Parallel Bounded Property Checking with SymC [⋆]

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

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Sand 13, 72076 Tübingen, Germany
{nalla,weissr,ruf,kropf,rosenstiel}@informatik.uni-tuebingen.de

Abstract. Today, verification of industrial size designs like multi-million gate ASICs (Application Specific Integrated Circuit) and SoC (System-on-a-Chip) processors consumes up to 75% of the design effort. The trend to augment functional verification with formal verification tries to alleviate this problem. Efficient property checking algorithms based on binary decision diagrams (BDDs) and satisfiability (SAT) solvers allow automatic verification of medium-sized designs. However, the steadily increasing design sizes still leave verification the major bottleneck, because formal methodologies do not yet scale to very large designs.
To address these problems, we developed the bounded property checking tool SymC. SymC takes properties and a system description as inputs and translates them into a symbolically simulatable representation. SymC performs forward state space traversal for verifying the properties. However, for larger designs SymC cannot complete the traversal due to the state space explosion problem.
Therefore, we propose a parallel version of SymC. The main idea of our approach is to split the state set into partitions and delegate traversal of these subsets to nodes on a cluster computer. Depending on the property and the quantification operator, detecting an accepting or rejecting state on one node can immediately abort computation on all other nodes and a witness/counterexample is produced. Otherwise, only the current search path is terminated and the remaining paths are traversed further. Parallel computation shows approximately linear speedups in execution time, enables faster verification of properties and we are able to handle larger designs.

1 Introduction

Formal verification is increasingly important in the design process of large circuits. The two most widely used formal verification techniques are symbolic model checking based on state space traversal using BDDs [1, 2] and bounded model checking (BMC) using satisfiability (SAT) solvers [3]. The idea of BMC is to unroll the sequential circuit into k time-frames, and counterexamples are searched in this unrolled system description. If no bug is found then one increases k until either a bug is found or some defined upper bound is reached. The BMC problem can be efficiently reduced to a propositional satisfiability problem. However, BMC works well for errors that are not too deep, i.e. not far from the initial states. For deep errors, the favorable option is state space traversal using BDDs. BDDs provide a canonical and compact symbolic representation of a Boolean function. In many cases BDDs can represent a large number of states very compactly and allow the verification and synthesis of systems having large state spaces [4]. But this technique faces the state space explosion problem as the design gets larger. In general, the size of the state space grows exponentially in the number of state elements present in the design. For some large designs BDD-based symbolic techniques do not allow complete analysis of the state space due to very deep errors. Further, BDDs are unstable, i.e. the performance of BDDs is sensitive to several parameters in constructing BDDs. The variable ordering plays a major role in deciding the size of BDDs. Small changes in parameters can yield significant variations in a BDD's memory requirements.

[⋆] 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.

To address the above problems we propose a method to parallelize the task of bounded property checking. In contrast to a classical model checker, our bounded property checker SymC performs one forward image computation at a time, i.e. the current set of states[1] is replaced by the set of states reachable within one transition. This results in an efficient verification for properties with time bounds by avoiding fixpoint iterations and reachable state set computations. However, for some bigger models computations using SymC require a lot of memory and also face the BDD explosion problem.

One proposed solution to the state explosion problem is partitioning the state space upon reaching a certain threshold and exploring each partition sequentially one at a time, while other partitions are kept in external memory [4, 5]. Our work is similar to this approach but all the partitions are explored simultaneously in a parallel framework.

Our approach parallelizes the symbolic state space traversal on a network of processors that communicate via the message passing paradigm. First, we start traversal in the set of initial states and iteratively compute the frontier set until S_{curr} reaches a given threshold size. Then S_{curr} is partitioned into subsets, where each subset is assigned to one node of the cluster computer. As soon as a node has its subset, it proceeds to compute forward state space traversal in iterative BFS (Breadth First Search) steps. In general, computation on a smaller subset requires less memory compared to the whole set. This method enables us to find errors that are far from the initial states. We can analyze bigger designs than with sequential SymC. Moreover, parallel computation takes less verification time compared to the partitioned sequential approach.

The remaining paper is organized as follows. Section 2 describes time bounded property checking in SymC and the motivation for parallelization of SymC. Section 3 describes parallel bounded property checking using SymC and also presents the two core algorithms executing on the master and slave nodes. Section 4 explains partitioning and the partition heuristics we considered in our work. Experimental results are presented in Section 5. Section 6 concludes and discusses future work.

2 SymC

The formal verification tool SymC [6, 7] combines bounded property checking and symbolic traversal. It takes a system description either as Verilog gate list or as a simple SMV-like [8] finite state description and temporal expressions in PSL (Property Specification Language) foundation language or FLTL (Finite Linear time Temporal Logic) [9]. The temporal logic formulas are converted to special finite state machines called AR-automata [10]. Later SymC translates both the system description and AR-automata into a BDD form. In order to avoid the construction of the complete transition relation we use a set of conjunctive partitioned transition relations. SymC traverses the design and the properties simultaneously and observes the state of the properties and reports success or failure to the user.

2.1 The Checking Algorithm of SymC

Our bounded property checking algorithm works in two steps. In the first step we compute the successor states of the AR-automata and we check whether a formula is accepted or

[1] Throughout the paper S_{curr} represents the current state set.

rejected. In the second step of each iteration we perform one symbolic execution step on the system under inspection. During image computation we build the conjunction of all partitions on-the-fly to obtain the successor state set. Fig. 1 shows the general operation of SymC.

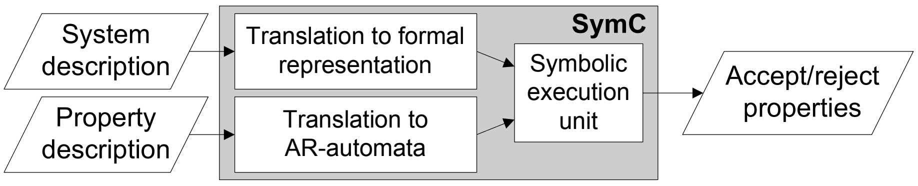

Fig. 1. Overview of SymC operation.

Similar to bounded model checking we do not traverse the state space exhaustively but from a given start set we examine all states reachable up to a given time bound, which is either given explicitly by the user or implicitly by the property.

2.2 Motivation for Parallelization of SymC

Sequential SymC partitions S_{curr} into smaller subsets when the BDD reaches a certain threshold size. Then, it explores these subsets sequentially. Once it reaches an accept/reject state, according to the existential/universal quantification of a property, SymC can save time and space by skipping exploration of the rest of the partitions. Assume, our design is error free or the reject state can be reached only in the final partition. Then we have to explore all the partitions, which is a time consuming process.

Let n be the number of partitions, t_i the time it takes for partition i to traverse fully, and $max(t_i)$ the maximum time taken by a partition for exploration of its state space. Then $\sum_{i=1}^{n} t_i$ is the time taken for full exploration of the system's state space.

Let a_j (r_j) be the time it takes to reach an accept (reject) state in partition j. For the rest of the text, we only talk about reject states without loss of generality. Suppose the partitions are scheduled such that we are reaching the reject state in partition j, then $r_j + \sum_{i=1}^{j-1} t_i$ is the time taken by the sequential algorithm to reach the reject state.

Let $R = \{P_1, \ldots, P_k\}$ be the set of partitions from which one can reach the reject state and $min(P_i)$ the minimum time taken by a partition to reach the reject state. When all partitions are executed in parallel, the time taken for exploration of the whole state space is $max(t_i)$ and the fastest reachability of the reject state is achieved in $min(P_i)$. In the sequential approach it may happen that partitions $P_i \notin R$ are explored unnecessarily as they can not reach the reject state. Even if we reach the reject state, it may not be the partition which takes the least time to reach the reject state that is explored first. The explanation of splitting sets into partitions is postponed to section 4.

In comparison to sequential SymC, the parallel approach has these two main advantages because it requires less memory on each node:

1. It allows faster verification of properties. This is not only true because of the reduced memory requirements, but also because the parallel algorithm is not sensitive to the scheduling of partition traversal.

2. The parallel tool can traverse further into large systems where the sequential algorithm fails due to memory exhaustion.

3 Parallelization of the Bounded Property Checking Algorithm

In this section we describe bounded property checking in parallel. Before getting into the details of our core algorithms we discuss how transmission of BDDs and communication between network nodes is handled.

For BDD transmission we use the available network drives and the DDDMP package of the CUDD library [11]. BDD serialization is performed in binary form. In binary mode, the BDD nodes are a sequence of bytes, representing the variable index, Then-index, and Else-index in an optimized way. While receiving, depending on this index information we can again frame the original BDD. For network communication we use TPO++ [12], an object oriented message-passing library written in C++ on top of MPI (Message Passing Interface) [13]. TPO++ allows easy transmission of objects and supports STL data types.

The nodes of a cluster computer are categorized into one master and otherwise slave nodes. First, the master node parses the system description and property specification and translates them into BDD form. These are basically the transition relation of the system and the properties' AR-automata. Once the master node has generated the BDDs, it dumps these BDDs onto disk and broadcasts the availability of the transition relation and AR-automata to the slave nodes, which are waiting for this event to occur. After successful message transmission, the master starts its symbolic state traversal algorithm, whereas the slaves will remain in waiting state after loading the BDDs from disk. The property checking algorithm for the master continues until the size of S_{curr} reaches the initial threshold limit. At this point, it stores S_{curr} on disk and indicates the slaves to load this set and split it. The main reason for doing this is to reduce the splitting time and distribute the splitting effort on all nodes in the network. In detail, the master node can split S_{curr} into n subsets. But this process consumes notable amount of time and splitting S_{curr} in parallel yields better results. Depending on the node rank[2], each node splits S_{curr} and obtains its subset. We restrict ourselves for better memory balancing to n nodes, where $n = 2^k$ and $k \in \{1, 2, 3, \dots\}$.

Fig. 2 illustrates the initial state set distribution algorithm. It iteratively splits a state set S into two parts and drops one of the resulting sets. In the end it keeps the subset that belongs to the node identified by its rank. The left hand side of Fig. 2 shows the distribution algorithm *getSubset*. It returns the subset S_{rank} for node *rank* from set S for n possible nodes. It calls algorithm *split* that partitions a set S into two disjunct subsets g and h. An example application of *getSubset*$(S, 8, 5, S_{rank})$ is shown on the right side of Fig. 2. The algorithm iterates $log_2(8) = 3$ times. First, it splits S into g and h and updates S_{rank} with g and skips h. Second, the algorithm splits S_{rank} and updates S_{rank} with h and skips g. Third, it splits S_{rank} and sets S_{rank} to g and skips h. The control flow of this example is indicated by the bold arrows in the figure.

After the assignment of the state subsets to each node, all nodes will proceed with forward state space traversal. Whenever one of the nodes detects the termination condition, it initiates abortion of the other nodes and optionally indicates the master node to

[2] In MPI each node is identified by a unique number called rank.

```
// get subset for rank with n nodes from set S
getSubset(in: S, n, rank; out: $S_{rank}$)
  $S_{rank}$ = S;
  for i = 1 … $\log_2(n)$
    split($S_{rank}$; g, h);
    if (rank % 2) $S_{rank}$ = g;  // skip h for odd rank
    else $S_{rank}$ = h;           // skip g for even rank
    rank = rank / 2;

// split state set S into two parts g and h
split(in: S; out: g, h)
```

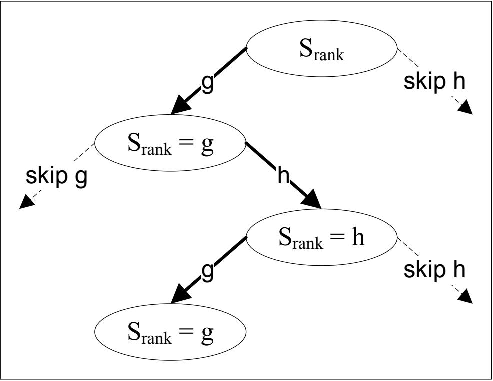

Fig. 2. Algorithm for state set distribution. The left hand side contains the distribution algorithms, an example application is shown on the right hand side.

compute the witness/counterexample. The left and right hand sides of Fig. 3 delineate the master and slave node computation loops, respectively. The termination condition in the inner loop is a generalization of the one given in [14] for distributed computing.

4 Splitting

When we work with parallel SymC and the size of S_{curr} is small, then many initial steps of state space traversal are identical to sequential SymC, i.e. S_{curr} can be represented using a single BDD and partitioning is delayed. Traversal is performed until a blow up in BDD size is detected. Blow up can be detected by measuring the size of the symbolic representation of S_{curr}. After a blow up is detected, we perform state set splitting based on the BDD variables representing S_{curr}. The goal of partitioning is to create small and relatively balanced partitions. They should be disjoint in order to avoid the duplication of work. Since state sets are represented as Boolean functions our partitioning is based on Boolean function slicing [4, 15, 5].

We experimented with several splitting algorithms, both with our own and already publicized heuristics [15, 16, 11]. Most of the partitioning algorithms are based on splitting a Boolean function represented as BDD into two parts depending on a variable, i.e. Shannon expansion is applied. If f is a function represented using a BDD and x is one of its variables, then f can be split into $f_x = f \wedge x$ and $f_{\bar{x}} = f \wedge \bar{x}$. The variable is chosen to balance the sizes of the two resulting functions and to keep them small. We will explain the heuristics for selecting a variable below.

4.1 Partitioning Algorithms from Literature

First we discuss the algorithms we considered from references. *Heavy branch subsetting* and *short path subsetting* are mentioned in [16]. The *heavy branch subsetting* algorithm computes a dense subset from a BDD. It determines how many states are in the function rooted at each internal node and builds a subset by throwing away one of the children of each node, starting from the root, until the result reaches a given threshold. The child that is eliminated from the result is the one that contributes fewer states. Whereas in *short path subsetting*, the main idea is that short paths in a BDD give many states and contribute few

```
// distribute the transition relation and AR-automata
sendTrans();
sendAR-Automata();

symbolicSimulate()
    N = nodesInCommWorld;
    $S_{curr}$ = Sys.start $\wedge$ AR-aut.start;
    splitFlag $=$ true;
    for i = 0 ... k // k is the time bound
        checkAbortCondition();
        if ( (|$S_{curr}$ | $\geq$ threshold) $\wedge$ (splitFlag) )
            distributeCurrentStateSet();
            $S_{curr}$ = getSubset($S_{curr}$, N, rank);
            splitFlag $=$ false;
        // Compute image of AR-Automata.
        $S_{curr} = \text{image}_{AR}(S_{curr})$;
        if (check Universally)
            if ( $S_{curr}$ $\wedge$ AR.reject $\neq$ false )
                reportFailure();
                abortSlaveNodes();
            if ( $S_{curr}$ $\wedge$ AR.accept $= S_{curr}$)
                reportAcceptance();
        if (check Existentially)
            if ( $S_{curr}$ $\wedge$ AR.accept $\neq$ false )
                reportAcceptance();
                abortSlaveNodes();
            if ( $S_{curr}$ $\wedge$ AR.reject $= S_{curr}$)
                reportRejectance();
        // Compute image of system.
        $S_{curr} = \text{image}_T(S_{curr})$;
```

```
// receive the transition relation and AR-automata
receiveTrans();
receiveAR-Automata();

symbolicSimulate()
    N = nodesInCommWorld;
    waitForCurrentStateSet();
    $S_{curr}$ = getSubset($S_{curr}$, N, rank);
    // n is the time point of the initial partitioning
    // and k is the time bound
    for i = n ... k
        checkAbortCondition();
        // Compute image of AR-Automata.
        $S_{curr} = \text{image}_{AR}(S_{curr})$;
        if (check Universally)
            if ( $S_{curr}$ $\wedge$ AR.reject $\neq$ false )
                reportFailure();
                abortOtherNodes();
            if ( $S_{curr}$ $\wedge$ AR.accept $= S_{curr}$)
                reportAcceptance();
        if (check Existentially)
            if ( $S_{curr}$ $\wedge$ AR.accept $\neq$ false )
                reportAcceptance();
                abortOtherNodes();
            if ( $S_{curr}$ $\wedge$ AR.reject $= S_{curr}$)
                reportRejectance();
        // Compute image of system.
        $S_{curr} = \text{image}_T(S_{curr})$;
```

Fig. 3. Master and slave node main computation loops.

nodes. The algorithm computes the short paths through each node and extracts the dense subset by removing the nodes with no short paths through them. These two algorithms work better for sequential SymC, i.e. whenever the size of S_{curr} reaches the threshold limit we apply one of these two algorithms and first traverse the dense subset, keeping the remaining state space on a stack. These algorithms are of minor interest in parallel SymC as they result in unbalanced cluster nodes in terms of memory and computation power consumption.

The other algorithms are *variable disjunctive decomposition* and *generative disjunctive decomposition* from [11]. These algorithms are similar to our algorithms but consume more time for decomposition. They try to estimate all the variables' positive and negative co-factors and take the best one. The final algorithm [15] takes reduction and redundancy factors into account for giving a better decomposition of a BDD but also consumes a notable amount of time.

4.2 New Splitting Heuristics

Compared to the above algorithms, our first algorithm tries to select a variable for which its positive and negative co-factors are well balanced according to a balancing condition. If it can not find such a variable, it picks the variable resulting in the least difference in the sizes of its positive and negative co-factors. Since we eagerly check for the variable that satisfies the balancing condition we call this algorithm *eager decomposition*. That is whenever we find an appropriate variable, we skip the exploration of the remaining variables.

```
selectVariable(in: $S$; out: bestVar)
    bestDiff = $|S|$; // initialize with number of BDD nodes of $S$
    for all $x_i$ // $x_i \in S.support()$
      balance = $\frac{|f_{x_i}|}{|f_{\bar{x}_i}|}$;
      // balancing condition
      if ( 0.75 $\leq$ balance $\leq$ 1.25 )
        bestVar = $x_i$;
        break;
      // otherwise check if variable is better than current one
      diff = $abs(|f_{x_i}| - |f_{\bar{x}_i}|)$;
      if ( diff $<$ bestDiff )
        bestDiff = diff;
        bestVar = $x_i$;
```

Fig. 4. Variable selection for *eager decomposition*.

After partitioning, each node is assigned with its state subset for symbolic state space traversal. A very important observation is that after a few steps of traversal, it may happen that there is a state overlap between network nodes.

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 minterms of the BDD.

Definition 2 Let S be a set and $P_1, \ldots, P_n \subseteq S$. Then we define the state overlap $o \in [0, 1]$ as:

$$o = \frac{\|\{p \in S : p \in P_i \cap P_j, 1 \leq i, j \leq n \wedge i \neq j\}\|}{\| \bigcup_{i=1}^{n} P_i \|} \tag{1}$$

Keeping this in mind, we developed the heuristic *minimal overlap* that aims at minimizing the state overlap. This should reduce the effort spent on the network nodes, as redundant computations are avoided.

We rely on the fact that the transition relation is conjunctively partitioned. All these partitions are statically analyzed to find dependencies between the state variables. First, we determine the number of present state variables that influence the truth value of a next state variable. Next, we order the state variables according to this dependency count. The variables with the highest dependency count are selected for splitting. The crucial point behind the *minimal overlap* heuristic is that if splitting is done on one of the selected variables, then image computations in these partitions are less likely to produce the same truth values in the dependent next state variables.

Often, the selected variable can not partition S_{curr} into balanced subsets. In order to overcome this problem, the best-cost algorithm [15] is called with the set of selected variables. It returns one variable for achieving a reasonably balanced partitioning.

Of course, in the worst case the minimal overlap condition holds only for one or few traversal steps, as many other conditions can change the value of state variables. For example, the variable with the maximal dependency count can depend on an input variable disjunctively. The selected set of variables is further scrutinized to avoid such trivial situations.

5 Experimental Results

We performed our experiments on the Kepler cluster at the University of Tuebingen. This cluster contains 128 computing nodes, each consisting of dual 650 MHz Pentium 3 processors with 1 GB of memory. The communication between nodes consists of a Myrinet 1.28 GBit/s switched LAN. The SCore Cluster System Software is used for communication between network nodes. We conducted our experiments on some of the circuits from the ISCAS89 benchmarks, and a model of the holonic production system nh2 described in [9]. We compared the results of parallel SymC and sequential SymC.

Checked properties For circuits from the ISCAS89 benchmarks we check for reachability of a certain state. In the holonic production system we check for consumption of a workpiece. The properties are specified with time bounds.

Partitioning Algorithms Since some of the partitioning algorithms have similar characteristics, for example *variable disjunctive decomposition* and *generative disjunctive decomposition*, we performed our experiments with a subset of the algorithms specified in section 4. Due to the reason given in section 4.1, none of the dense approximation techniques have been used for the parallel version of SymC. The heuristic from [15] is labeled *Haifa decomposition*.

Fig. 5 shows the results of running the mentioned circuits in partitioned sequential SymC and parallel SymC. The first column indicates the circuit names. The column *threshold limit* indicates that whenever the size of S_{curr} reaches this limit, the given splitting algorithm is called. The third column represents the time bound specified in a property. Column *time* specifies the time in seconds taken by an approach to reach the time bound or to get results. For parallel SymC, we also mention the number of nodes used in the cluster computer.

Discussion In Fig. 5 the runtimes marked with bold text denote the splitting algorithm yielding the best result. For circuit *nh2*, the parallel approach using 16 nodes can not complete traversal in one hour due to the huge state overlap between network nodes. However, with 32 nodes parallel SymC decreases the verification time by a factor from 3 to 5. For the larger ISCAS89 circuits *s1512* and *s4863* sequential SymC was not able to produce any results with most splitting algorithms, whereas parallel SymC traversal completes and we obtain results.

6 Conclusion and Future Work

In this paper we presented a parallel version of the bounded property checking tool SymC and successfully tested big designs with large time bounds. The idea of the approach is to partition the state space upon reaching a threshold limit and assign traversal of the

Circuit	Threshold Limit	Time Bound	Partitioning Algorithm	Time taken by nr. of nodes		Time
				16	32	Seq.
nh2	50000	300	VarDisjDecomp	mtaoh	204.06	741.95
			MinimalOverlap	mtaoh	***197.38***	539.01
			EagerDecomp	mtaoh	mtaoh	591.84
			HaifaDecomp	mtaoh	208.5	902.05
			HeavyBranchSubset			***452.21***
		1000	VarDisjDecomp	mtaoh	375.77	mtaoh
			MinimalOverlap	mtaoh	379.74	1273.29
			EagerDecomp	mtaoh	mtaoh	2489.54
			HaifaDecomp	mtaoh	***272.25***	2332.3
			HeavyBranchSubset			***1242.65***
		2000	VarDisjDecomp	mtaoh	450.5	mtaoh
			MinimalOverlap	mtaoh	566.15	***1738.11***
			EagerDecomp	mtaoh	mtaoh	3195.98
			HaifaDecomp	mtaoh	***357.29***	3074.1
			HeavyBranchSubset			1824.18
s1512	50000	100	VarDisjDecomp	2146.25	2103.45	mtaoh
			MinimalOverlap	***2135.46***	***2083.24***	mtaoh
			EagerDecomp	2772.98	2750.55	mtaoh
			HaifaDecomp	2450.75	2363.23	mtaoh
			HeavyBranchSubset			mtaoh
s1269	5000	10	VarDisjDecomp	***30.34***	***21.74***	mtaoh
			MinimalOverlap	41.79	22.88	***168.92***
			EagerDecomp	36.01	31.67	307.81
			HaifaDecomp	32.32	22.58	182.67
			HeavyBranchSubset			213.38
s1423	50000	10	VarDisjDecomp	***215.12***	***203.69***	519.35
			MinimalOverlap	243.79	233.02	***503.5***
			EagerDecomp	245.73	224.2	746.25
			HaifaDecomp	303.73	293.88	503.62
			HeavyBranchSubset			585.48
s4863	20000	5	VarDisjDecomp	952.22	813.69	mo
			MinimalOverlap	***714.49***	***579.86***	***1786.88***
			EagerDecomp	mo	mo	mo
			HaifaDecomp	967.95	851.39	mo
			HeavyBranchSubset			2802.77

Fig. 5. Comparison of parallel SymC with partitioned sequential SymC, where *mtaoh* denotes measurement terminated after one hour and *mo* denotes memory overflow.

subsets to network nodes. The parallel algorithm has several advantages. It enables the verification of larger models than those with the regular nonparallel version. Sequential SymC fails for these designs because it often encounters state space explosion early on in the computation, after which it could not make much progress due to memory limitations. However, the reduced memory requirements for the cluster nodes in parallel SymC still allow progress in the traversal process. Therefore, it is able to finish large circuits. The parallel approach can exploit any network size and its utilization of network resources make it suitable for solving very large verification problems. Also, because parallel SymC is not sensitive to the traversal scheduling order of partitions, the termination condition is found as soon as possible.

Our current focus is on state overlap reduction. In this work we reduced the state overlap between network nodes using the *minimal overlap* algorithm for state set splitting. Experiments show that this technique efficiently reduces the overlap for certain circuits. However, in general the overlap may still pursue after a few iterations. In order to dynamically avoid overlap, each node dumps its S_{curr} every n time steps onto the disk. Later nodes will remove the states already visited at time step n by other nodes. As of the time of writing this paper, we are ready with the basic implementation and tested a few designs. Early results look very promising.

Currently, we partition S_{curr} into smaller subsets and distributed them among network nodes. We would like to apply the same principle to the transition relation, i.e. we distribute restricted parts of the transition relation to the network nodes. In contrast to [15] we use network drives for BDD transmission. In the future we will evaluate using MPI directly for BDD transmission.

References

1. Bryant, R.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers **C-35** (1986) 677–691
2. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys **24(3)** (1992) 293–318
3. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. In Zelkowitz, M., ed.: Highly Dependable Software. Volume 58 of Advances in Computers. Academic Press (2003)
4. Narayan, A., Isles, A.J., Jain, J., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Reachability analysis using partitioned-ROBDDs. In: 1997 IEEE/ACM International Conference on CAD, ACM and IEEE Computer Society Press (1997) 388–393
5. Sahoo, D., Iyer, S.K., Jain, J., Stangier, C., Narayan, A., Dill, D.L., Emerson, E.A.: A partitioning methodology for BDD-based verification. In Hu, A.J., Martin, A.K., eds.: Formal Methods in Computer-Aided Design, Fifth International Conference. Volume 3312 of Lecture Notes in Computer Science., Springer (2004) 399–413
6. Ruf, J., Peranandam, P.: Bounded property checking with symbolic simulation. In: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, GI/ITG/GMM Workshop, Shaker Verlag (2003) 209–218
7. Peranandam, P.M., Weiss, R.J., Ruf, J., Kropf, T., Rosenstiel, W.: Dynamic guiding of bounded property checking. In: IEEE International High Level Design Validation and Test Workshop 2004 (HLDVT 04). (2004)
8. McMillan, K.: Symbolic Model Checking: An Approach to the State Explosion Problem. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1992) CMU-CS-92-131.
9. Ruf, J., Weiss, R.J., Kropf, T., Rosenstiel, W.: Modeling and formal verification of production automation systems. In et. al., E., ed.: Integration of Software Specification Techniques for Applications in Engineering. Volume 3147 of Lecture Notes in Computer Science. Springer (2004) 541–566
10. Ruf, J., Hoffmann, D.W., Kropf, T., Rosenstiel, W.: Simulation-guided property checking based on a multi-valued AR-automata. In Nebel, W., Jerraya, A., eds.: Design, Automation and Test in Europe 2001, IEEE Press (2001) 742–748
11. Somenzi, F.: CUDD: CU decision diagram package, release 2.4.0. `http://vlsi.colorado.edu/~fabio/CUDD` (2004)

12. Grundmann, T., Ritt, M., Rosenstiel, W.: TPO++: An object-oriented message-passing library in c++. In: 2000 International Conference on Parallel Processing, IEEE Computer Society (2000) 43–50
13. William Gropp, E.L., Skjellum, A.: Using MPI - Portable Parallel Programming with the Message Passing Interface. 2nd edn. MIT Press (1999)
14. Ruf, J., Peranandam, P.M., Kropf, T., Rosenstiel, W.: Bounded property checking with symbolic simulation. In: Forum on Specification and Design Languages 2003. (2003)
15. Heyman, T., Geist, D., Grumberg, O., Schuster, A.: Achieving scalability in parallel reachability analysis of very large circuits. In Emerson, E.A., Sistla, A.P., eds.: Computer Aided Verification, 12th International Conference. Volume 1855 of Lecture Notes in Computer Science., Springer Verlag (2000) 20–35
16. Ravi, K., McMillan, K.L., Shiple, T.R., Somenzi, F.: Approximation and decomposition of binary decision diagrams. In: 35th Conference on Design Automation, ACM Press (1998) 445–450