Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Scalable Distributed On-the-Fly Symbolic Model Checking*

Shoham Ben-David[2], Orna Grumberg[1], Tamir Heyman[1,2], Assaf Schuster[1]

[1] Computer Science Department, Technion, Haifa, Israel

[2] IBM Haifa Research Laboratories, Haifa, Israel

The date of receipt and acceptance will be inserted by the editor

Received: date / Revised version: date

Abstract. This paper presents a scalable method for parallel symbolic on-the-fly model checking in a distributed memory environment. Our method combines a scheme for on-the-fly model checking for safety properties with a scheme for scalable reachability analysis. We suggest an efficient, BDD-based algorithm for a distributed construction of a counterexample. The extra memory requirement for counterexample generation is evenly distributed among the processes by a memory balancing procedure. At no point during computation does the memory of a single process contain all the data. This enhances scalability. Collaboration between the parallel processes during counterexample generation reduces memory utilization for the backward step.

We implemented our method on a standard, loosely- connected environment of workstations, using a high-performance model checker. Our initial performance evaluation, carried out on several large circuits, shows that our method can check models that are too large to fit in the memory of a single node. Our on-the-fly approach may find counterexamples even when the model is too large to fit in the memory of the parallel system.

1 Introduction

A model checking algorithm takes a model and a specification written as a temporal formula. If the model satisfies the formula, the algorithm returns 'true'; otherwise it returns 'false' and provides a counterexample demonstrating why the model does not satisfy the formula. The counterexample feature is vital to the debugging of the system.

Model checking tools have successfully uncovered subtle errors in medium-sized complex designs. However, the large memory requirements of these tools limit their applicability to large designs. This is their main drawback. Many approaches to reducing the memory requirements of model checking tools have been investigated. One of the most successful approaches is *symbolic model checking* [8], in which computation is done over a set of states. Many model checkers represent these sets using binary decision diagrams (BDDs) [6].

Another approach is *on-the-fly model checking*, in which parts of the model are developed whenever the need arises. The check is usually guided by an automaton that *monitors* the behavior of the system in order to detect errors and stop the evaluation as soon as an error is found. Several on-the-fly algorithms [13,20,5] for CTL* use a depth-first search (DFS) traversal of the state space. Since BDD-based methods work efficiently on sets of states, we use an on-the-fly algorithm suggested by Beer et al. [4]. This algorithm uses breadth-first search (BFS) for traversal of the state space. It model checks specifications given as regular expressions describing "bad" (unwanted) behaviors. Note the difference from regular model checking in which the specification formula describes the good behaviors. In this method, a regular expression is translated into an automaton, using the standard algorithm [15]. The acceptance state of the automaton indicates an error state in the model for the given specification. The automaton and the model are then multiplied. Finally, a BFS is used for reachability analysis. The BFS stops as soon as an error state is detected. Industrial temporal languages such as Sugar [2] and ForSpec [1] employ regular expressions. See appendix A for a detailed description of model checking regular expressions on-the-fly.

Other approaches [9,19,18,21,14] aim to reduce the memory requirements of model checking algorithms by partitioning the work into several tasks. This can be done by parallelizing an explicit-state model checker that does not use symbolic methods [21]; by using a single computer that handles one task at a time while keeping others in an external memory [9,19,18]; or by means of a distributed, symbolic

* This work is supported by a grant from the Israeli Science Foundation and by a grant from Intel Academic Relations.

algorithm for reachability analysis that works on a network of processes with distributed memory [14]. The algorithm in [14] achieved an average memory scale-up of 55 on 130 processes. This made it possible to handle designs that could not fit into the memory of a single machine.

In this work we combine the approaches of [4] and [14], obtaining a distributed symbolic on-the-fly model checking method that can handle very large designs. Our method includes a distributed algorithm that employs several processes for counterexample generation: the entire set of states is never held in a single process.

Producing the counterexample requires additional storage of sets of states during reachability analysis, one set for each step. In the distributed algorithm each process stores only part of each set. In order to balance the parts of the sets across the processes, we apply a slicing function that defines for each process the parts of the set it should store. The parts a process stores may belong to different parts of the state space. This makes the distributed counterexample generation somewhat tricky: we need to track the steps backwards while switching different slices and maintaining the memory requirement at a low level.

We implemented our method inside the high-performance verification tool RuleBase [3], developed by the IBM Haifa Research Lab. We used a distributed, non-dedicated, slow network system of 32 standard workstations. The performance results show that our method scales well. Large examples that could not fit into the memory of a single machine terminate using the parallel system. The parallel system appears to be balanced with respect to memory utilization. Furthermore, communication over the network does not become a bottleneck.

We were also able to show that the distributed algorithm is more effective for on-the-fly model checking that includes counterexample generation than it is for reachability analysis. There are two main reasons for this. First, the counterexample generation procedure requires that sets of states be saved, and this consumes more space. The parallel system, however, enables the effective splitting and balancing of this additional space. This enhances scalability. Second, the parallel system, even when failing to complete reachability to the fixpoint, is usually able to proceed for several steps beyond the point reached by a single machine. This improves the chances that our on-the-fly model checking will find an error state during these steps.

The rest of the paper is organized as follows. Section 2 describes the sequential on-the-fly algorithm for checking regular expressions. Section 3 presents our distributed on-the-fly model checking scheme. Section 4 provides our performance evaluation and Section 5 presents our conclusions.

2 The Sequential On-the-Fly Algorithm

In this section we describe the main characteristics of the sequential on-the-fly model checking algorithm presented in [4]. This algorithm is the basis for our distributed method.

Given a system model M and a regular expression φ describing "bad" behavior, the corresponding automaton $\mathcal{A}$ is constructed and combined with M. $\mathcal{A}$ monitors the behavior of M. If it detects an erroneous behavior, an error flag is set. $\mathcal{A}$ then enters a special state and stays there forever. We call a state that satisfies the error flag an *error state*. Thus, M does not contains any bad behaviors that satisfies φ if and only if the combination of M and $\mathcal{A}$ (that is, $M \times \mathcal{A}$) does not reach an error state. In order to check that M satisfies φ, we run a reachability analysis on $M \times \mathcal{A}$ that constantly checks whether an error state has been encountered. The algorithm traverses the (combined) model using a breadth–first search (BFS). Starting from the set of initial states, it constructs a *doughnut* at each iteration. This doughnut is the set of new states found in that iteration. The doughnuts are kept for later use in the generation of the counterexample. Keeping the doughnuts increases the space requirements of this algorithm, and they exceed those of (pure) reachability analysis.

The model checking algorithm terminates successfully if all reachable states have been traversed and no error state has been found. If at any stage an error state is encountered, the model checking algorithm stops and the generation of a counterexample begins.

A counterexample is a sequence of states that starts with an initial state and ends with an error state. It is generated backwards. The algorithm begins with an error state and selects a state from among its predecessors. Then the generation continues, following the doughnuts that were produced and stored by the reachability analysis algorithm. All these selected states are saved in the order in which they were found. Counterexample generation terminates when the doughnut of the initial states is reached. At this point the selected states comprise a complete counterexample sequence.

Figure 1 presents the sequential algorithm for on-the-fly model checking, including the counterexample generation procedure. The algorithm differs from simple BFS in three ways: it evaluates the formula while computing the set of reachable states; it saves the sets of states for the counterexample generation; if it reaches an error state, it constructs a counterexample. The counterexample generation procedure is based on the one in [11]. Lines 1–9 describe the model checking phase. At each iteration i, the set of `new` states that have not yet been reached is kept in doughnut S_i.

The algorithm terminates if either no new states are found ($\texttt{new} = \emptyset$), in which case it announces success, or if an `error` state is found ($\texttt{new} \cap \texttt{error} \neq \emptyset$), in which case it announces failure.

In lines 16–22, the counterexample $\texttt{Ce}_0, \ldots \texttt{Ce}_k$ is generated. The counterexample is of length $k+1$ (line 14), since an error state was first found in the k-th iteration. We choose $\texttt{Ce}_k \in S_k$ from among the error states reached. Having already chosen a state $\texttt{Ce}_i \in S_i$, we compute the set of `bad` states by finding the set of predecessors for $\texttt{Ce}_i$: $\texttt{pred}(\texttt{Ce}_i)$. We then intersect it with the doughnut S_{i-1} (line 19). Since each state in S_i is a successor of some state in S_{i-1}, the set `bad` will not be empty. We now choose $\texttt{Ce}_{i-1}$ from the set of

```
1 reachable = new = initialStates
2 i = 0
3 while ((new ≠ ∅)&&(new ∩ error = ∅)) {
4   S_i = new
5   i = i+1
6   next = nextStateImage(new)
7   new = next \ reachable
8   reachable = reachable ∪ next
9 }
10 if (new = ∅) {
11  print ``formula is true in the model''
12  return
13 }
14 k = i
15 print ``formula is false in the model''
16 bad = new ∩ error
17 while (i>=0) {
18  Ce_i = choose one state from bad
19  if (i>0) bad=pred(Ce_i)∩S_{i-1}
20  i = i-1
21 }
22 print ``counterexample is:'' Ce_0 ··· Ce_k
```

Fig. 1. Sequential algorithm for on-the-fly model checking, including counterexample generation

bad states. The generation of the counterexample continues until $\mathtt{Ce}_0$ is chosen.

3 Distributed Algorithm

The distributed algorithm for on-the-fly model checking consists of two phases:

- The model checking phase
- The counterexample generation phase

3.1 Distributed Model Checking

In the distributed algorithm, an initial sequential stage precedes the distributed stage. The reachable states are first computed on a single process. When a certain memory requirement threshold is reached, the state space is partitioned into k slices, whose union is the whole state space. This partition, or slicing, should require less memory. Furthermore, the subsets should be disjoint. Disjoint subsets will allow us to avoid duplication of work during reachability analysis. The slicing algorithm [14,18,10] selects a variable and uses it to slice a set into two disjoint subsets. Using the slicing algorithm k times results in k subsets that are distributed to k processes. This ends the sequential stage.

The distributed stage begins with each process being informed of the slice it *owns*, and of the slices owned by each of the other processes (which are *non-owned* by this process). The process receives its own slice and proceeds to compute the reachable states for that slice in iterative BFS steps. At each such step, the set of new states is kept in a doughnut.

Each process computes the set next of states that are reached directly from the states in its new set. The next set contains owned as well as non-owned states. Each process splits its next set according to the k slices and sends the non-owned states to their corresponding owners. At the same time, the process receives the set of states it owns from the other processes.

The model checking phase for one process P_j is given in lines 1–13 of Figure 2. Lines 1–3 describe the setup stage where the process receives the slice it owns and the initial sets of states it needs to compute from. Lines 5–17 describe the iterative computation.

Distributed termination detection (line 5) is used to determine when this phase should end. *All* processes should end at this phase if one of two conditions holds: none of the processes found a new state or one of them found an error state. In the first case, the specification has been proven correct and the algorithm terminates. In the second case the specification is false, and all processes proceed to the counterexample generation phase. In order to distinguish between the two cases, the termination detection procedure is used (line 14) with the error parameter equal 0.

Several points distinguish distributed model checking from sequential model checking. When distributed model checking is used,

- the set next is modified (lines 9–10) through communication with the other processes and is restricted to include only owned states;
- distributed termination detection is applied;
- for each doughnut i, each process P_j stores the slice of the doughnut $S_{(i,j)}$ it owns.

Our distributed algorithm is made particularly effective by the *memory balancing* procedure, which maintains approximately equal memory requirements across the processes during the entire computation. This is accomplished by pairing large slices with small ones and reslicing their union in a balanced way. As a result, a process owns (and stores) different slices of the doughnuts in different iterations. Therefore, in some iteration, a process may own a state that does not have any predecessors stored in the slices of the doughnuts it owned previously. The distributed generation of a (correct) counterexample is nonetheless guaranteed by the following property, which is true by construction:

$$S_i = \bigcup_j S_{(i,j)}, \tag{1}$$

where S_i is the doughnut computed by the sequential algorithm at iteration i.

3.2 Distributed Counterexample Generation

To generate a counterexample, our algorithm uses the doughnut slices that are stored in the memory of the processes. The distributed counterexample generation algorithm consists of *local phases* and *coordination phases*. In the local phase, all

processes run in parallel. Each process takes the counterexample generated so far, denoted by the suffix $\texttt{Ce}_i \ldots \texttt{Ce}_k$. It then executes the sequential algorithm for counterexample generation, adding the additional states $\texttt{Ce}_{i-1}$,$\texttt{Ce}_{i-2}$,... until it can proceed no further. A process may get stuck after producing a counterexample with suffix $\texttt{Ce}_i \ldots \texttt{Ce}_k$ if it cannot find a predecessor for $\texttt{Ce}_i$ in its own slice of the (i-1)th doughnut. However, by property (1) and by the fact that each element in S_i has a predecessor in S_{i-1}, there must be a process that has such a predecessor for $\texttt{Ce}_i$.

In the coordination phase, the process that produces the largest suffix is selected and used to reinitiate the local phase in all processes. If this suffix is complete (i.e., it contains all states $\texttt{Ce}_0 \ldots \texttt{Ce}_k$), the process simply prints its counterexample and all processes terminate. Otherwise, the process broadcasts its suffix, together with its iteration number, to all other processes. Each process updates its data accordingly and reinitiates the local phase from that point. The algorithm continues until a complete suffix is found.

Lines 18–35 of Figure 2 describe the algorithm. Lines 22–26 contain the local phase, while lines 27–35 contain the coordination phase. The algorithm uses the following three variables:

- `myId`, which is the index of the process (`myId`=j for process P_j);
- `minIte`, the smallest iteration number, chosen at the start of the coordination phase;
- `minProc`, the smallest index among the processes with the smallest iteration number.

3.3 Reducing Peak Memory Requirement

In order to generate the counterexample, the sets `bad` = $\texttt{pred}(\texttt{Ce}_i) \cap S_{(i,j)}$ must be computed. This is done by intersecting the doughnut slice $S_{(i,j)}$ with the set of predecessors of the state $\texttt{Ce}_i$ (lines 24, 35). The BDDs for $\texttt{Ce}_i$ and `bad` are usually small. However, a very large peak in memory use may be caused by intermediate BDDs obtained during the computation of `bad`. This phenomenon can be viewed in example GXI (Figure 9), where a significant increase in memory use causes the parallel system to overflow during the computation of the counterexample.

Changing the order of operations can, however, produce smaller intermediate BDDs. This, in turn, reduces the peak memory requirement. In the new order, we first restrict the transition relation of our model to the doughnut slice $S_{(i,j)}$ and only then use it to compute $\texttt{pred}(\texttt{Ce}_i)$. Since our implementation is based on the partitioned transition relation [7], we actually restrict each one of the partitions to the doughnut slice.

To increase precision, we define the operations we perform by means of Boolean functions (represented as BDDs). Assume that our model consists of a set of Boolean variables V. The Boolean function $TR(V, V')$ represents the transition relation of the model, where V and V' represent the current and next state, respectively.

```
 1 mySlice = receive(fromSingle)
 2 reachable = receive(fromSingle)
 3 new = receive(fromSingle)
 4 i = 0
 5 while (Termination(new,error)==0) {
 6   S(i,j) = new
 7   i = i+1
 8   next = nextStateImage(new)
 9   next = sendReceiveAll(next)
10   next = next ∩ mySlice
11   new = next \ reachable
12   reachable = reachable ∪ next
13 }
14 if (Termination(new,0)==1) {
15   print ``formula is true in the model''
16   return
17 }
18 k = i
19 print ``formula is false in the model''
20 bad = new ∩ error
21 while (i>=0) {
22   while ((i>=0) &&(bad ≠ ∅)) {
23     Cei = choose one state from bad
24     if (i>0) bad=pred(Cei)∩ S(i-1,j)
25     i = i-1
26   }
27   (minIte,minProc)=MinIteFromAll(i,myId)
28   i = minIte
29   if (i<0) {
30     if (myId == minProc)
31       print ``counterexample is:'' Ce0···Cek
32     return
33   }
34   Cei+1···Cek=broadcast(minProc,Cei+1···Cek)
35   bad=pred(Cei+1)∩ S(i,j)
  }
```

Fig. 2. Process P_j in the distributed algorithm for on-the-fly model checking, including the generation of a counterexample.

Let $\texttt{Ce}_i(V)$ be the Boolean function for the singleton set consisting of the state $\texttt{Ce}_i$, and let $S_{(i,j)}(V)$ be the Boolean function for the slice $S_{(i,j)}$. Then the computation of `bad` at the j'th process can be described by the expression

$$\exists V' \, [\, TR(V, V') \wedge Ce_i(V') \,] \; \wedge \; S_{(i,j)}(V). \qquad (2)$$

Our transition relation consists of partitions $PTR_n(V, V')$ such that $TR(V, V') = \bigwedge_n PTR_n(V, V')$. Consequently, the previous expression can be rewritten as

$$\exists V' \, [\, \bigwedge_n PTR_n(V, V') \wedge Ce_i(V') \,] \; \wedge \; S_{(i,j)}(V). \qquad (3)$$

Since $S_{(i,j)}(V)$ does not depend on V', it can be moved into the scope of the quantifier, resulting in an equivalent expression:

$$\exists V' [\bigwedge_n \big(PTR_n(V, V') \wedge S_{(i,j)}(V)\big) \wedge Ce_i(V')]. \qquad (4)$$

This expression describes the computation at the j'th process . First, each partition of the transition relation is restricted to

the doughnut slice $S_{(i,j)}$, and then the predecessors of Ce_i are computed.

This computation can be made more efficient by using the *simplify-assuming* technique [12]. Let $f : E \longrightarrow \{0,1\}$ be a Boolean function over some domain E. If we are concerned only with the value of f over some subset D of E, then we may reduce the BDD size for f. This can be done by finding another function f' which agrees with f on D and can have any value on elements not in D.

Formally, given a function $f : E \longrightarrow \{0,1\}$ and an assumption $D \subseteq E$, we say that a function $f' : E \longrightarrow \{0,1\}$ *simplifies* f *assuming* D if it satisfies

$$f' \wedge D = f \wedge D. \tag{5}$$

We denote such an f' by $f|_D$.

The algorithm given by [12] guarantees that the BDD size of f' is equal to or smaller than the BDD size of f. We use this technique to reduce the size of each partition in the transition relation. The reduced partition sizes decrease the memory requirement during the computation of the expression in (4). Instead of intersecting each $PTR_n(V,V')$ with $S_{(i,j)}(V)$, we simplify $PTR_n(V,V')$ assuming $S_{(i,j)}(V)$ and intersect the result with $S_{(i,j)}(V)$. Since simplify-assuming satisfies (5), the expression in (4) is equivalent to

$$\exists V'[\bigwedge_n \Big(PTR_n(V,V')|_{S_{(i,j)}(V)} \wedge S_{(i,j)}(V)\Big) \wedge Ce_i(V')]. \tag{6}$$

Since $S_{(i,j)}(V)$ does not depend on V', it can be moved outside of the scope of the quantifier, resulting in an equivalent expression:

$$\exists V'[\bigwedge_n \Big(PTR_n(V,V')|_{S_{(i,j)}(V)}\Big) \wedge Ce_i(V')] \wedge S_{(i,j)}(V). \tag{7}$$

The improvement described above uses precise information in order to restrict the partitions of the transition relation. This requires computing a different restriction for each doughnut in each step of the counterexample generation. We suggest a different method of restriction, which is computed only once for each process. Process P_j simplifies $PTR_n(V,V')$ assuming U_j, where $U_j = \cup_i S_{(i,j)}$ is the union of all the doughnut slices owned by P_j. Since $S_{(i,j)} \subseteq U_j$, the expression in (4) is equivalent to

$$\exists V'[\bigwedge_n \big(PTR_n(V,V')|_{U_j(V)}\big) \wedge Ce_i(V')] \wedge S_{(i,j)}(V). \tag{8}$$

Note that $PTR_n(V,V')|_{U_j(V)}$ is computed only once at the beginning of the counterexample generation process.

We next suggest an orthogonal improvement that exploits the fact that we compute the set of predecessors of a singleton ($Ce_i(V')$, which contains only one state, Ce_i). We replace the intersection of $PTR_n(V,V')$ and $Ce_i(V')$ by substituting the state Ce_i for V' in $PTR_n(V,V')$. The existential quantifier is then redundant and can be removed. Modified thus, equation (3) can first be rewritten as

$$\exists V' \, [\bigwedge_n \, (\, PTR_n(V, Ce_i) \,) \,] \, \wedge S_{(i,j)}(V). \tag{9}$$

The existential quantifier is then redundant and can be removed to obtain

$$\bigwedge_n \, (\, PTR_n(V, Ce_i) \,) \, \wedge S_{(i,j)}(V). \tag{10}$$

Combining the above optimization with simplify-assuming, we can compute (3) as

$$S_{(i,j)}(V) \wedge \bigwedge_n \, \big(\, PTR_n(V, Ce_i)|_{U_j(V)} \, \big) \, . \tag{11}$$

Again, $PTR_n(V,V')|_{U_j(V)}$ is computed only once. At step i of the counterexample generation procedure, P_j assigns Ce_i to each of $PTR_n(V,V')|_{U_j(V)}$ and then intersects $S_{(i,j)}(V)$ with them.

Experimental results show that all of the suggested optimizations significantly reduce memory requirements. Compare, for instance, the results in Figure 9, where example GXI is run without any optimization, to the results in Figure 10 where it runs with the optimization in expression 10. On the other hand, we found the optimization in expression 11 to have no significant advantage over the one in expression 10.

4 Experimental Results

In this section we report on the performance evaluation of our approach. We implemented our method inside the high-performance verification tool RuleBase [3], which is based on McMillan's SMV [17] and was developed by the IBM Haifa Research lab. Our parallel test-bed includes 32 RS6000 machines, each consisting of a 225 MHz PowerPC processor and 512 MB memory. The communication between the nodes consists of a 100 Megabit/second token ring.

We selected large circuits to show that the parallel system can find errors that the sequential algorithm cannot find. This is because the sequential algorithm uses more memory than is available in a single machine. We experimented with two of the largest circuits we found in the benchmarks: ISCAS89 +addendum'93. In order to test the counterexample generation, we used common properties that are often tested when verifying hardware designs. We also used two large examples, BIQ and GXI, which are components of IBM's Gigahertz processor. We used the original properties for these examples. These properties are explained in Figure 3. We mapped properties to automata using the IBM implementation as described in [4]. Characteristics of the circuits and the automata are given in Figure 4.

4.1 Space Reduction Using the Distributed Algorithm

This section presents the results for on-the-fly model checking of the benchmark suite using our 32 machine test-bed. Figures 5 to 11 summarize memory utilization, giving the peak memory consumption for every step. Each of the graphs compares the memory utilization in the single-machine execution to that of the parallel system. For the parallel system

we give the highest (peak) memory utilization in any of the machines.

We give examples for four models and six properties. Two properties are checked for the BIQ and S1423 models: one that overflows on a single machine, and another that completes the computation even when only a single machine is used.

As Figure 5 shows, an overflow occurs at cycle 15 while the algorithm searches for an error state in BIQ using a single machine. The overflow occurs because the counterexample generation (CE) phase requires that the doughnuts be saved, and this consumes a lot of memory. In contrast, the parallel algorithm does not overflow. It finds the error state in BIQ at cycle 17.

At the cycle where the error state is found, we see a drop in memory utilization. This drop is due to the fact that the CE phase will begin in the next step, making state exchange and load balancing unnecessary. State exchange and load balancing may contribute significantly to the observed peak in memory requirement. This effect is particularly strong in BIQ spec1, s1423 spec1, and s5378 spec1 (Figure 7, Figure 5 and Figure 11).

Figure 6 shows another drop in memory utilization, at the first counterexample cycle. This drop, which is characteristic of many examples, is caused by two factors. First, the transition relation computations during a backward step are usually simpler than those performed during a forward step, and they require less memory. This is due to the relative simplicity of the relation consisting of a single origin (the last state in the CE found so far). Second, the set of reachable states can be released since it is not needed for counterexample generation.

The parallel algorithm finds an error state in cycle 14 of S1423, as depicted in Figure 7. In this case, finding the error state on-the-fly is essential because, even with our parallel system, we were not able to complete reachability analysis on this example.

Figure 9 demonstrates why our optimizations are necessary. In this example, an overflow occurs during a step backwards from a single state using the original transition relation. The overflow occurs because we are using the partitioned transition relation, in which backward steps are much harder to perform than forward. We can avoid this problem by using substitution of the singleton (as describe in Section 3.3) instead of quantification over the partitioned transition relation. Figure 10 demonstrates the effect of this method on our example.

4.2 Timing and Communication in the Distributed Algorithm

Figure 12 gives the timing breakdown for on-the-fly model checking on our benchmark suite. The parallel reachability stage takes most of the computation time. As shown in [14], communication does not become a bottleneck at this stage.

5 Conclusions

Large clusters of computers are readily available nowadays. We believe that these environments should be regarded as huge memory pools that can be harnessed for problem solving. Our methods can be seen as a globalization of memory systems, using the network as an intermediate level in the memory hierarchy. This intermediate level resides between the main memory and the disk. Although the network is a lot faster than the disk (the difference in latency and bandwidth is from three to five orders of magnitude), it is also much slower than the main memory module (about three orders of magnitude for very fast networks). Locality, then, is still an important issue.

Our method was integrated into a high-performance model checker, thus proving its industrial potential. Its successful integration also shows that our parallelization method is orthogonal to other important optimizations and does not harm their applicability. The orthogonality may lead to further improvement of our results by applying other optimizations on the slices.

To the best of our knowledge, this is the first study reporting on parallel on-the-fly symbolic model checking. Our positive results clearly indicate that this is a promising direction, and one that deserves more attention.

6 Acknowledgments

We would like to thank the Formal Method group at IBM Haifa, and specifically Sharon Keidar and Yoav Rodeh, for helping us to embed our algorithm in the IBM model checker RuleBase.

References

1. R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. Springer-Verlag, 2002.
2. I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The Temporal Logic Sugar. In *Proc. of the 13th International Conference on Computer Aided Verification*. Springer-Verlag, June 2001.
3. I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rulebase: An Industry-Oriented Formal Verification Tool. In *33rd Design Automation Conference*, pages 655–660, 1996.
4. I. Beer, S. Ben-David, and A. Landver. On-the-Fly Model Checking of RCTL Formulas. In *Proc. of the 10th International Conference on Computer Aided Verification, LNCS 818*, pages 184–194. Springer-Verlag, June-July 1998.
5. G. Bhat, R. Cleaveland, and O. Grumberg. Efficient On-the-Fly Model Checking for CTL*. In *Proc. of the Conference on Logic in Computer Science (LICS'95)*, June 1995.

6. R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
7. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the 1991 International Conference on Very Large Scale Integration*, August 1991.
8. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.
9. G. Cabodi, P. Camurati, and S. Quer. Improved Reachability Analysis of Large FSM. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 354–360. IEEE Computer Society Press, June 1996.
10. G. Cabodi, P. Camurati, and S. Quer. Improving the Efficiency of BDD-Based Operators by Means of Partitioning. *IEEE Transactions on Computer-Aided Design*, pages 545–556, May 1999.
11. E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In *32rd Design Automation Conference*, pages 655–660, 1995.
12. O. Coudert, J. C. Madre, and C. Berthet. Verifying Temporal Properties of Sequential Machines without Building Their State Diagrams. In R. Kurshan and E. M. Clarke, editors, *Workshop on Computer Aided Verification, DIMACS, LNCS 531*, pages 23–32. Springer-Verlag, New Brunswick, NJ, June 1990.
13. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1:275–288, 1992.
14. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits. In *Proc. of the 12th International Conference on Computer Aided Verification*. Springer-Verlag, June 2000.
15. J.E. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesely, 1979.
16. D. E. Long. *Model Checking, Abstraction, and Compositional Reasoning*. PhD thesis, Carnegie Mellon University, 1993.
17. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
18. A. Narayan, A. Isles, J. Jain, R. Brayton, and A. L. Sangiovanni-Vincentelli. Reachability Analysis Using Partitioned-ROBDDs. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 388–393. IEEE Computer Society Press, June 1997.
19. A. Narayan, J. Jain, M. Fujita, and A. L. Sangiovanni-Vincentelli. Partitioned-ROBDDs. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 547–554. IEEE Computer Society Press, June 1996.
20. Doron Peled. Combining Partial Order Reductions with On-the-Fly Model Checking. In *Proc. of the Sixth International Conference on Computer Aided Verification, LNCS 818*, pages 377–390. Springer-Verlag, June 1994.
21. U. Stern and D. L. Dill. Parallelizing the Murphy Verifier. In *Proc. of the 9th International Conference on Computer Aided Verification, LNCS 1254*, pages 256–267. Springer-Verlag, June 1997.

A Regular Expressions in Symbolic Model Checking

When specifying a formula in temporal logic, one describes what should *hold* in the model. Another way to specify a property is to describe what should *never hold* in the model, that is, to describe the set of *bad* computations rather than the good ones. A nice way to describe a set of finite bad computations is by means of *regular expressions* (RE), as follows: Let W be a finite set of symbols (in our case, signal names in the model under test). The alphabet Σ, over which the regular expressions are defined, is the set of all Boolean expressions over W.

As an example, consider a model with two signals: req and ack, and consider a property specifying that every req must be followed by an ack in the next cycle. Σ in this case consists of all 16 possible Boolean functions ($true$, $false$, req, $\neg req$, $req \vee ack$, etc.). A description of the bad computations of this property would state that sequences with req holding in one state and ack *not* holding in the next state are illegal.

Using regular expressions, we get the following:

$$(true*)(req)(\neg ack)$$

In order to check a given model M against a RE specification r, one has to build the corresponding automaton A_r [15] and check that any word in $L(A_r)$ is not a prefix of a computation path in M.

Using RuleBase, we perform this check in the following way: First, we translate A_r into a corresponding nondeterministic finite state machine F_r in the input language of SMV, with final states $q_1, ..., q_n$. We then model-check the CTL formula

$$AG(\neg q_1 \wedge, ..., \wedge \neg q_n)$$

against the model $M \times F_r$.

Note that the formula to be checked is of the form $AG(p)$, where p is a Boolean formula. It can thus be checked on-the-fly [16,4], saving a lot of time and space. Model checking of regular expressions is more efficient in most cases than model checking of CTL formulas.

The expressive power of the regular expressions we have described above differs from that of temporal logics. In [4], Beer et al. present an algorithm for translating a subset of CTL formulas to RE specifications. The subset of CTL which can be translated to RE is called $RCTL$.

BIQ spec 1
If the $writePtr$ points to P and the value on the bus is D, then four cycles after the next time a $read$ from P occurs, the value going out should be D. Sugar: $\{[*], (writePtr(0..3) = P(0..3))\&(dataIn(0) = D(0)), goto(readPtr(0..3) = P(0..3))\}(AX[4](dataOut(0) = D(0)))$
BIQ spec 2
If the $writePtr$ points to P and the value on the bus is D, then two cycles after the next time a $read$ from P occurs, the value going out should be D. Sugar: $\{[*], (writePtr(0..3) = P(0..3))\&(dataIn(0) = D(0)), goto(readPtr(0..3) = P(0..3))\}(AX[2](dataOut(0) = D(0)))$
s1423 spec 1
If $G729$ and $G726$ are true, then $G726$ is true ten cycles later. Sugar: $AG(G729\&G726 \rightarrow AX[10](G726))$
s1423 spec 2
If $G729$ and $G726$ are true, then $G726$ is true seven cycles later. Sugar: $AG(G729\&G726 \rightarrow AX[7](G726))$
GXI spec 1
If $start$ is true and the address is A, then if two cycles later a rejection occurs, then between 2 to 32 cycle later, start should hold again, with address equals A. Sugar: $\{[*], START\&ADDR(0..2) = A, true, reject\}$ $(ABF[2..32](START\&ADDR(0..2) = A))$
s5378 spec 1
If $n3104gat$ is true, then starting six cycle later, $n3106gat$ should hold before $n3104gat$ holds. Sugar: $AG(n3104gat \rightarrow AX[6](n3106gat$ before $n3104gat))$

Fig. 3. The specifications in Sugar with explanations.

Circuit	#vars + sat	peak		spec check		
		size	step	time	steps	CE
BIQ						
spec 1	102 + 5	5.85M	15	15,059	Ov(15)	
spec 2	102 + 5	5.33M	14	3,811	15	95
s1423						
spec 1	91 + 4	8.64M	12	2,024	Ov(12)	
spec 2	91 + 3	1.54M	10	625	11	58
GXI						
spec 1	292 + 6	8.14M	44	16,222	Ov(44)	
s5378						
spec 1	188 + 4	9.66M	6	4,440	Ov(6)	

Fig. 4. #vars gives the number of variables in the model and in the sat(ellite). Sizes are given in million BDD nodes, and all times in seconds. The peak is the maximal memory requirement at any point during a step. In order to mask the effect of garbage collection scheduling decisions, the peak is measured after gc invocations. Spec check is the number of steps it takes to find an error state, and the time it takes to generate a counterexample (CE). Ov(x) designates memory overflow during step x. All measurements were taken using an RS6000 machine consisting of a 225 MHz PowerPC processor with 512 MB memory.

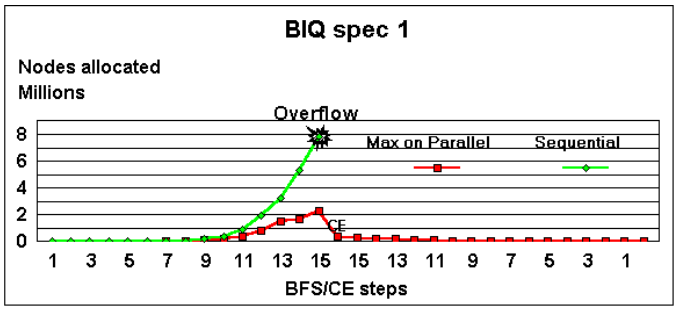

Fig. 5. Memory utilization during on-the-fly model checking of BIQ (spec 1)

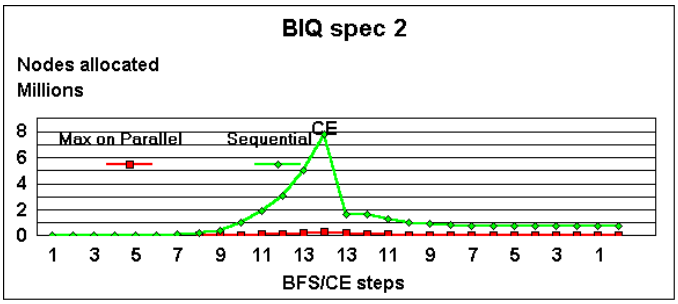

Fig. 6. Memory utilization during on-the-fly model checking of BIQ (spec 2)

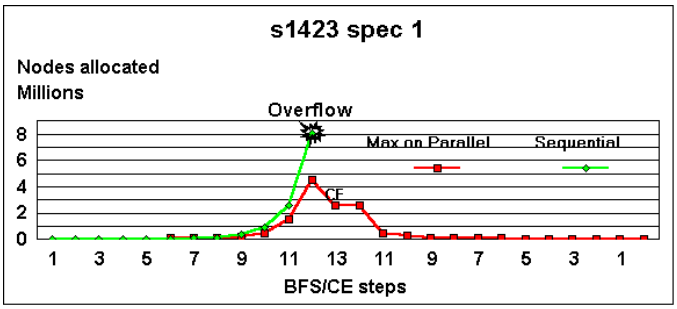

Fig. 7. Memory utilization during on-the-fly model checking of s1423 (spec 1)

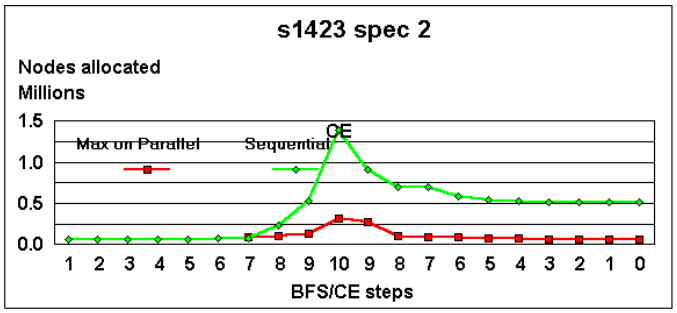

Fig. 8. Memory utilization during on-the-fly model checking of s1423 (spec 2)

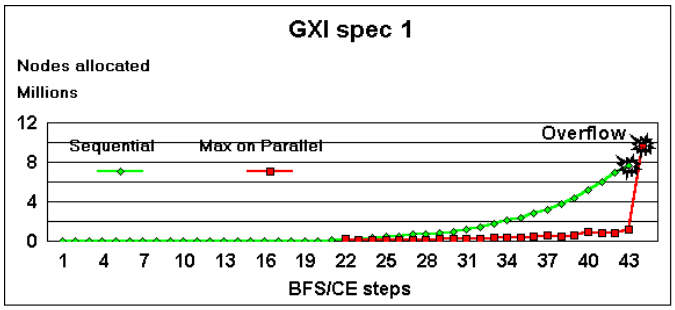

Fig. 9. Memory utilization during on-the-fly model checking of GXI (spec 1), using quantification. An overflow occurs during counterexample generation.

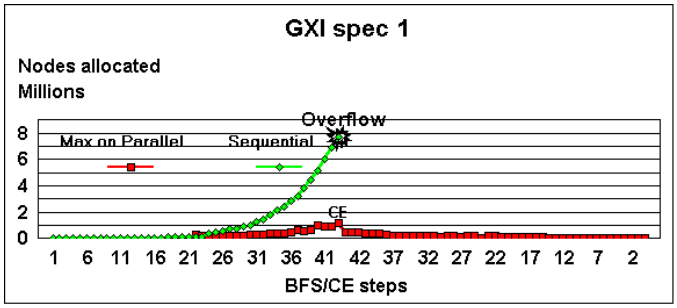

Fig. 10. Memory utilization during on-the-fly model checking of GXI (spec 1), using substitution

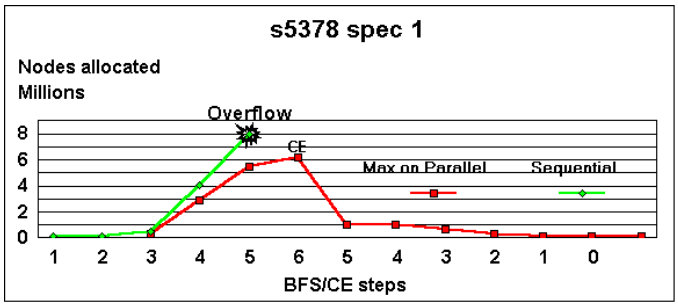

Fig. 11. Memory utilization during on-the-fly model checking of s5378 (spec 1)

Circuit spec	steps	total	Reachability		Spec check	
			seq	par	eval	CE
BIQ 1	17(15)	1,957	174	1,804	31	74
BIQ 2	15	921	184	731	24	52
s1423 1	14(12)	16,032	13	15,911	35	117
s1423 2	11	521	116	337	10	102
GXI 1	45(44)	8,468	1,866	6,570	26	138
s5378 1	7(6)	12,873	384	11,509	69	105

Fig. 12. Timing data (seconds) for parallel execution on 32×512MB machines. Each of the measures is the worst sample over all the machines. The steps count shows that the parallel system always reaches a point beyond that at which a single machine overflows (this point is given in brackets). Total is the total time over all steps, including the sequential stage, parallel reachability stage and counterexample generation time. Note that the total time is the maxima over sums and not the sum over maxima. Seq(uential) is the time it took to reach the threshold at which the parallel stage started. Par(allel) is the parallel reachability analysis time. Eval(uation) is the total time it took to evaluate, at each step, whether one of the processes found an error state. (Note that in the sequential stage, Eval is a single BDD operation, while in the parallel stage it also requires global interaction over the network). CE is the time it took to generate the counterexample.