# Scalable Distributed On-the-Fly Symbolic Model Checking

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

[1] Computer Science Department, Technion, Haifa, Israel
[2] IBM Haifa Research Laboratories, Haifa, Israel

**Abstract.** This paper presents a scalable method for parallel symbolic on-the-fly model checking on a distributed-memory environment of workstations. Our method combines a parallel version of an on-the-fly model checker for safety properties with a scalable scheme for reachability analysis. The extra load of storage required for counter example generation is evenly distributed among the processes by our memory balancing. For the sake of scalability, at no point during computation the memory of a single process contains all the data from any of the cycles. The counter example generation is thus performed through collaboration of the parallel processes. We develop a method for the counter example generation keeping a low peak memory requirement during the backward step and the computation of the inverse transition relation.
We implemented our method on a standard, loosely-connected environment of workstations, using a high-performance SMV-based model checker. Our initial performance evaluation using 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 counter examples even when the model is too large to fit in the memory of the parallel system.

## 1 Introduction

A model checking algorithm gets a system model and a specification written as a temporal logic formula. It returns 'true' if the model satisfies the formula and returns 'false' otherwise. In the latter case it also provides a counter example to demonstrate why the model does not satisfy the formula. The counter example feature is extremely important in the debugging process of the system.

Model checking tools have been successful in finding subtle errors in complex designs. Their main drawback, however, is their high space requirements that limits their applicability to large designs. Many approaches to reducing the space requirements have been investigated. One of the most successful approaches is *symbolic model checking* [6] in which the model as well as intermediate results are represented using Binary Decision Diagrams (BDDs) [4].

A different approach is *on-the-fly model checking* that develops parts of the model as needed for checking the formula. Usually, the check is guided by an automaton that *monitors* the behavior of the system in order to detect erroneous behaviors. On-the-fly algorithms [10, 17, 3] are usually based on a depth first search (DFS) traversal of the state space and therefore do not combine well with BDD-based methods. The method in [2], on the other hand, reduces model checking to reachability analysis and is therefore successfully implemented with BDDs. Furthermore, their monitoring automaton is linear in the size of the formula whereas other methods use exponential size automata. The method can handle a large class of safety properties, including RCTL (see appendix A for a detailed description).

In [7, 16, 15, 18, 11], other approaches to reducing the space requirements were introduced. These studies suggest to partition the work into several tasks: [18] shows how to parallelize an explicit-state model checker that does not use symbolic methods; [7, 16, 15] use a single computer that handles one task at a time, while the other tasks are kept in an external memory. [11] suggests a distributed, symbolic algorithm for reachability analysis that works on a network of processes with distributed memory. The algorithm in [11] achieved an average memory scale-up of 55 on 130 processes. Thus, it made it possible to handle designs that could not fit in a single machine.

In this work we combine the approaches of [2] and [11], obtaining a distributed, symbolic, on-the-fly model checking that can handle very large designs. We also provide a counter example that is computed on the distributed state space by several processes, without ever holding the whole set of states in one process.

Producing the counter example requires additional storage of sets of states during reachability, one set for each cycle. In order to balance this extra storage across the processes we apply a slicing function which may vary from one cycle to another. This turns the efficient distributed production of a counter example to be somewhat tricky: we need to track the cycles backwards while switching different slices and keeping the peak temporal space at a low level.

We implemented our method within the high-performance verification tool RuleBase [1] of IBM, Haifa. Our initial performance results, measured on a distributed, non-dedicated system of 32 standard workstations and using a slow network interconnection, 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, and the network resource does not become a bottleneck.

In addition to the above, we show that our method is more effective when applied to on-the-fly model checking including counterexample generation than when applied to reachability analysis. There are two main reasons for this phenomenon. First, note that counter example generation generally requires saving sets of states which consume more space. Effective splitting and balancing the extra space across the distributed system enhances scalability. Second, the parallel system, even when failing to complete reachability to the fix-point, is commonly able to proceed reachability for several steps beyond the point reached by a single machine. This improves the chances for our on-the-fly model checking to find an error state during those steps.

The rest of the paper is organized as follows. Section 2 describes the sequential on-the-fly algorithm for checking RCTL safety properties. Section 3 presents our distributed on-the-fly model checking scheme. Section 4 provides our initial performance evaluation. Finally, we give our conclusions in Section 5.

## 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 [2]. This algorithm is the basis for our distributed algorithm.

Given a system model $M$ and an RCTL formula $\varphi$, an automaton (satellite) $\mathcal{A}$ is constructed and combined with $M$. $\mathcal{A}$ monitors the behavior of $M$. If it detects an erroneous behavior it enters a special error state and stays there forever. Thus, $M$ satisfies $\varphi$ iff the combination of $M$ and $\mathcal{A}$, $M \times \mathcal{A}$, does not reach an error state. In order to check that $M$ satisfies $\varphi$ we therefore 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 breadth first search (BFS). Starting from the set of initial states, at each iteration it constructs a *doughnut*, which is the set of newly found states that have not been found in previous iterations. The doughnuts are kept for later use in the generation of the counter example. Having to keep the doughnuts, the space requirements of this algorithm exceed those of (pure) reachability analysis.

The model checking phase terminates successfully if all reachable states have been traversed and no error state has been found. If at any stage an error state is found, the model checking phase stops and the generation of counter example starts.

The counter example is a sequence of states leading from an initial state to an error state. It is generated starting from an error state and going backwards, following the doughnuts produced and stored by the model checking algorithm.

Figure 1 presents the sequential algorithm for on-the-fly model checking, including counter example generation. This algorithm for constracting the counterexample is based on the one in [8]. Lines 1–9 describe the model checking phase. At each iteration $i$, the set of `new` states that have not been reached before is kept in doughnut $S_i$.

The algorithm terminates either if 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 counter example $\texttt{Ce}_0, \ldots \texttt{Ce}_k$ is generated. The counter example is of length $k + 1$ (line 14) since an error state was first found in the $k$-th iteration. We choose $\texttt{Ce}_k \in S_k$ to be one of the reached error states. 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 $\texttt{Ce}_i$, pred($\texttt{Ce}_i$), and intersecting 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 bad.

The generation of the counter example is completed when $\texttt{Ce}_0$ is chosen.

```
1 reachable = new = initialStates;
2 i = 0;
3 while ((new $\neq \emptyset$)&&(new $\cap$ error = $\emptyset$)) {
4   $S_i$ = new;
5   i = i+1;
6   next = nextStateImage(new);
7   new = next $\setminus$ reachable;
8   reachable = reachable $\cup$ next;
9 }
10 if (new $= \emptyset$) {
11  print ``formula is true in the model'';
12  return;
13 }
14 k = i;
15 print ``formula is false in the model, failed at cycle k'';
16 bad = new $\cap$ error;
17 while (i>=0) {
18  Ce$_i$ = choose one state from bad;
19  if (i>0) bad=pred(state)$\cap S_{i-1}$;
20  i = i-1;
21 }
22 print ``counter examples is:'' Ce$_0 \cdots$Ce$_k$;
```

**Fig. 1.** Sequential algorithm for on-the-fly model checking, including counter examples generation

## 3 The distributed algorithm

The distributed algorithm for on-the-fly model checking also consists of two phases: the model checking phase and the phase in which the counter example is generated.

### 3.1 Distributed model checking

The distributed algorithm is composed of an initial sequential stage, and a parallel stage. In the sequential stage, the reachable states are computed on a single process as long as memory requirements are below a certain threshold. When the threshold is reached, the state space is partitioned into $k$ slices (the algorithm for slicing is described in [11]) and $k$ processes are initialized. Each process is 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.

During a single step each process computes the set `next` of states that are directly reached 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, it receives set of states it owns from other processes.

The model checking phase for one process $P_j$ is given in lines 1-13 of Figure 3.2. Lines 1-3 describe the setup stage: 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.

A *distributed termination detection* (line 5) is used in order to determine when this phase should end. *All* processes should end this phase if one of two conditions hold: Either 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. If the second case the specification is false and all processes proceed to the next phase in which a counter example will be generated.

The distributed model checking is different from the sequential one in the following points:

– The set `next` is modified (lines 9-10) as the result of communication with the other processes, and stricted to include only owned states.
– Distributed termination detection is used.
– Each process $P_j$ stores, for each doughnut $i$, the slice of the doughnut $S_{(i,j)}$ it owns.

One of the most important factors in making our distributed algorithm effective is *memory balancing* which keeps approximately equal memory requirement across the processes during the entire computation. Balance is maintained during the whole computation by pairing large slices with small ones and re-slicing their union in a balanced way.

It should be noted that as a result of memory balancing, a process owns (and stores) a different slice of the doughnuts in different iterations. This, however, does not effect the correctness of our distributed generation of the counter example. To guarantee that the distributed algorithm always finds a (correct) counter example, all we need is the following property, which is true by the 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 counter example generation

In this section we present an algorithm which generates a counter example. The algorithm uses the doughnut slices that are stored in the memory of the processes.

The algorithm consists of *local phases* and *coordination phases*. In the local phase all processes run in parallel. Each process executes the sequential algorithm for counter example generation until it cannot proceed any more. A process may get stuck after producing a suffix $\texttt{Ce}_i \ldots \texttt{Ce}_k$ of the counter example 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 $S_{i-1}$, we know that there must be a process that has a predecessor for $\texttt{Ce}_i$ in its slice of the (i-1)-th doughnut.

We would like to re-initiate the local phase in all processes from the largest suffix produced so far. In the coordination phase, a process which produced a largest suffix is selected. If this suffix is complete, i.e., it contains all of $\texttt{Ce}_0 \ldots \texttt{Ce}_k$, then the process prints its counter example 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 re-initiates the local phase from that point.

Lines 18-35 of Figure 3.2 describe the algorithm. Lines 22-26 contain the local phase while lines 27-35 contain the coordination phase. The algorithm uses the following variables. `myId` is the index of the process (`myId`=$j$ for process $P_j$). The coordination phase, starts by choosing the smallest iteration number `minIte` and then, among the processes with that number, the smallest index `minProc` of such a process.

## 3.3 Reducing peak in memory requirement

The generation of the counter example involves the computation of the sets `bad` = `pred(Ce)`$\cap S_{(i,j)}$, in which the doughnut slice $S_{(i,j)}$ is intersected with the set of predecessors of the state `Ce` (lines 24, 35). The BDDs for `Ce` and `bad` are usually small. However, a very large peak in memory requirement may occur by intermediate BDDs obtained during the computation of `bad`. In particular, the BDD for `pred` might be extremely big. This phenomenon can be viewed for instance in example GXI (Figure 7), where a significant increase in the memory use occurs once the computation of the counter example starts.

Examining more closely the computation of `bad` we notice that by changing the order of operations we can obtain smaller intermediate BDDs, thus reduce the memory peak. This can be done by first restricting the transition relation of our model to the doughnut slice $S_{(i,j)}$ and only then using it to compute `pred(Ce)`. Since our implementation is based on partitioned transition relation [5], we actually restrict each one of the partitions to the doughnut slice.

To make this precise, 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.

Let $\texttt{Ce}(V)$ be the boolean function for the singleton set consisting of the state `Ce`, and $S_{(i,j)}(V)$ be the boolean function for the slice $S_{(i,j)}$. Then the computation of `bad` can be described by:

$$\exists V' \, [\, TR(V, V') \wedge Ce(V') \,] \;\wedge\; S_{(i,j)}(V) \tag{2}$$

```
mySlice = receive(fromSingle);
reachable = receive(fromSingle);
new = receive(fromSingle);
i = 0;
while (Termination(new,error)==0) {
    S$_{(i,j)}$ = new;
    i = i+1;
    next = nextStateImage(new);
    next = sendRecieveAll(next);
    next = next $\cap$ mySlice;
    new = next $\setminus$ reachable;
    reachable = reachable $\cup$ next;
}
if (new $= \emptyset$) {
   print ``formula is true in the model'';
   return;
}
k = i;
print ``formula is false in the model, failed at cycle k'';
bad = new $\cap$ error;
while (i>=0) {
   while ((i>=0) &&(bad $\neq \emptyset$)) {
       Ce$_i$ = choose one state from bad;
       if (i>0) bad=pred(Ce$_i$)$\cap$ S$_{(i-1,j)}$;
       i = i-1;
   }
   (minIte,minProc)=chooseMinIteFromAll(i,myId);
   i = minIte;
   if (i<0) {
       if (myId == minProc)
           print ``counter examples is:'' Ce$_0 \cdots$Ce$_k$;
       return;
   }
   Ce$_{i+1} \cdots$Ce$_k$=broadcast(minProc,Ce$_{i+1} \cdots$Ce$_k$);
   bad=pred(Ce$_{i+1}$)$\cap$ S$_{(i,j)}$;
 }
```

Process $P_j$ in the distributed algorithm for on-the-fly model checking, including the generation of a counter example.

Our transition relation is partitioned, which means that it consists of partitions $N_n(V, V')$ so that $TR(V, V') = \bigwedge_n N_n(V, V')$. Consequently, the previous expression can be rewritten as:

$$\exists V' \; [ \bigwedge_n N_n(V, V') \wedge Ce(V') \; ] \; \wedge \; S_{(i,j)}(V) \tag{3}$$

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:

$$\exists V' \; [ \bigwedge_n \; ( \; N_n(V, V') \wedge S_{(i,j)}(V) \; ) \; \wedge \; Ce(V') \; ] \tag{4}$$

This expression describes the computation in which, first, each partition of the transition relation is restricted to the doughnut slice $S_{(i,j)}$, and then the predecessors of $Ce$ are computed.

This computation can be made more efficient by using the *simplify-assuming* technique [9]. Instead of intersecting each $N_n(V, V')$ with $S_{(i,j)}(V)$ we simplify $N_n(V, V')$ assuming $S_{(i,j)}(V)$ and intersect only the final result with $S_{(i,j)}(V)$.

The improvement described above uses precise information in order to restrict the partitions of the transition relation. This requires, however, to compute a different restriction (with respect to different slice) in each step of the counter example generation.

We next suggest a different restriction that can be computed only once for each process. Process $P_j$ restricts $N_n(V, V')$ to $U_j$, where $U_j = \cup_i S_{(i,j)}$ is the union of all the doughnut slices owned by $P_j$. This restriction is expected to give a similar improvement in space reduction.

We now prove that the computation of `bad` with the new restriction results in exactly the same set of states. Since $S_{(i,j)} \subseteq U_j$, the expression in (4) is equivalent to:

$$\exists V' \; [ \bigwedge_n \; ( \; N_n(V, V') \wedge U_j(V) \wedge S_{(i,j)}(V) \; ) \; \wedge Ce(V') \; ] \tag{5}$$

This, in turn, is equivalent to:

$$\exists V' \; [ \bigwedge_n \; ( \; N_n(V, V') \wedge U_j(V) \; ) \; \wedge Ce(V') \; ] \; \wedge \; S_{(i,j)}(V) \tag{6}$$

This final expression describes the computation in which each partition is restricted to $U_j$ and then used in finding the predecessors of $Ce$.

Note that, $N_n(V, V') \wedge U_j(V)$ can be computed only once at the beginning of the counter example generation. Here again a better improvement may be obtained by simplifying $N_n(V, V')$ assuming $U_j(V)$.

We next suggest an orthogonal improvement that exploits the fact that we compute the set of predecessors of a singleton ($Ce(V')$ contains only one state $Ce$). We replace the intersection of $N_n(V, V')$ and $Ce(V')$ by assigning $Ce$ to $V'$ in $N_n$. The existential quantifier is then redundant and can be removed. Applying these operation, equation (3) can first be rewritten as

$$\exists V' \; [ \bigwedge_n \; ( \; N_n(V, Ce) \; ) \; ] \; \wedge S_{(i,j)}(V) \tag{7}$$

Removing the redundant quantification, we get

$$\bigwedge_n \ ( \ N_n(V, Ce) \ ) \ \wedge S_{(i,j)}(V) \tag{8}$$

To combine the two optimizations we can compute (3) as

$$\bigwedge_n \ \big( \ N_n(V, Ce) \wedge S_{(i,j)}(V) \ \big) \tag{9}$$

This expression describes the computation in which, first the state $Ce$ is assigned to each partition of the transition relation, the result is then restricted to the doughnut slice $S_{(i,j)}$, and finally the intersection of all the partitions is computed.

## 4 Experimental results

In this section we report initial performance evaluation of our approach. We implemented our On-The-Fly model checker and embedded it in an enhanced version of McMillan's SMV [14], developed at IBM Haifa Research Laboratory [1].

Our parallel testbed includes 32 RS6000 machines, each consisting of a 225MHz PowerPC processor and 512MB memory. The communication between the nodes consists of a 16Mbit/second token ring. The nodes are non-dedicated; i.e., they are mostly workstations of employees who would often use them (and the network) at the same time that we ran our experiments.

We experimented using two of the largest circuits we found in the benchmarks ISCAS89 +addendum'93. In order to test the counter examples 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. For these examples we used the original properties. We mapped properties to automata using the IBM implementation as described in [2]. Characteristics of the circuits and the automata are given in Figure 2.

### 4.1 Parallel On-The-Fly model checking – Space Reduction

We now present the results for On-The-Fly model checking of the benchmark suit using our 32 machine testbed. Figures 3 to 8 summarize the memory usage, giving the store size and peak usage for every step. Each of the graphs compares the memory usage in the single-machine execution to that of the parallel system. For the parallel system we give both average and highest (peak) memory utilization in any of the machines.

We give examples for four models and six properties. For two of the models, BIQ and S1423, two properties are checked: one that overflows using a single machine, and another one which completes the computation even when using only a single machine.

As can be seen in Figure 3, an overflow occurs at cycle 15 while searching for an error state in BIQ using a single machine. The overflow occurs because counter example generation (CE) requires saving the doughnuts which consume a lot of memory. In contrast, the parallel algorithm does not overflow, and finds the error state in BIQ at cycle 17, at which point counter example generation begins.

At the first counter example cycle we see a drop in the memory store size. This drop is characteristic to all examples, and 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 require less memory. The reason for that is 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 the counter example generation.

The parallel algorithm finds an error state in cycle 14 of S1423 as can be seen in Figure 5. In this case, finding the error state On-The-Fly is essential, since we were not able to finish reachability on this example even using our parallel system.

Figure 7 demonstrates an incompleteness of our implementation which we did not make it to repair by the submission deadline. In this example, a step backwards from a single state using the original transition relation may result in a very large set which includes a lot of unreachable states. Thus, the processes which are busy in generating the counter example require a lot more memory than the others, resulting in a large difference of the maximum and the average memory utilization. The method described in Section 3.3 ensures that this will not happen by intersecting each partition of the transition releatoin with the local doughnuts during the backward step.

### 4.2 Parallel On-The-Fly model checking – Timing and Communication

Figure 9 gives the timing breakdown for On-The-Fly model checking on our benchmark suit. The parallel reachability stage takes most of the computation time. As shown in [11], communication does not become a bottleneck at that 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 solving joint goals. Our methods can be seen as a globalization of memory systems, using the network as an intermediate level of the memory hierarchy. This intermediate level resides between the main memory and the disk: on the one hand it is a lot faster than the disk (the difference in latency and bandwidth is between three to five orders of magnitude); on the other hand, it is a lot slower than the main memory module (about three orders of magnitude for very fast networks), so locality is still an important issue.

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 that deserves more attention.

It is important to note that our method was integrated into a high-performance model checker, thus proving its industrial potential. This fact also shows that our parallelization method is orthogonal in many respects to other important optimizations, and does not hurt their applicability.

## 6 Acknowledgments

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

## References

1. 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.
2. 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.
3. 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.
4. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
5. 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. Winner of the Sidney Michaelson Best Paper Award.
6. 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.
7. Gianpiero Cabodi, Paolo Camurati, and Stefano Que. 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.
8. 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.
9. Olivier Coudert, Jean C. Madre, and Christian 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.
10. 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.
11. 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.
12. J.E. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesely Pub. Co, 1979.
13. D. E. Long. *Model Checking, Abstraction, and Compositional Reasoning*. PhD thesis, Carnegie Mellon University, 1993.
14. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
15. Adrian A. Narayan, Jain J. Jawahar Isles, Robert K. Brayton, and Alberto 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.

16. Adrian A. Narayan, Jain Jawahar, M. Fujita, and A. Sangiovanni-Vincentelli. Partitioned-ROBDDs. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 547–554. IEEE Computer Society Press, June 1996.
17. 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.
18. Ulrich Stern and David 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, thus describing the set of BAD computations rather than the good computations. A nice way to describe a set of finite bad computations is by using *Regular Expressions*(RE), in a special way, as described in the following. Let $W$ be a finite set of symbols (in our case: signal names of the model under test). The alphabet $\Sigma$ over which the regular expressions are expressed, 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. $\Sigma$ in this case consists of all 16 possible Boolean functions ($true, false, req, \neg req, req \vee ack$ etc.). Describing the bad computations of this property, we say 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 $A_r$ [12], and check that $L(M \cap A_r) = \Phi$.

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

$$AG(F_r \neq q_1 \wedge ... \wedge F_r \neq q_n)$$

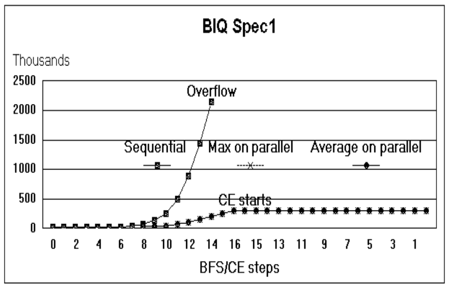
against the model $M \times F_r$.

Note that the formula to check is of the form $AG(p)$, where $p$ is a boolean formula, and thus can be checked On-The-Fly [13, 2], saving a lot of time and space. Therefore model-checking of Regular Expressions is more efficient in most cases than model-checking of CTL formulas.

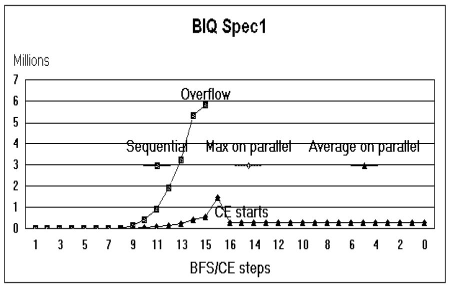
Regular Expressions as described above, have a different expressive power than temporal logics. Beer at al in [2] discuss the relations between RE and CTL, and give an algorithm to translate a subset of CTL formulas to Regular Expression specifications. The subset of CTL which can be translated to RE is called $RCTL$.

| Circuit | #vars | | max store size | | peak | | spec check | | | gc |
|---|---|---|---|---|---|---|---|---|---|---|
| | model | sat | size | step | size | step | time | steps | CE | time |
| BIQ | | | | | | | | | | |
| $SPEC1 : \{[*], (writePtr(0..3) = place(0..3))\&(dataIn(0) = D(0)), goto((readPtr(0..3) = place(0..3))\|Complete)\} (AX[4](DataOut(0) = D(0)))$ $SPEC2 : \{[*], (writePtr(0..3) = place(0..3))\&(dataIn(0) = D(0)), goto((readPtr(0..3) = place(0..3))\|Complete)\} (AX[2](DataOut(0) = D(0)))$ | | | | | | | | | | |
| SPEC 1 | 102 | 5 | 2,145,201 | 14 | 5,852,485 | 15 | 15,059 | Ov(15) | | 1,816 |
| SPEC 2 | 102 | 5 | 2,145,201 | 14 | 5,332,126 | 14 | 3,811 | 15 | 95 | 1,172 |
| s1423 | | | | | | | | | | |
| $SPEC1 : AG(G729\&G726 \rightarrow AX[10](G726))$ $SPEC2 : AG(G729\&G726 \rightarrow AX[7](G726))$ | | | | | | | | | | |
| SPEC 1 | 91 | 4 | 2,574,709 | 11 | 8,640,069 | 12 | 2,024 | Ov(12) | | 366 |
| SPEC 2 | 91 | 3 | 914,046 | 10 | 1,540,999 | 10 | 625 | 11 | 58 | 132 |
| GXI | | | | | | | | | | |
| SPEC 1: {[*],PACKET-START0& ADDR-DATA(0..2)=slot, true, RESP-DATA-IN(0..3)=0010B} (ABF[2.. 32](PACKET-START0 & ADDR-DATA(0..2)=slot)) | | | | | | | | | | |
| SPEC 1 | 292 | 6 | 3,880,164 | 43 | 8,141,305 | 44 | 16,222 | Ov(44) | | 3,258 |
| s5378 | | | | | | | | | | |
| $SPEC1 : AG(n3104gat \rightarrow AX[6]((n3106gat)\text{before}(n3104gat)))$ | | | | | | | | | | |
| SPEC 1 | 188 | 4 | 3,914,409 | 5 | 9,663,200 | 6 | 4,440 | Ov(6) | | 679 |

**Fig. 2.** #vars gives the number of variables in the model and the sat(ellite). All sizes are given in BDD nodes, and all times in seconds. Let reachable be the set of nodes already reached, new – the set of nodes reached but not yet developed, doughnuts – the list of new in preceding steps. Then max store size is the maximal (over the steps) of (reachable + new + doughnuts). The peak is the maximal size at any point during a step. In order to mask the effect of garbage collection (gc) scheduling decisions, the peak is measured after gc invocations. Spec check is the number of steps/time it takes to find an error state, and the time it takes to generate a Counter Example (CE). Ov($x$) means memory overflow during step $x$. All measurements were done using an RS6000 machine, consisting of a 225MHz PowerPC processor with 512MB memory.

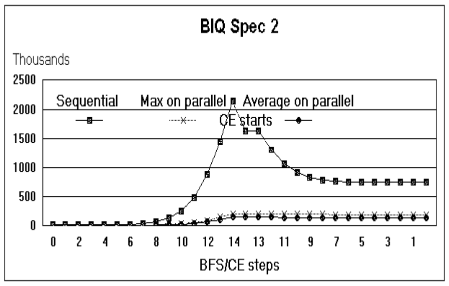


(a) **Size of store**

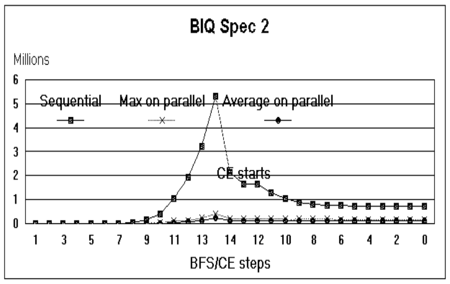


(b) **Nodes allocated (peak)**

**Fig. 3.** Memory utilization during On-The-Fly model checking of BIQ spec 1

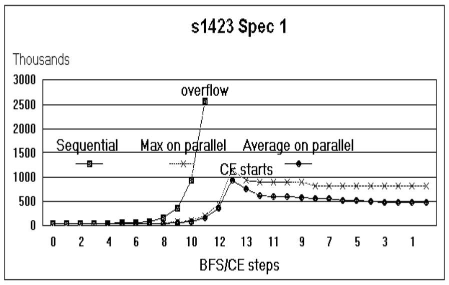


(a) **Size of store**

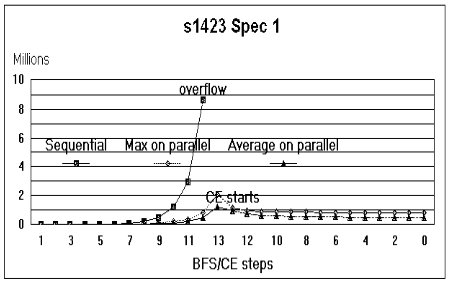


(b) **Nodes allocated (peak)**

**Fig. 4.** Memory utilization during On-The-Fly model checking of BIQ, spec 2

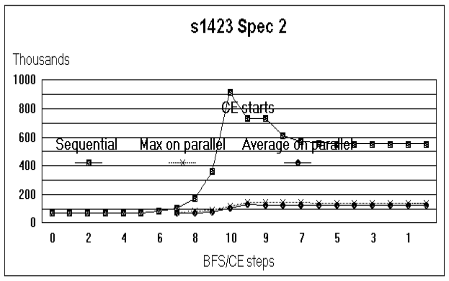


(a) **Size of store**

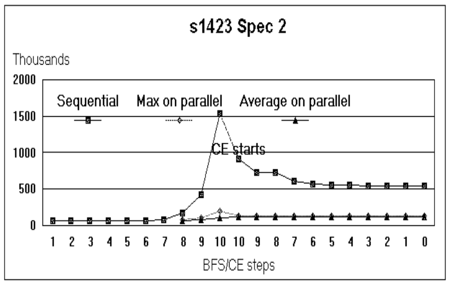


(b) **Nodes allocated (peak)**

**Fig. 5.** Memory utilization during On-The-Fly model checking of s1423, spec1

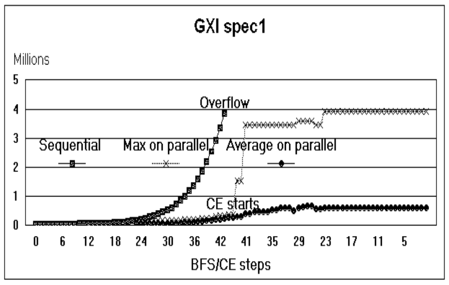


(a) **Size of store**

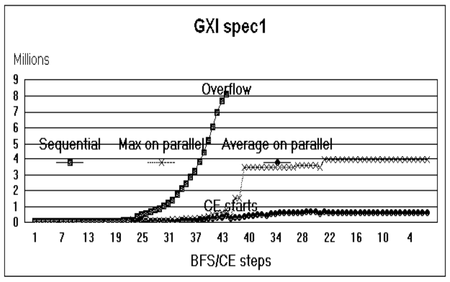


(b) **Nodes allocated (peak)**

**Fig. 6.** Memory utilization during On-The-Fly model checking of s1423, spec2

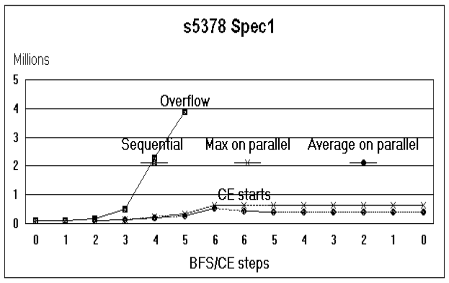


(a) **Size of store**

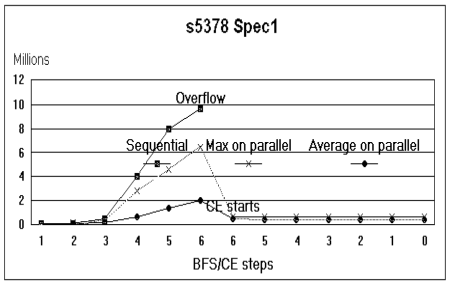


(b) **Nodes allocated (peak)**

**Fig. 7.** Memory utilization during On-The-Fly model checking of GXI, spec 1



(a) **Size of store**



(b) **Nodes allocated (peak)**

**Fig. 8.** Memory utilization during model checking of s5378, spec 1

| Circuit | SPEC | steps | total | Reachability | | Spec check | | gc |
|---|---|---|---|---|---|---|---|---|
| | | | | seq | par | eval | CE | |
| BIQ | 1 | 17(15) | 1,957 | 174 | 1,804 | 31 | 74 | 159 |
| | 2 | 15 | 921 | 184 | 731 | 24 | 52 | 88 |
| s1423 | 1 | 14(12) | 16,032 | 13 | 15,911 | 35 | 117 | 1,950 |
| | 2 | 11 | 521 | 116 | 337 | 10 | 102 | 52 |
| GXI | 1 | 45(44) | 10,268 | 1,866 | 6,570 | 50 | 1,738 | 447 |
| s5378 | 1 | 7(6) | 12,873 | 384 | 11,509 | 69 | 105 | 903 |

**Fig. 9.** 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 gets farther from where a single-machine overflows (given in brackets). Total is the total time over all steps, including the sequential stage, parallel reachability stage, counter example generation, and garbage collection (gc) 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 get to the threshold at which point the parallel stage started. Par(allel) is the parallel reachability analysis time. Eval(uation) is the total time 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 to generate the counter example.