minCost}$ then
minCost := thisCost; bestCost := $w$
$v := \text{bestCost}$
do_split: $S_1 := S_v$; $S_2 := S_v'$
```

Fig. 4. State set splitting with the $MinOverlap$ algorithm.

5.2 Dynamic overlap reduction

Initially, the overlap between state sets of network nodes is reduced by applying the $MinOverlap$ algorithm. However, in general the overlap may still pursue after a few steps of state space traversal. In order to further confine the overlap we perform dynamic overlap reduction. This is a methodology where we allow overlap to some extent and heuristically select a time frame to remove it periodically. We perform overlap removal after state set distribution (see section 4.1). This method is iteratively performed either throughout the verification process or up to $n$ times. An extra node called coordinator organizes the communication between the nodes and performs dynamic removal of state overlap. The overlap removal algorithm for each node works in three steps:

(i) Upon reaching a reduction time point the node dumps its current state

2 The state set distribution time point and the reduction period determine the reduction time points.
set onto the network drive and sends a message to the coordinator.

(ii) The coordinator removes the overlap of the node with respect to the
already visited state space by other nodes at this time point, and updates
the history of the visited state space. Then it informs the corresponding
node to proceed with the reduced state space by dumping the trimmed
state set.

(iii) Finally, if all nodes passed a reduction time point, the coordinator re-
moves the state space history of that time point.

Fig. 5 delineates the usage of overlap reduction in the main computation
loop of the symbolic simulation algorithm in the parallel stage. We have to
check the termination condition locally, i.e. only in the current subset (line
10), and globally, which requires communication with the other nodes (line
8). For example, in order to show an universal validation, all nodes have to
finish in accept states locally, which can only be checked globally.

```
// S is the set of initial states
// t is the checking time bound
// p is the period of steps at which overlap removal is performed
// n is the overlap removal limit, 0 indicates continuous reduction
simulate(in: S, t, p, n)
 reduction_limit := 0; reduction_step := 0
if n > 0 then tillEnd := false else tillEnd := true
while iteration < t
  checkTerminationConditionGlobally()
  S := image_AR(S) // Compute image of AR-automata.
  checkTerminationConditionLocally(S);
  S := image_T(S) // Compute image of the system.
  if (reduction_limit < n) ∨ tillEnd then
    reduction_step++
    if reduction_step = p then
      S := removeOverlap(S)
      reduction_step := 0; reduction_limit++
```

Fig. 5. Main computation loop for state overlap removal.

The main advantage of our dynamic reduction method is that nodes do
not have to wait for slow nodes. After dumping their current state set, faster
nodes can continue to traverse the product automaton. Therefore, we achieve
asynchronous overlap removal between network nodes. Although nodes have
to wait for the coordinator to update their state set, this time is not significant
compared to the time spent on image computation.

An interesting side effect of our asynchronous methodology is the resulting
natural load balancing. The very last node that reaches a reduction time
point gets its overlap removed with respect to all other nodes. So this last
node has no states in common with the other nodes at this reduction step.
Our experiments state that usually the last node after overlap removal has
the smallest subset. This in turn means faster image computation enabling
this node to reach the forthcoming reduction time point faster. Hence, at
that reduction time point this particular node will arrive earlier than other
nodes, and therefore continues with a larger state set. This process alternates
among the nodes accordingly depending on the weight of image computation,
resulting in natural load balancing between the network nodes.

For some models, the overlap is so high that the late nodes become empty
after overlap removal. This special situation is handled by state set sharing
with the following node that reaches any reduction time point.

6 Experimental Results

We performed our experiments on the Kepler cluster at the University of
Tuebingen\(^3\). This cluster contains 98 computing nodes, each consisting of
dual 650 MHz Pentium-III processors with 1 GB of shared memory (512 MB
for each processor). We conducted our experiments on some of the circuits
from the ISCAS89 benchmarks and a model of a holonic production system
[29]. All experiments were performed with dynamic variable ordering disabled
in the BDD package. For circuits from the ISCAS89 benchmarks we check for
reachability of a state at high hamming distance from the initial states (see
equation 4) along with properties from [2]. In the holonic production system
we check for consumption of a workpiece (see equation 5). All properties are
checked universally. The properties written in FLTL look like this, where
\( b > 0 \) are explicit time bounds on the properties:

\[
G[b]  ! (s1512.\text{start} & s1512.\text{video} \ & \ldots & \ s1512.I1733) \quad (4)
\]

\[
F[b] \ OutBuffer.s_{\text{consume}} \quad (5)
\]

6.1 Static overlap reduction

In this part we concentrate on comparing the static overlap reduction heuristic \( \text{MinOverlap} \) to an altered version of the slicing heuristic from [16]
labeled \( \text{EqualDist} \), and the variable disjunction decomposition algorithm from the
CUDD package [31] labeled as \( \text{VarDisj} \). The \( \text{MinOverlap} \) algorithm is de-
noted by the influence \( \Phi_{l_1,l_2} \) used for ordering the state variables. In these
experiments, we use a balancing condition of \( \max(|f_1|,|f_2|) \leq \frac{2}{3}|f| \) for the
\( \text{MinOverlap} \) algorithm. The results are shown in Fig. 6.

\(^3\) http://kepler.sfb382-zdv.uni-tuebingen.de
Fig. 6. Comparison of \textit{MinOverlap} with other heuristics. The first column lists the design, followed by the number of processors used, the number of state variables and the splitting threshold. The second column indicates the splitting algorithm. The third column gives at each splitting recursion level the indexes of the selected splitting variables. The CUDD package identifies variables by index. Then the fourth column shows the overlap at different iteration steps. The fifth and sixth columns list the average splitting time \( s_t \), and the total verification time \( v_t \) (or a memory overflow is indicated by \#, followed by the maximum number of steps), respectively. The splitting time corresponds to the time spent in algorithm \textit{split} as described in figure 4.

\begin{table}[h]
\begin{tabular}{|c|c|c|c|c|c|}
\hline
Design & Split. alg. & Recursion level : Splitting vars. & Step : \( o_k \cdot 100 \) & \( s_t \) & \( v_t \) \\
\hline
s1269 & \( \Phi_{1,1} \) & 1:32 & 1:31.5 & 0.11 & 214.13 (0.55) \\
#2 & \( \Phi_{1,0} \) & 1:32 & 1:31.5 & 0.11 & 217.5 (0.51) \\
37 & EqualDist & 1:0 & 1:58.6 & 0.36 & 236.7 (0.28) \\
5000 & VarDisj & 1:14 & 1:57.4 & 0.15 & 371.2 (0.25) \\
\hline
s1512 & \( \Phi_{5,1} \) & 1:96 & 5:64.8 / 10:89.3 & 0.15 & 3204.85 (4.52) \\
#2 & \( \Phi_{1,0} \) & 1:10 & 5:89.3 / 10:91.8 & 0.11 & 3330.31 (4.4) \\
57 & EqualDist & 1:10 & 5:89.3 / 10:91.8 & 0.87 & 3244.9 (3.72) \\
10000 & VarDisj & 1:10 & 5:89.3 / 10:91.8 & 0.38 & 3312.75 (3.75) \\
\hline
s1269 & \( \Phi_{1,1} \) & 1:32 / 2:34 / 3:36 & 1:9.1 & 0.68 & 69.9 (0.55) \\
#8 & \( \Phi_{1,0} \) & 1:32 / 2:34 / 3:40,36 & 1:9.1 & 0.68 & 69.43 (0.52) \\
37 & EqualDist & 1:0 / 2:42,6 / 3:12,8,40 & 1:12.7 & 0.55 & 58.8 (0.29) \\
5000 & VarDisj & 1:14 / 2:12 / 3:10,26 & NA & 0.26 & 58.4 (0.27) \\
\hline
s1512 & \( \Phi_{5,1} \) & 1:96 / 2:98 / 3:100,10 & 10:54.6 / 15:68.3 & 0.23 & 2891.7 (4.6) \\
#8 & \( \Phi_{1,0} \) & 1:10 / 2:12 / 3:14 & 10:76.4 / 15:94.2 & 0.15 & 2993.0 (4.4) \\
57 & EqualDist & 1:10 / 2:96,12 / 3:12,14,94 & 10:76.6 / 15:94.2 & 1.5 & 2988.9 (3.7) \\
10000 & VarDisj & 1:10 / 2:12,94,96 / 3:12,96,14 & 10:76.4 / 15:94.2 & 0.71 & 3029.2 (3.7) \\
\hline
nh2 & \( \Phi_{1,1} \) & 1:14 / 2:48,16 / 3:10,18,52 & 60:11.7 / 100:25.4 & 0.65 & \#481 (7.12) \\
#8 & \( \Phi_{1,0} \) & 1:14 / 2:54.16 / 3:48,18,10 & 60:12.2 / 100:26.6 & 0.51 & \#272 (7.0) \\
118 & EqualDist & 1:4 / 2:58,176, / 3:24,134,54,40 & 60:22.1 / 100:40.8 & 13.4 & \#151 (6.04) \\
50000 & VarDisj & 1:4 / 2:58,40 / 3:44,54,18,176 & 60:20.9 / 100:41.0 & 13.3 & \#145 (6.07) \\
\hline
\end{tabular}
\end{table}

**Discussion:** The preprocessing step of the \textit{MinOverlap} algorithm does not require a significant amount of time, in all experiments it consumed less than 1\% of the verification time. For two processor, design \textit{s1269} shows a significant reduction in overlap by selecting high influence variables and hence a gain in overall verification time can be observed. Both \textit{MinOverlap} and \textit{EqualDist} picked high influence variables, but only \textit{MinOverlap} reduced the overlap significantly. This is due to the influence lookahead condition explained in Section 5.1. The influence stays positive for \textit{MinOverlap} and becomes negative for \textit{EqualDist} with \( \Phi_{1,1} \).

For eight processors, design \textit{s1269} has low overlap with all splitting algorithms. But the other two designs clearly show the benefit of applying \textit{MinOverlap}, both for designs with huge and moderate overlap. Design \textit{s1512} belongs to the category with huge overlap after a few steps. However, \textit{Min-}
Overlap with a lookahead of \( l_1 = 5 \) is able to significantly delay the occurrence of overlap and reduce the verification time. Nevertheless, this reveals that high influence variables can only help to reduce the overlap for a few steps but cannot avoid it beyond a limit, making dynamic removal techniques a must. The overlap in design \( nh2 \) increases much slower than in the other design, but its size leads to memory overflow. Again, \( \text{MinOverlap} \) is able to reduce the overlap, even after 100 steps. This allows the nodes to go a lot further without memory overflow.

6.2 Dynamic overlap reduction

First, we ran some of the larger designs in sequential SymC with all relevant optimizations switched on. The sequential algorithm splits the state set repeatedly upon reaching the threshold, whereas the parallel version does it only during state set distribution. The results are available in Fig. 7. For most of the designs the sequential algorithm cannot complete traversal due to memory overflow or time out problems.

<table>
<thead>
<tr>
<th>Design</th>
<th>Threshold</th>
<th>Influence</th>
<th>Time bound</th>
<th>Peak node count</th>
<th>( v_t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>s4863</td>
<td>20000</td>
<td>( \Phi_{1,1} )</td>
<td>5</td>
<td>4.48</td>
<td>#2</td>
</tr>
<tr>
<td>s1512</td>
<td>50000</td>
<td>( \Phi_{2,1} )</td>
<td>100</td>
<td>3.80</td>
<td>*80</td>
</tr>
<tr>
<td>s1423p1</td>
<td>50000</td>
<td>( \Phi_{1,0} )</td>
<td>-</td>
<td>13.55</td>
<td>#11</td>
</tr>
<tr>
<td>s1423p3</td>
<td>50000</td>
<td>( \Phi_{1,0} )</td>
<td>12</td>
<td>14.27</td>
<td>#11</td>
</tr>
<tr>
<td>nh2</td>
<td>50000</td>
<td>( \Phi_{1,1} )</td>
<td>1000</td>
<td>2.30</td>
<td>663</td>
</tr>
</tbody>
</table>

Fig. 7. Results for fully optimized sequential SymC. The first column lists the design name. Column two gives the splitting threshold. The third column shows the influence used for \( \text{MinOverlap} \) splitting. The fourth and fifth columns list the time bound specified in the property and maximum peak node count in millions, respectively. The last column shows the overall verification time. #n or *n denote memory overflow or time out at step n.

Fig. 8 shows the results of the distributed approach with dynamic overlap removal using 32 processors dedicated to the checking algorithm and one processor acting as the coordinator. In these experiments, dynamic overlap removal is applied throughout the verification process repeatedly every \( p \) steps.

Discussion: First of all, the parallel algorithm is able to finish all the problems that the sequential approach was not able to handle due to space or time restrictions.

Designs \( s4863 \) and \( s1512 \) clearly show the advantage of both parallelization and dynamic overlap removal, i.e. decreasing \( p \) reduces verification time. Also, traversal of design \( nh2 \) completes with a speedup of 2.8 compared to the sequential version.

4 Experiments were stopped after one hour.
For design $s1423$ we considered three properties $p1$, $p2$ and $p3$. Both $p1$ and $p2$ are from [2] and pure LTL properties, hence there is no time bound specified in the property. In comparison to [2], SymC finds errors in the designs significantly faster, even taking different hardware configurations into account. However, design $s1423$ behaves unexpectedly as verification time increases with shorter dynamic overlap reduction periods. This effect is caused by the behavior of the BDDs representing the state sets. Removing states from the sets actually increases their BDD representation. This opens a new thread for heuristics when and how to apply dynamic removal. We also investigate if dynamic variable reordering takes care of this problem.

Fig. 9 depicts the natural load balance graph for the circuit $s1512$ with reduction period 2. Only four nodes are shown for clear visibility of the graph. The load balancing effect can be seen very well when nodes 0 and 24 swap their arrival order during execution.

Finally, measurements indicate that reading and writing BDDs to and from disk does not contribute to the overall verification time significantly. Thus, network I/O is not a bottleneck of the distributed algorithm.
7 Conclusions and Future Work

This paper presents the parallelization of a BDD-based bounded property checking algorithm. The two main contributions are a novel splitting algorithm taking overlap reduction into account and a distributed on-the-fly algorithm for asynchronous state space traversal with dynamic overlap reduction resulting in natural load balancing.

The MinOverlap splitting heuristic enhances current decomposition algorithms by preprocessing the transition relation and using this information for ordering the list of potential splitting variables. The experiments show that this preprocessing step is able to actually reduce the overlap and the splitting time. Furthermore, MinOverlap almost never degrades the splitting runtime or the resulting overlap significantly.

Dynamic overlap reduction is an important technique in enabling verification of larger designs and significantly improves the applicability of the distributed algorithm. Reassigning idle nodes avoids wasted computation power. However, for some designs overlap reduction can actually increase the BDD representation of sets with fewer states. This seems to be related to the characteristic that a fixed BDD variable order is kept after the sequential stage. We experiment with different variable orderings on computation nodes to handle these cases. Furthermore, we are extending our experiments to designs from the VIS suite and recent IBM examples.

8 Acknowledgements

We want to thank the reviewers for their detailed comments that helped in enhancing the quality of this paper.

References


