Saturation NOW[*]

Ming-Ying Chung and Gianfranco Ciardo

Department of Computer Science and Engineering

University of California, Riverside

{chung,ciardo}@cs.ucr.edu

Abstract

We present a distributed version of the saturation *algorithm for symbolic state-space generation of discrete-state models. The execution is strictly sequential but utilizes the overall available memory. Thanks to a* level-based *allocation of the decision diagram nodes onto the workstations, no additional node or work is created. A dynamic memory load balancing heuristic helps coping with the uneven growth of the decision diagram levels allocated to each workstation. Experiments on a conventional network of workstations show that the runtime of our distributed implementation is close to the sequential one even when balancing is triggered, while it is of course much better when the sequential implementation is forced to rely on virtual memory.*

1 Introduction

Formal verification has been widely used in industry for quality assurance. In particular, model checking [1] can be used to detect design errors in a finite-state machine, and an exhaustive state-space generation is usually a first step. The increasing complexity of concurrent systems and protocols stresses the limits of most software verification tools. Symbolic representations like binary decision diagrams (BDDs) [2] help, but model checking for complex systems remains very memory-intensive. Reliance upon virtual memory can greatly increase the execution time, even when the algorithm attempts to improve the efficiency of memory swapping [3]. Over the years, much research has then focused on exploiting the computational resources available in parallel and distributed architectures.

To cope with the large memory requirements of state-space generation, we present a distributed message-passing algorithm based on the *saturation* algorithm [4], which generates the state-space using a network of workstations (NOW) and exploits the overall available distributed memory. With saturation, sets of states are encoded via multi-way decision diagrams (MDDs) and the transition relation is encoded via a boolean Kronecker matrix representation. To retain the compactness of the MDD representation and the efficiency of *event locality*, we perform a (horizontal) *level-based allocation* of the MDD nodes onto the available workstations. Although our distributed approach is purely sequential, it partitions the memory load for state-space generation onto the workstations of the NOW, thus it achieves good scalability.

The rest of the paper is organized as follows. Section 2 gives the necessary background on reachability analysis, MDDs, Kronecker encoding, and saturation. Section 3 describes how we allocate the MDD nodes onto workstations and the overall algorithm. Section 4 illustrates our dynamic memory load balancing. Section 5 shows experimental results. Section 6 discusses related work.

2 Symbolic state space generation

A discrete-state model is a triple $(\widehat{\mathcal{S}}, \mathbf{s}^{init}, \mathcal{N})$, where $\widehat{\mathcal{S}}$ is the set of *potential states* of the model, $\mathbf{s}^{init} \in \widehat{\mathcal{S}}$ is the *initial state*, and $\mathcal{N} : \widehat{\mathcal{S}} \rightarrow 2^{\widehat{\mathcal{S}}}$ is the *next-state function* specifying the states reachable from each state in a single step.

Since we target asynchronous systems, we decompose

[*]Work supported in part by the National Aeronautics and Space Administration under grant NAG-1-02095 and by the National Science Foundation under grants CCR-0219745 and ACI-0203971.

$\mathcal{N}$ into a disjunction of next-state functions [5]: $\mathcal{N}(\mathbf{i}) = \bigcup_{e\in\mathcal{E}} \mathcal{N}_e(\mathbf{i})$, where $\mathcal{E}$ is a finite set of *events* and $\mathcal{N}_e$ is the next-state function associated with event e, i.e., $\mathcal{N}_e(\mathbf{i})$ is the set of states the system can enter when e occurs, or *fires*, in state $\mathbf{i}$. An event e is said to be *disabled* in $\mathbf{i}$ if $\mathcal{N}_e(\mathbf{i}) = \emptyset$; otherwise, it is *enabled*.

The *reachable state space* $\mathcal{S} \subseteq \widehat{\mathcal{S}}$ is the smallest set containing $\mathbf{s}^{init}$ and closed with respect to $\mathcal{N}$:

$$\mathcal{S} = \{\mathbf{s}^{init}\} \cup \mathcal{N}(\mathbf{s}^{init}) \cup \mathcal{N}(\mathcal{N}(\mathbf{s}^{init})) \cup \cdots = \mathcal{N}^*(\mathbf{s}^{init}),$$

where "*" denotes the reflexive and transitive closure and we let $\mathcal{N}(\mathcal{X}) = \bigcup_{s\in\mathcal{X}} \mathcal{N}(s)$.

The systems we target can be decomposed into interacting *submodels*. For a model composed of K submodels, a *global* system state is a K-tuple $(\mathbf{i}_K, \ldots, \mathbf{i}_1)$, where $\mathbf{i}_k$ is the *local* state of submodel k, for $K \geq k \geq 1$. Thus, the potential state space $\widehat{\mathcal{S}}$ is given by the cross-product $\mathcal{S}_K \times \cdots \times \mathcal{S}_1$ of K *local state spaces*. This decomposition allows us to use techniques targeted at exploiting system structure, in particular, *symbolic* state space storage techniques based on decision diagrams.

2.1 Multiway decision diagrams for $\mathcal{S}$

While not a requirement (we showed in [6] that it is possible to generate the local state spaces $\mathcal{S}_k$ *on the fly* by interleaving symbolic global state-space generation with explicit local state-space generation), we assume that each $\mathcal{S}_k$ is known a priori. Then, we use the mappings $\psi_k : \mathcal{S}_k \to \{0, 1, \ldots, n_k - 1\}$, with $n_k = |\mathcal{S}_k|$, and ignore the difference between a local state $\mathbf{i}_k$ and its index $i_k = \psi_k(\mathbf{i}_k)$. Thus, from now on, we identify $\mathcal{S}_k$ with $\{0, 1, \ldots, n_k - 1\}$ and encode any set of states $\mathcal{X} \subseteq \widehat{\mathcal{S}}$ via a (*quasi-reduced ordered*) MDD over $\widehat{\mathcal{S}}$, i.e., a directed acyclic edge-labeled multi-graph where:

- Nodes are organized into $K + 1$ *levels*. We write $\langle k|p\rangle$ to denote a generic node, where k is the level and p is a unique index for that level.
- Level K contains only a single *non-terminal* node $\langle K|r\rangle$, the *root*, whereas levels $K-1$ through 1 contain one or more non-terminal nodes.
- Level 0 consists of two *terminal* nodes, $\langle 0|0\rangle$ and $\langle 0|1\rangle$.
- A non-terminal node $\langle k|p\rangle$ has n_k arcs pointing to nodes at level $k-1$. If the i^{th} arc, for $i \in \mathcal{S}_k$, is to node $\langle k-1|q\rangle$, we write $\langle k|p\rangle[i] = q$. *Duplicate* nodes are not allowed but, unlike the *reduced* case, *redundant* nodes where all arcs point to the same node are allowed (both versions are canonical).

The set of states encoded by the MDD is $\mathcal{B}(K, r)$, where $\mathcal{B}$ is recursively defined as:

$$\mathcal{B}(k,p) = \begin{cases} \{i_1 : \langle 1|p\rangle[i_1] = 1\} & \text{if } k = 1 \\ \bigcup_{i_k\in\mathcal{S}_k} \{i_k\} \times \mathcal{B}(k-1, \langle k|p\rangle[i_k]) & \text{if } k > 1 \end{cases} .$$

We reserve indices 0 and 1 at each level k, so that $\mathcal{B}(k, 0) = \emptyset$ and $\mathcal{B}(k, 1) = \mathcal{S}_k \times \cdots \times \mathcal{S}_1$ [4].

2.2 Kronecker encoding for $\mathcal{N}$

We adopt a Kronecker representation of $\mathcal{N}$ inspired by work on Markov chains [7]. We use a *consistent* model partition [4, 8, 9], where each $\mathcal{N}_e$ is decomposed into K local next-state functions $\mathcal{N}_{k,e}$, for $K \geq k \geq 1$, satisfying $\forall (i_K,\ldots,i_1) \in \widehat{\mathcal{S}}, \mathcal{N}_e(i_K,\ldots,i_1) = \mathcal{N}_{K,e}(i_K) \times \cdots \times \mathcal{N}_{1,e}(i_1)$. By defining $K \cdot |\mathcal{E}|$ matrices $\mathbf{N}_{k,e} \in \{0,1\}^{n_k \times n_k}$ indexed consistently with ψ_k, where $\mathbf{N}_{k,e}[i_k, j_k] = 1 \Leftrightarrow \mathbf{j}_k \in \mathcal{N}_{k,e}(\mathbf{i}_k)$, the next-state function is encoded as an incidence matrix given by the (boolean) sum of Kronecker products $\sum_{e\in\mathcal{E}} \bigotimes_{K\geq k\geq 1} \mathbf{N}_{k,e}$. The $\mathbf{N}_{k,e}$ matrices are extremely sparse: for standard Petri nets, each row contains at most one nonzero entry.

2.3 Saturation

In addition to efficiently representing $\mathcal{N}$, the Kronecker encoding allows us to exploit event *locality* [8, 9] and employ saturation [4]. We say that event e is *independent* of level k if $\mathbf{N}_{k,e} = \mathbf{I}$, the identity matrix. Let $Top(e)$ and $Bot(e)$ denote the highest and lowest levels on which e depends.

A node $\langle k|p\rangle$ is said to be saturated if it is a fixed point with respect to all $\mathcal{N}_e$ such that $Top(e) \leq k$, $\mathcal{B}(k,p) = \mathcal{B}(k,p) \cup \mathcal{N}_{\leq k}(\mathcal{B}(k,p))$, where $\mathcal{N}_{\leq k} = \bigcup_{e:\,Top(e)\leq k} \mathcal{N}_e$. To saturate a node $\langle k|p\rangle$ once all its descendants have been saturated, we *modify it in place* so that it encodes also any state in $\mathcal{N}_{k,e} \times \cdots \times \mathcal{N}_{1,e}(\mathcal{B}(k,p))$, for any event e such that $Top(e) = k$. This can create new nodes at levels below k, which are then saturated immediately prior to completing the saturation of $\langle k|p\rangle$.

If we start with the MDD that encodes the initial state $\mathbf{s}^{init}$ and saturate its nodes bottom up, we know that, at the end, the root $\langle K|r\rangle$ encodes the reachable state space $\mathcal{S} = \mathcal{N}^*(\mathbf{s}^{init})$, because:

- $\mathcal{B}(K, r) \supseteq \{\mathbf{s}^{init}\}$, since saturation only adds states,
- $\mathcal{B}(K, r) \supseteq \mathcal{N}_{\leq K}(\mathcal{B}(K, r)) = \mathcal{N}(\mathcal{B}(K, r))$, by definition of saturated node, and
- $\mathcal{N}^*(\mathbf{s}^{init}) \supseteq \mathcal{B}(K, r)$, since saturation only adds states reachable through legal event firings.

3 Distributed message-passing saturation

Results in [4, 6, 10, 11] consistently show that saturation greatly outperforms traditional breadth-first strategies by many orders of magnitude in both memory and time, making it arguably the most efficient generation algorithm for globally-asynchronous locally-synchronous discrete event systems. Thus, it makes sense to attempt its parallelization, while parallelizing the less efficient breadth-first approaches would not offset the enormous speedups and memory reductions of saturation.

We target large models where a single workstation must rely on virtual memory or, worse, runs out of memory, and choose to distribute the algorithm on a NOW with W workstations. Our solution effectively increases the available memory by a factor W, at the cost of an acceptable communication overhead. However, saturation is an inherently sequential algorithm. As it has been the case in the past for BDD-based message-passing algorithms, we do not achieve a speedup with respect to a single ideal workstation with W times as much memory. Indeed, a theoretical speedup is not in the scope of the algorithm we propose, since there is exactly one *active* workstation at any time, i.e., actually computing the union of two nodes or firing an event on a node; all other workstations are waiting for a work request, or for a reply to a work request. Our goal is to *balance the memory load* with *negligible memory overhead*, to achieve *nearly ideal memory scalability*.

We discuss the algorithm first, postponing dynamic memory load balance issues to Section 4.

3.1 Level-based MDD node allocation

We assume that $W \leq K$ workstations are available in the NOW, numbered from W down to 1. Each workstation w has two *neighbors*: one "below", $w - 1$ (unless $w = 1$), and one "above", $w + 1$ (unless $w = W$). We evenly allocate the K MDD levels to the W workstations accordingly. Initially, w owns the MDD levels $\lfloor w \cdot K/W \rfloor$ through $\lfloor (w-1) \cdot K/W \rfloor + 1$ included, and its two local variables $mytop$ and $mybot$ are initialized to the highest- and lowest-numbered levels it owns, respectively. To balance the load, the allocation of levels can change during execution, but it is always true that $mytop = K$ for workstation W, $mybot = 1$ for workstation 1, and $mytop \geq mybot$ for any workstation.

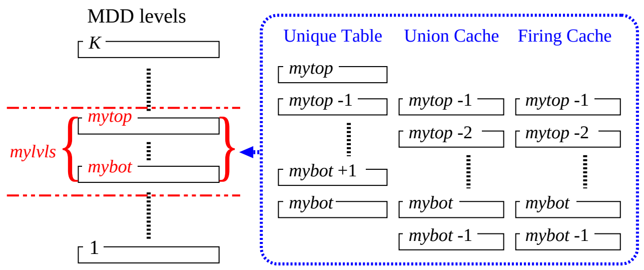

Figure 1: Allocation of levels of UT, FC, and UC.

As a first step in our distributed saturation algorithm, each workstation independently generates the Kronecker matrices $\mathbf{N}_{k,e}$ encoding $\mathcal{N}$, without requiring any synchronization. This is a simplification made possible by the fact that these matrices require little space and can be generated efficiently in isolation. In principle, since workstation w needs only those matrices $\mathbf{N}_{k,e}$ for which $mytop \geq k \geq mybot$, it could generate just thse matrices *on-the-fly* [6] and exchange them when reallocating levels.

3.2 Unique table and operation cache

Efficient canonicity enforcement requires a *unique table*, a hash table which, given $[q_0, ..., q_{n_k - 1}]$, determines whether there is a node $\langle k|p\rangle$ with $\langle k|p\rangle[i_k] = q_{i_k}$, for $0 \leq i_k < n_k$. Especially with quasi-reduced decision diagrams, it is natural to partition the unique table into K disjoint tables, one per level. In our approach, then, each workstation owns the unique tables $UT[mytop]$ through $UT[mybot]$ and can, and needs to, modify only nodes referenced in them. Analogously, efficient manipulation of decision diagrams requires an *operation cache*. The saturation algorithm, in particular, caches the results of the $Fire$ and $Union$ operations, which are also the only ones that require to change the contents of the unique table. We organize these caches by levels as well.

Since the caches referring to nodes at level $k-1$ are needed when working at level k, their ownership is shifted by one with respect to that of the unique tables. Each workstation owns the operation caches for levels $mytop - 1$ through $mybot - 1$, namely $FC[mytop-1]$ through $FC[mybot-1]$ and $UC[mytop-1]$ through $UC[mybot-1]$, with the exception of workstation 1, since there is no cache at level 0. No cache at level K is needed by the saturation algorithm. Fig. 1 illustrates the ownership of unique tables and caches.

3.3 The algorithm

Fig. 2 contains the pseudo-code for our distributed saturation algorithm, which we implemented in the tool SMART (we use "SMARTNOW" to refer to this implementation). The key routines are:

$DistGenerate()$. The main routine, each workstation runs it after generating the $\mathbf{N}_{k,e}$ matrices.

$Fire(\text{in } e{:}event,\ l{:}lvl,\ q{:}idx){:}idx$. Build the MDD encoding $\mathcal{N}^*_{\leq l}(\mathcal{N}_e(\mathcal{B}(l,q)))$.

$Union(\text{in } k{:}lvl,\ p{:}idx,\ q{:}idx){:}idx$. Build the MDD encoding $\mathcal{B}(k,p) \cup \mathcal{B}(k,q)$.

$Saturate(\text{in } k{:}lvl,\ pidx){:}idx$. Update $\langle k|p\rangle$ in place, so that it encodes $\mathcal{N}^*_{\leq k}(\mathcal{B}(k,p))$.

In addition, the following functions are assumed:

$CheckIn(\text{in } k{:}lvl,\ \text{inout } p{:}idx)$. If, for each $i \in \mathcal{S}_k, \langle k|p\rangle[i] = 0$, delete $\langle k|p\rangle$ and set p to 0. If, for each $i \in \mathcal{S}_k, \langle k|p\rangle[i] = 1$, delete $\langle k|p\rangle$ and set p to 1. If $\langle k|p\rangle$ duplicates an existing node $\langle k|q\rangle$, delete $\langle k|p\rangle$ and set p to q. Otherwise, insert $\langle k|p\rangle$ in the unique table $UT[k]$ and leave p unchanged.

$Cache(\text{in } c{:}cache,\ p{:}idx,\ q{:}idx,\ \text{out } r{:}idx)$ or $Cache(\text{in } c{:}cache,\ p{:}idx,\ e{:}event,\ \text{out } r{:}idx)$. Associate the result r with $\{p,q\}$ in the union cache c, or with (p,e) in the firing cache c.

$Find(\text{in } c{:}cache,\ p{:}idx,\ q{:}idx\ \text{inout } r{:}idx)$ or $Find(\text{in } c{:}cache,\ p{:}idx,\ e{:}event\ \text{inout } r{:}idx)$. Search $\{p,q\}$ in the union cache c, or (p,e) in the firing cache c. If found, set r to the associated value and return $true$. Otherwise, return $false$.

$SndLvl(\text{in } w{:}pid,\ k{:}lvl)$. Send $UT[k]$, $UC[k-1]$, and $FC[k-1]$ to workstation w and then decrease $mytop$, if $k = mytop$, or increase $mybot$, if $k = mytop$.

$RcvLvl(\text{in } w{:}pid,\ k{:}lvl)$. Receive $UT[k]$, $UC[k-1]$, and $FC[k-1]$ from workstation w and then increase $mytop$, if $k = mytop + 1$, or decrease $mybot$, if $k = mybot - 1$.

$Bot(e{:}event)$. Return the lowest level affected by event e.

The linear organization of the W workstations suffices, since each workstation only needs to communicate with its two neighbors to perform all required operations. When a workstation $w > 1$ would normally issue a recursive function call to level $mybot-1$, it sends instead a request to perform this operation to its neighbor below, $w-1$, and waits for a reply. Thus, except for $DistGenerate$, which is started independently by each workstation, any work is "triggered from above". After saturating the node $\langle mytop|0\rangle$, which initially encodes $(\mathbf{s}^{init}_{mytop}, \ldots, \mathbf{s}^{init}_1)$, and passing the result[1] to workstation $w+1$, each workstation $w < W$ is idle until it receives a request from $w+1$.

3.4 Message types

Table 1 shows the message types we use. For any workstation $w > 1$, $DistGenerate$ directly receives one message of type $INIT$, containing the result of the saturation of node $\langle mybot-1|0\rangle$, i.e., node $\langle mytop|0\rangle$ of workstation $w-1$. Each workstation $w < W$ also retrieves work requests messages in the repeat forever loop of $DistGenerate$. If a message is present, a case on the type of request is used to call the appropriate routine and send the reply to workstation $w+1$.

Finally, two special message types are used to stop the execution: $HALT$, originated by workstation W if the root is correctly saturated, and $ABRT$, originated by any workstation if it determines that the memory requirements exceed even those of the cluster (this includes the case where a single level does not fit in a single workstation, since our algorithm does not split a level across multiple workstations). For brevity, function $Signal$ for these two message types

[1]The index of the node encoding the saturation of a generic node $\langle k|p\rangle$ is usually still p, but not always. It can be q, if there exists already a node $\langle k|q\rangle$ satisfying $\mathcal{N}^*_{\leq k}(\mathcal{B}(k,p)) = \mathcal{B}(k,q)$, or it can be 1, if $\mathcal{N}^*_{\leq k}(\mathcal{B}(k,p)) = \mathcal{S}_k \times \cdots \times \mathcal{S}_1$.

```
$DistGenerate()$
  declare $r, p : idx$; $k : lvl$; $i : id$;
   1. if $w > 1$ then
   2.   while $Rcv(w-1, p) = FAIL$ do          • if not $FAIL$, it is $INIT$
   3.     $RcvLvl(w-1, mybot-1)$;
   4. else $p \leftarrow 1$;
   5. for $k = mybot$ to $mytop$ do
   6.   $r \leftarrow NewNode(k)$;
   7.   $\langle k|r \rangle[0] \leftarrow p$;          • $\psi_k(\mathbf{s}_k^{init}) = 0$ in our schema
   8.   $p \leftarrow Saturate(k, r)$;
   9.   if $p = NOMEM$ then
  10.     if $w = W$ or $k = mybot$ then $Signal(ABRT)$;
  11.     $p \leftarrow r$;
  12.     while $mytop \geq k$ do
  13.       $Snd(w+1, FAIL)$;          • ran out of memory
  14.       $SndLvl(w+1, mytop)$;
  15. if $w < W$ then
  16.   $Snd(w+1, INIT, p)$;          • successful
  17.   repeat forever          • execute work from above
  18.     case $Rcv(w+1, parms)$ of
  19.       $FIRE$: $r \leftarrow Fire(parms)$;
  20.               if $r = NOMEM$ then $Snd(w+1, FAIL)$;
  21.               else $Snd(w+1, RESP, r)$;
  22.       $UNIN$: $r \leftarrow Union(parms)$;
  23.               if $r = NOMEM$ then $Snd(w+1, FAIL)$;
  24.               else $Snd(w+1, RESP, r)$;
  25.       $PUTL$: $SndLvl(w+1, mytop)$;
  26.       $GETL$: $RcvLvl(w+1, mytop+1)$;
  27. else
  28.   $CheckIn(K, p)$;
  29.   $Signal(HALT)$;
```

```
$MemBal$(in $k : lvl$) : $bool$
   1. if $xfer = TOP$ or $k = mybot$ then
   2.   return $false$;
   3. $Snd(w-1, GETL, mybot)$;
   4. $SndLvl(w-1, mybot)$;
   5. return $true$;
```

```
$Saturate$(in $k : lvl, p : idx$) : $idx$
  declare $e : event$; $i, j : lcl$; $\mathcal{L}$ : set of $lcl$;
  declare $f, u, r : idx$; $pCng : bool$;
   1. repeat
   2.   $pCng \leftarrow false$;
   3.   for $e \in \mathcal{E}^k$ do
   4.     $\mathcal{L} \leftarrow \{i \in \mathcal{S}_k : \langle k|p \rangle[i] \neq 0 \wedge \exists j, \mathbf{N}_{k,e}[i,j] = 1\}$;
   5.     while $\mathcal{L} \neq \emptyset$ do
   6.       pick and remove $i$ from $\mathcal{L}$;
   7.       repeat
   8.         $f \leftarrow Fire(e, k-1, \langle k|p \rangle[i])$;
   9.         if $f = NOMEM$ then
  10.           if $MemBal(k-1) = false$ then return $NOMEM$;
  11.       until $f \neq NOMEM$;
  12.       if $f \neq 0$ then
  13.         for $j \in \mathcal{N}_{k,e}(i)$ do
  14.           repeat
  15.             $u \leftarrow Union(k-1, f, \langle k|p \rangle[j])$;
  16.             if $u = NOMEM$ then
  17.               if $MemBal(k-1) = false$ then return $NOMEM$;
  18.           until $u \neq NOMEM$;
  19.           if $u \neq \langle k|p \rangle[j]$ then
  20.             $\langle k|p \rangle[j] \leftarrow u$; $pCng \leftarrow true$;
  21.             if $\mathcal{N}_{k,e}(j) \neq \emptyset$ then $\mathcal{L} \leftarrow \mathcal{L} \cup \{j\}$;
  22. until $pCng = false$
  23. return $p$;
```

```
$NewNode$(in $k : lvl$) : $idx$
  declare $p : idx$;
   1. $xfer \leftarrow Decision(mymem, abovemem, belowmem)$;
   2. if $xfer \neq NONE$ then
   3.   if $MemBal(k) = false$ then return $NOMEM$;
   4. $p \leftarrow AllocateNode(k)$;
   5. for $i \in \mathcal{S}_k$ do $\langle k|p \rangle[i] \leftarrow 0$;
   6. return $p$;
```

```
$Union$(in $k : lvl, p : idx, q : idx$) : $idx$
  declare $s, r, u : idx$;
   1. if $p = 0$ then return $q$;
   2. if $q = 0$ then return $p$;
   3. if $p = 1$ or $q = 1$ then return 1;
   4. if $Find(UC[k], p, q, r)$ then return $r$;
   5. if $k < mybot$ then
   6.   $Snd(w-1, UNIN, (k, p, q))$;
   7.   if $Rcv(w-1, s) \neq FAIL$ then
   8.     $CheckIn(k, s)$; $Cache(UC[k], p, q, s)$; return $s$;
   9.   $Snd(w-1, PUTL, mybot-1)$;
  10.   $RcvLvl(w-1, mybot-1)$;
  11. $s \leftarrow NewNode(k)$;
  12. if $s = NOMEM$ then return $NOMEM$;
  13. for $i = 0$ to $|\mathcal{S}^k| - 1$ do
  14.   repeat
  15.     $u \leftarrow Union(k-1, \langle k|p \rangle[i], \langle k|q \rangle[i])$;
  16.     if $u = NOMEM$ then
  17.       if $MemBal(k-1) = false$ then return $NOMEM$;
  18.   until $u \neq NOMEM$;
  19.   $\langle k|s \rangle[i] \leftarrow u$;
  20. $CheckIn(k, s)$; $Cache(UC[k], p, q, s)$; return $s$;
```

```
$Fire$(in $e : event, l : lvl, q : idx$) : $idx$
  declare $i, j : lcl$; $\mathcal{L}$ : set of $lcl$; $f, u, s, r : idx$; $sCng : bool$;
   1. if $l < Bot(e)$ then return $q$;
   2. if $Find(FC[l], q, e, r)$ then return $r$
   3. if $l < mybot$ then
   4.   $Snd(w-1, FIRE, (e, l, q))$;
   5.   if $Rcv(w-1, s) \neq FAIL$ then
   6.     $CheckIn(l, s)$; $Cache(FC[l], q, e, s)$; return $s$;
   7.   $Snd(w-1, PUTL, mybot-1)$;
   8.   $RcvLvl(w-1, mybot-1)$;
   9. $s \leftarrow NewNode(k)$; $sCng \leftarrow false$;
  10. if $s = NOMEM$ then return $NOMEM$;
  11. $\mathcal{L} \leftarrow \{i \in \mathcal{S}_l : \langle k|q \rangle[i] \neq 0 \wedge \exists j, \mathbf{N}_{l,e}[i,j] = 1\}$;
  12. while $\mathcal{L} \neq \emptyset$ do
  13.   pick and remove $i$ from $\mathcal{L}$;
  14.   repeat
  15.     $f \leftarrow Fire(e, l-1, \langle l|q \rangle[i])$;
  16.     if $f = NOMEM$ then
  17.       if $MemBal(l-1) = false$ then return $NOMEM$;
  18.   until $f \neq NOMEM$;
  19.   if $f \neq 0$ then
  20.     for $j \in \mathcal{N}_{l,e}(i)$ do
  21.       repeat
  22.         $u \leftarrow Union(l-1, f, \langle l|s \rangle[j])$;
  23.         if $u = NOMEM$ then
  24.           if $MemBal(l-1) = false$ then return $NOMEM$;
  25.       until $u \neq NOMEM$;
  26.       if $u \neq \langle l|s \rangle[j]$ then
  27.         $\langle l|s \rangle[j] \leftarrow u$; $sCng \leftarrow true$;
  28. if $sCng$ then
  29.   $s \leftarrow Saturate(l, s)$;
  30.   if $s = NOMEM$ then return $NOMEM$;
  31. $CheckIn(l, s)$; $Cache(FC[l], q, e, s)$; return $s$;
```

Figure 2: Distributed version of the saturation algorithm.

Type	From	Params	Meaning
$FIRE$	above	(e,k,p)	request to fire e in $\langle k\|p\rangle$
$UNIN$	above	(k,p,q)	request to union $\langle k\|p\rangle$ and $\langle k\|q\rangle$
$FAIL$	below	none	workstation below is out of memory
$INIT$	below	(r)	node index to initialize the MDD
$RESP$	below	(r)	fire or union response
$GETL$	above	(k)	request receiver to take level k
$PUTL$	above	(k)	request receiver to give level k
$ABRT$	above	none	abort the state-space generation
$HALT$	above	none	termination

Table 1: The message types, source, params, and meaning.

uses a broadcast, but this effect is achieved with pairwise communications.

3.5 Union heuristics

In [4], we devised a *union cache prediction* heuristic, which exploits the fact that, if $Union(k,p,q)$ returns r, then also $Union(k,p,r)$ and $Union(k,q,r)$ must return r, thus these two additional results can be cached immediately after caching that for $Union(k,p,q)$. Experiments indicated that this heuristics can reduce the computation time of SMART by up to 12% in some benchmarks, but it might also waste space by caching results that are never used. In our distributed algorithm, a remote union request is obviously much more expensive than a local recursive function call. Thus, we apply this heuristics, but only for nodes at level $mybot-1$ of each workstation $w>1$.

4 Dynamic memory load balancing

A common problem for distributed reachability analysis approaches is that the memory requirements of each workstation might evolve quite differently. With a static allocation of levels, state-space generation might fail because of excessive memory requirements at a single workstation, even if other workstations still have large amounts of unused memory. To better utilize the resources of each workstation, we apply dynamic memory load balancing (DMLB).

We stress that, unlike for explicit state-space generation where the memory requirements of each workstation can only increase over time, *even an optimal static allocation of levels to workstations could still be inferior to a good, but sub-optimal, dynamic approach*. This is because, usually, the number of nodes at a given MDD level both increases and decreases dramatically during the execution. Workstation w might be using much less memory than workstation w' at some point in time, while the reverse might occur later on. By reallocating levels from w' to w first, and from w to w' later on, we cope with such dynamic peak requirements.

4.1 Single memory threshold dilemma

A simple DMLB scheme could be to predefine a memory usage threshold, say 80% of the amount of memory initially declared to be available by the operating system. Each workstation w keeps track of its own memory usage in a variable, $mymem$, and attaches it to each message it sends. Thus, w always knows the up-to-date memory usage for its two neighbors, $abovemem$ and $belowmem$, since: (i) a workstation can update $mymem$ only when active, (ii) messages are sent from an active workstation before becoming inactive, and (iii) are received by a workstation upon becoming active. If $mymem$ for the active workstation w reaches the threshold, w computes the memory usage that its neighbors would reach if they received a level from w:

$$abovemem' = abovemem + Mem(mytop)$$
$$belowmem' = belowmem + Mem(mybot),$$

where $Mem(k)$ is the memory usage due the unique table at level k and the caches at level $k-1$. Then, if $mymem > abovemem'$ and $belowmem' > abovemem'$, w reallocates its level $mytop$ to $w+1$; if $mymem > belowmem'$ and $abovemem' > belowmem'$, w reallocates its level $mybot$ to $w-1$; otherwise, no reallocation takes place.

Since workstations are organized in a linear fashion and only the active workstation w can trigger DMLB, it might happen that w reallocates some of its levels to $w+1$ and $w-1$, after which all three workstations, $w+1$, w, and $w-1$, are above the threshold, while workstations further away from w might still have much available space. We say that w is *trapped* in this situation.

4.2 Nested DMLB actions

The approach we suggest to prevent workstations from being trapped is to recursively propagate DMLB to all workstations with free space, still using pairwise exchanges. The new rule is that a workstation still triggers DMLB based on a single threshold. However, now, a workstation can trigger a DMLB exchange with one of its neighbors even if it is already in the middle of a DMLB exchange with the other neighbor.

Function $NewNode$ in Fig. 2 achieves this nesting of DMLB actions because it is called not only to allocate a single node in $Union$ or $Fire$, but also to allocate all the nodes of a level received by a workstation through a call to $RcvLvl$. Before actually allocating a node, $NewNode$ calls $Decision$ to determine whether DMLB should be triggered. The outcome, which can be $NONE$, if no DMLB is required, BOT, if level $mybot$ is to be passed to workstation $w-1$, or TOP, if level $mytop$ is to be passed to workstation $w+1$, is recorded in the variable $xfer$, which is globally visible within w. Since $NONE$ is always returned if $mymem$ is less than the threshold, no DMLB overhead is incurred if the decision diagram is small enough to fit comfortably in the memory of each workstation, given the initial allocation. In other words, we do not correct a memory unbalance if even the most loaded workstation is still far from running out of memory.

4.3 Dealing with $NOMEM$

The repeat loops at lines 7 and 14 in $Saturate$, 14 and 21 in $Fire$, and 14 in $Union$ deal with the possibility that a recursive function call to the level immediately below, say k, returns $NOMEM$. If this happens, the code tries to perform DMLB by calling $MemBal$ on level k. We say that level k is *stable* if it is above a level that needs to be reallocated by the DMLB action.

If $MemBal$ returns $false$, it is either because it was determined (by the last call to $NewNode$) that the level to be reallocated is $mytop$ (thus, level k is not above it, hence it is not stable) or because the level to be reallocated is $mybot$ but k equals $mybot$ (thus, the recursion must step back to the stable level $k+1$ before reallocating level $k = mybot$ to workstation $w-1$); the repeat loop is exited by returning a $NOMEM$ value to the level immediately above. If $MemBal$ returns $true$, instead, it is because level $mybot$ was successfully reallocated to workstation $w-1$.

Then, the repeat loop makes the same recursive function call again, since, now, workstation w is using less memory and there is a hope that the call will not return $NOMEM$. The repeat loop allows for the possibility of reallocating multiple levels, one at time, to workstation $w-1$.

When $MemBal$ returns $false$, the code must backtrack to a stable level, either $mytop+1$, if $mytop$ is to reallocated to $w+1$, or $mybot+1$, if $mybot$ is to reallocated to $w-1$ and $k = mybot$. This implies a minimal work loss of one unfinished node per backtracked level, since we do not need to remove nodes that have been checked in the unique table or results that have been cached.

Only two neighboring workstations, $w+1$ and w, or w and $w-1$, are involved in a DMLB. In the first case, level $mybot$ of $w+1$ is stable; in the second case, level $mybot+1$ of w is stable. Once the data for the reallocated level has been transferred, w continues the saturation procedure by calling the last unfinished operation again, which is at the (new) level $mytop$, formerly level $mytop-1$, of w in the first case, or the (new) level $mytop$ of $w-1$, formerly level $mybot$ of w, in the second case.

We stress that the reallocation procedure does not change the structure of the MDD, nor the indices of nodes for in the exchanged level, so it does not alter the MDD canonicity.

5 Experimental results

We now compare SMARTNOW, which we implemented in C using the message-passing library MPICH, with the sequential version of saturation in SMART. The hardware platform is a cluster of Pentium IV 3GHz workstations, each with 512MB of RAM, connected by Gigabit Ethernet. The software environment is Red-Hat Linux 9.0 with a standard MPI2 running on top of TCP/IP. As examples, we use two Petri net models, of a *slotted ring network protocol* [12] and a *round robin mutex protocol* [13], with a parameter N affecting the size of the state space and the number of levels.

Due to memory reserved by the operating system and background jobs, the amount of available RAM differs somewhat for each workstation. In our test cases, it averaged 385MB. Using our memory threshold of 80%, this translates into 308MB on average, before triggering DMLB.

Table 2 reports our experimental results. Column SMART shows the time to run SMART on one workstation. The nine SMARTNOW columns show the time to run on 2, 4, or 8 workstations, using Gigabit Ethernet, possibly configured to run at 100 or 10 Megabit using `ethtool`; the numbers in parentheses report the time spent for DMLB in the Gigabit case. A “*” indicates that SMART, or at least one workstation for SMARTNOW, triggered swapping.

When the RAM of a single workstation is sufficient to run the test case, the runtime of SMART is better than that of SMARTNOW on multiple workstations. Considering first the Gigabit columns, Table 2 shows that the message-passing overhead, while not huge, is not trivial either. In the smallest test cases, the runtime of SMARTNOW with $W = 8$ is sev-

Benchmark			SMART	SMARTNOW								
N	$\lvert\mathcal{S}\rvert$	**MB**	**1 @ 512MB**	**2 @ 512MB**			**4 @ 512MB**			**8 @ 512MB**		
				Gigabit	100M	10M	Gigabit	100M	10M	Gigabit	100M	10M
Slotted ring network model $K = N$, $\lvert\mathcal{S}_k\rvert = 15$ for all k, $\lvert\mathcal{E}\rvert = 3N$												
100	$2.60 \cdot 10^{105}$	36	12	16	16	22	24	25	45	42	43	75
150	$4.53 \cdot 10^{168}$	125	44	50	52	61	64	68	115	91	97	152
200	$8.38 \cdot 10^{211}$	286	108	119	121	137	139	143	193	182	199	287
250	$1.59 \cdot 10^{265}$	545	* 392	(9) 248	274	508	267	298	340	337	371	491
275	$7.04 \cdot 10^{291}$	737	* 57414	(26) 355	486	1098	358	368	441	443	449	621
300	$3.11 \cdot 10^{318}$	962	—	* (29) 552	2663	12457	(23) 490	1981	7333	564	589	763
325	$1.38 \cdot 10^{345}$	1213	—	* (668) $>$ 24hr	$>$ 24hr	$>$ 24hr	(80) 656	9082	26199	716	743	941
350	$6.15 \cdot 10^{371}$	1588	—	—	—	—	* (238) 1148	46910	$>$ 24hr	878	919	2470
Round robin mutex protocol $K = N + 1$, $\lvert\mathcal{S}_k\rvert = 10$ for all k except $\lvert\mathcal{S}_1\rvert = N + 1$, $\lvert\mathcal{E}\rvert = 5N$												
300	$1.37 \cdot 10^{93}$	71	3	4	6	11	7	9	23	13	15	35
450	$2.94 \cdot 10^{138}$	158	8	9	11	19	12	15	38	20	31	63
600	$5.60 \cdot 10^{183}$	280	14	16	20	35	18	25	32	30	37	94
750	$9.99 \cdot 10^{228}$	434	21	24	29	58	29	39	113	41	55	130
900	$1.71 \cdot 10^{274}$	621	* 928	(8) 41	101	558	40	51	152	54	71	302
1050	$2.85 \cdot 10^{319}$	870	* 36751	* (15) 72	501	2713	(5) 56	148	481	68	96	376
1200	$4.64 \cdot 10^{364}$	1201	—	* (23) 368	1169	6610	(45) 91	1542	5410	87	143	472
1350	$4.76 \cdot 10^{409}$	1503	—	—	—	—	* (98) 196	3815	10754	(3) 112	1099	2245

Table 2: Runtime (in seconds) and memory requirements: SMART vs. SMARTNOW.

eral times higher than that of SMART. However, the difference diminishes as the model size grows, even before memory swapping becomes an issue (e.g., for the slotted ring model, the runtime of SMARTNOW is 3.5 times that of SMART when $N = 100$, but not even 1.7 times when $N = 200$).

Once the operating system triggers memory swapping, the runtime for SMART increases substantially. If SMART can be run at all, its runtime can be orders of magnitude larger than that of SMARTNOW. Indeed, such huge differences arise even when comparing SMARTNOW with itself using different values of W. For example, in the round robin model with $N = 900$, SMARTNOW runs much faster than SMART, and the best choices are $W = 2$ and $W = 4$, almost tied at 41 and 40 seconds, respectively; however, for $N = 1050$, $W = 4$ is substantially better than $W = 2$; for $N = 1200$, $W = 4$ and $W = 8$ are enormously better than $W = 2$, and SMART fails to complete; for $N = 1350$, $W = 8$ is by far the best choice, while even SMARTNOW with $W = 2$ fails to complete. Thus, the obvious conclusion is that the optimal number W of workstations to use is that where almost no memory swapping takes place. While such W cannot be known a priori, the results clearly show that using too many workstations affects the runtime only by a small factor, while using too few, or just one, results in very large penalties, if the algorithm completes at all.

Fig. 3 shows how our DMLB scheme allows SMARTNOW to fully utilize the memory of each workstation. In the plot, w_3 runs out of memory first and triggers DMLB; w_4 and w_2 keep receiving MDD levels from w_3 to relieve its peak memory consumption. Eventually, also w_2 hits the memory threshold, and performs DMLB to w_1. Yet, Table 2

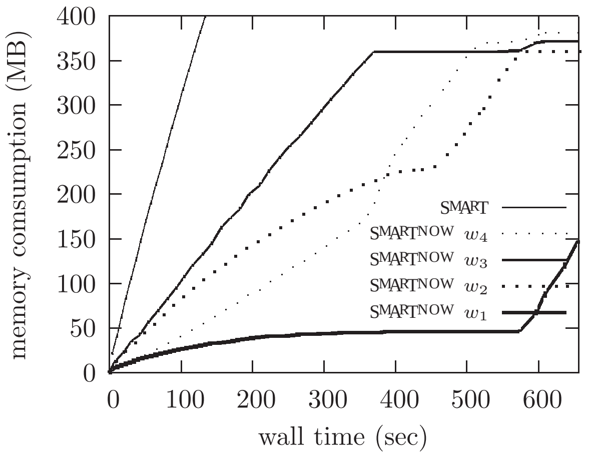

Figure 3: Slotted ring network protocol with 325 slots.

also indicates that the tradeoff in fully utilizing memory resources is that a substantial portion of the runtime is spent on DMLB, 80 out of 656 seconds; some MDD levels are passed between two workstations more than twice. For this model, the time to run SMARTNOW with $W = 4$ is almost as high as with $W = 8$ and, since neither case triggers memory swapping, this is a case where the cost of DMLB with $W = 4$ almost offsets the additional communication costs when $W = 8$.

Finally, a comparison of the Gigabit with the 100 and 10 Megabit columns shows how the speed of the connection becomes a major factor when DMLB is triggered. Clearly, the communication needs of the saturation algorithm itself are much less critical than those of DMLB.

6 Related work

[14, 15] perform *explicit* state-space exploration or model checking over a cluster of workstations and report speedup on some benchmarks but the large memory consumption limits explicit techniques to relatively small models. Recently, *out-of-core* or *on-the-fly* techniques have rendered some gigantic systems tractable by explicit approaches, but the computation time tradeoff is considerable. Even though a symbolic data structure can dramatically shrink the memory requirement for representing sets of states, it still faces the difficulty of parallelizing state-space generation. To avoid the expensive communication cost of synchronizing hash tables, [16, 17] parallelize BDD manipulation on a shared-memory multi-processor (SMP) or distributed shared memory (DSM) by decomposing boolean operations during the expansion phase onto processors within breadth-first or partial-breadth-first behaviors. Yet, scalability of these approaches is still limited.

To further extend the capability of symbolic approaches, [18, 19] suggest to parallelize BDD manipulations by decomposing boolean functions in breath-first (BF) fashion and distributing the workload over a cluster of workstations via message passing. Yet, the tradeoff of parallelizing breadth-first BDD traversals is to overlap the image computation, hurting the scalability of this scheme. To achieve scalability, researchers have recently focused on memory load distribution. [20] has suggested to employ a host processor to manage the job queue for load-balance purposes and to minimize the overlap of image computation by slicing boolean functions with optimal choices of variables. However, no speedup is achieved. Using an alternative distribution of the memory load, [21] partitions the BDD horizontally onto a cluster of workstations, so that hash table synchronization is avoided and the distributed image computation does not create any redundant work at all. The tradeoff in maintaining the canonicity of the distributed data structure is again no speedup.

Compared to *boolean functions (vertical) slicing* approaches, our scheme has several major advantages. With vertical slicing, processors overlap image computation, so synchronization is required to minimize redundant work. If the slicing choice is poor, a substantial number of additional nodes is created, and it is generally agreed that finding a good slicing is not trivial [22]. Similar to [21], our MDDs node allocation partitions instead the nodes of the MDD so that each workstation exclusively owns a contiguous range of MDD levels. Also, our distributed saturation does not perform any duplicate work at runtime and does not have any synchronization overhead. No work queue or central manager is needed in our scheme; in particular, this implies that expensive operations such as global broadcasting or deadlock detection are not needed by our mechanism. The only overhead in our approach arises from peer-to-peer communication. Unlike [21], our implementation scales well and the DMLB scheme utilizes each workstation within the cluster.

7 Conclusion and future work

We presented a distributed version of our symbolic state-space generation algorithm, saturation. Its level-based node allocation scheme does not create additional nodes and, paired with a dynamic memory load balance heuristic and lack of central manager, achieves excellent memory distribution and scalability over a computer cluster. Thanks to the ever increasing network speed, our approach effectively provides the large amounts of memory needed when studying large systems, although it offers no theoretical speedup. We plan to address this aspect in our future research.

References

[1] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[2] R. E. Bryant, “Graph-based algorithms for boolean function manipulation,” *IEEE Trans. Comp.*, vol. 35, pp. 677–691, Aug. 1986.

[3] H. Ochi, Y. Kouichi, and S. Yajima, “Breadth-first manipulation of very large binary-decision diagrams,” in *ICCAD*, pp. 48–55, IEEE Comp. Soc. Press, 1993.

[4] G. Ciardo, G. Luettgen, and R. Siminiceanu, “Saturation: An efficient iteration strategy for symbolic state space generation,” in *TACAS*, LNCS 2031, pp. 328–342, Springer, Apr. 2001.

[5] J.R. Burch, E.M. Clarke, and D.E. Long, “Symbolic model checking with partitioned transition relations,” in *Int. Conference on Very Large Scale Integration*, pp. 49–58, IFIP Transactions, North-Holland, Aug. 1991.

[6] G. Ciardo, R. Marmorstein, and R. Siminiceanu, “Saturation unbound,” in *TACAS*, LNCS 2619, pp. 379–393, Springer, Apr. 2003.

[7] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper, “Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models,” *INFORMS J. Comp.*, vol. 12, no. 3, pp. 203–222, 2000.

[8] G. Ciardo, G. Luettgen, and R. Siminiceanu, “Efficient symbolic state-space construction for asynchronous systems,” in *ICATPN*, LNCS 1825, pp. 103–122, Springer, June 2000.

[9] A. S. Miner and G. Ciardo, “Efficient reachability set generation and storage using decision diagrams,” in *ICATPN*, LNCS 1639, pp. 6–25, Springer, June 1999.

[10] G. Ciardo and R. Siminiceanu, “Using edge-valued decision diagrams for symbolic generation of shortest paths,” in *FMCAD*, LNCS 2517, pp. 256–273, Springer, Nov. 2002.

[11] G. Ciardo and R. Siminiceanu, “Structural symbolic CTL model checking of asynchronous systems,” in *CAV*, LNCS 2725, pp. 40–53, Springer, July 2003.

[12] E. Pastor, O. Roig, J. Cortadella, and R. Badia, “Petri net analysis using boolean manipulation,” in *ICATPN*, LNCS 815,, pp. 416–435, Springer, June 1994.

[13] S. Graf, B. Steffen, and G. Lüttgen, “Compositional minimisation of finite state systems using interface specifications,” *Formal Asp. of Comp.*, vol. 8, no. 5, pp. 607–616, 1996.

[14] D. Nicol and G. Ciardo, “Automated parallelization of discrete state-space generation,” *J. Par. and Distr. Comp.*, vol. 47, pp. 153–167, 1997.

[15] U. Stern and D. L. Dill, “Parallelizing the Murphy verifier,” in *CAV*, LNCS 1254, pp. 256–267, Springer, 1997.

[16] S. Kimura and E. M. Clarke, “A parallel algorithm for constructing binary decision diagrams,” in *ICCD*, pp. 220–223, IEEE Comp. Soc. Press, 1990.

[17] Y. Parasuram, E. Stabler, and S.-K. Chin, “Parallel implementation of BDD algorithms using a distributed shared memory,” in *The Hawaii International Conference on System Sciences*, vol. 1, pp. 16–25, IEEE Comp. Soc. Press, 1994.

[18] T. Stornetta and F. Brewer, “Implementation of an efficient parallel BDD package,” in *DAC*, pp. 641–644, ACM Press, 1996.

[19] K. Milvang-Jensen and A. J. Hu, “BDDNOW: A parallel BDD package,” in *FMCAD*, LNCS 1522, pp. 501–507, Springer, 1998.

[20] O. Grumberg, T. Heyman, and A. Schuster, “A work-efficient distributed algorithm for reachability analysis,” in *CAV*, LNCS 2725, pp. 54–66, Springer, 2003.

[21] R. K. Ranjan, J. V. Snaghavi, R. K. Brayton, and A. Sangiovanni-Vincentelli, “Binary decision diagrams on network of workstations),” in *Proc. ICCD*, pp. 358–364, IEEE Comp. Soc. Press, Oct. 1996.

[22] A. Narayan, A. J. Isles, J. Jain, R. K. Brayton, and A. Sangiovanni-Vincentelli, “Reachability analysis using Partitioned-ROBDDs,” in *ICCAD*, pp. 388–393, ACM and IEEE Comp. Soc. Press, 1997.