DISTRIBUTED SATURATION

Ming-Ying Chung, * Gianfranco Ciardo, † and Radu I. Siminiceanu ‡ §

ABSTRACT

The *Saturation* algorithm for symbolic state-space generation, has been a recent breakthrough in the exhaustive verification of complex systems, in particular globally-asynchronous/locally-synchronous systems. The algorithm uses a very compact Multiway Decision Diagram (MDD) encoding for states and the fastest symbolic exploration algorithm to date. The distributed version of Saturation uses the overall memory available on a network of workstations (NOW) to efficiently spread the memory load during the highly irregular exploration. A crucial factor in limiting the memory consumption during the symbolic state-space generation is the ability to perform *garbage collection* to free up the memory occupied by dead nodes. However, garbage collection over a NOW requires a nontrivial communication overhead. In addition, operation cache policies become critical while analyzing large-scale systems using the symbolic approach. In this technical report, we develop a garbage collection scheme and several operation cache policies to help on solving extremely complex systems. Experiments show that our schemes improve the performance of the original distributed implementation, SMART-NOW, in terms of time and memory efficiency.

1 INTRODUCTION

Formal verification techniques such as model checking and theorem proving have become widely used in industry for quality assurance, as they can be used to detect errors early in the design lifecycle. State-space generation, also called reachability analysis, is an essential but very memory-intensive step in model checking. The increasing complexity of system designs stresses the limits of most model checkers. Even though symbolic state-space encodings based on *binary decision diagrams* (BDDs) [2] and *multiway decision diagrams* (MDDs) [20] help cope with the inherent state-space explosion of discrete-state systems, the analysis of some industrial size models may still rely on the use of virtual memory. Our discussion regarding state-space generation focuses on symbolic state-space generation.

A natural way to deal with the excessive memory consumption of reachability analysis is using parallel and distributed approaches. Most of the research in this area has been focused on *vertical* slicing schemes to parallelize BDD manipulations, by decomposing boolean functions lines and distributing the computation over a NOW [18, 21, 25]. This scheme allows algorithms to overlap the *image computation* (the application of the next-state function to a set of states encoded by a decision diagram node), but the distributed state-space generation is still synchronous, consisting of interleaved rounds of computation and communication, in which the fastest or the most lightly loaded workstation must wait for the heavily loaded

*University of California, Riverside, CA 92521. Email: chung@cs.ucr.edu

†University of California, Riverside, CA 92521. Email: ciardo@cs.ucr.edu

‡National Institute of Aerospace, 100 Exploration Way, Hampton VA, 23666. E-mail: radu@nianet.org

§This work was supported in part by the National Aeronautics and Space Administration under the cooperative agreement NCC-1-02043

ones at the end of each round. Thus, the global synchronization required at the end of each round is detrimental to this scheme in terms of scalability. To overcome this drawback, [17] introduced an asynchronous version of the vertical slicing approach which not only performs image computation and message passing concurrently, but also incorporates an adaptive mechanism taking into account the availability of free computational power to split workload.

In [5], we instead use MDDs and partition them *horizontally* onto a NOW, so that each workstation exclusively owns a contiguous range of MDD levels. Therefore, the memory required for state-space encoding is mutually exclusively partitioned onto workstations. Since the horizontally distributed state-space generation does not create any redundant or duplicate work at all, synchronization is avoided. Furthermore, within the horizontal slicing scheme, only peer-to-peer communication between neighboring workstations is used, so scalability is not an issue. However, this approach comes with a severe tradeoff. Given the highly optimized nature of saturation, which was designed as a sequential algorithm, only one workstation is active at anytime, hence the distributed computation is virtually sequentialized. This leaves only limited opportunities for speedup. To tackle this drawback, in [6, 7], we introduced an idea to speedup distributed state-space generation by using workstations' idle time to speculatively perform image computations.

Also, during distributed state-space generation, performing *garbage collection* for dead MDD nodes over a NOW requires a nontrivial communication overhead. In addition, MDD cache policies become relatively critical in a large-scale symbolic reachability analysis. Thus, in this paper, we develop a garbage collection scheme and several operation cache policies to help on solving extremely complex systems. The paper is organized as follows. Sect. 2 gives the necessary background on state-space generation, decision diagrams, Kronecker encoding, and the evolution of saturation algorithm. Sect. 3 details our new garbage collection scheme tailor-made for distributed state-space generation. Sect. 4 discusses several operation cache policies which might help on solving complex systems. Sect. 5 shows experimental results. Sect. 6 draws conclusions and discusses future research directions.

2 BACKGROUND

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 $\mathcal{N} : \widehat{\mathcal{S}} \to 2^{\widehat{\mathcal{S}}}$ is the *next-state function* specifying the states reachable from each state in a single step. We assume that the model is composed of K *submodels*. Thus, a (*global*) state $\mathbf{i}$ is a K-tuple $(\mathbf{i}_K, ..., \mathbf{i}_1)$, where $\mathbf{i}_k$ is the *local* state of submodel k, $K \geq k \geq 1$, and $\widehat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$ is the cross-product of K *local state spaces*. This allows us to use techniques targeted at exploiting system structure, in particular, symbolic techniques to store the state space based on decision diagrams.

Since we target globally-asynchronous locally-synchronous systems, we decompose $\mathcal{N}$ into a disjunction of next-state functions [4]: $\mathcal{N}(\mathbf{i}) = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e(\mathbf{i})$, where $\mathcal{E}$ is a finite set of *events* and $\mathcal{N}_e$ is the next-state function associated with event e. We then seek to build the *(reachable) state space* $\mathcal{S} \subseteq \widehat{\mathcal{S}}$, 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 reflexive and transitive closure and $\mathcal{N}(\mathcal{X}) = \bigcup_{\mathbf{i} \in \mathcal{X}} \mathcal{N}(\mathbf{i})$.

2.1 Symbolic encoding of $\mathcal{S}$

In the sequel, we assume that each $\mathcal{S}_k$ is known a priori. In practice, the local state spaces $\mathcal{S}_k$ can actually be generated "on-the-fly" by interleaving symbolic global state-space generation with explicit local state-space generation [12]. We then use the mappings $\psi_k : \mathcal{S}_k \to \{0, 1, \ldots, n_k-1\}$, with $n_k = |\mathcal{S}_k|$, identify local state $\mathbf{i}_k$ with its index $i_k = \psi_k(\mathbf{i}_k)$, thus $\mathcal{S}_k$ with $\{0, 1, \ldots, n_k-1\}$, and encode any set $\mathcal{X} \subseteq \widehat{\mathcal{S}}$ in a *(quasi-reduced ordered) MDD* over $\widehat{\mathcal{S}}$. Formally, an MDD is a directed acyclic edge-labeled multi-graph where:

- Each node p belongs to a *level* $k \in \{K, ..., 1, 0\}$, denoted $p.lvl$.
- There is a single *root* node r at level K.
- Level 0 can only contain the two *terminal* nodes *Zero* and *One*.
- A node p at level $k > 0$ has n_k outgoing edges, labeled from 0 to n_k-1. The edge labeled by i_k points to a node q at level $k-1$; we write $p[i_k] = q$.
- Given nodes p and q at level k, if $p[i_k] = q[i_k]$ for all $i_k \in \mathcal{S}_k$, then $p = q$, i.e., there are no *duplicates*.

The MDD encodes a set of states $\mathcal{B}(r)$, defined by the recursive formula:

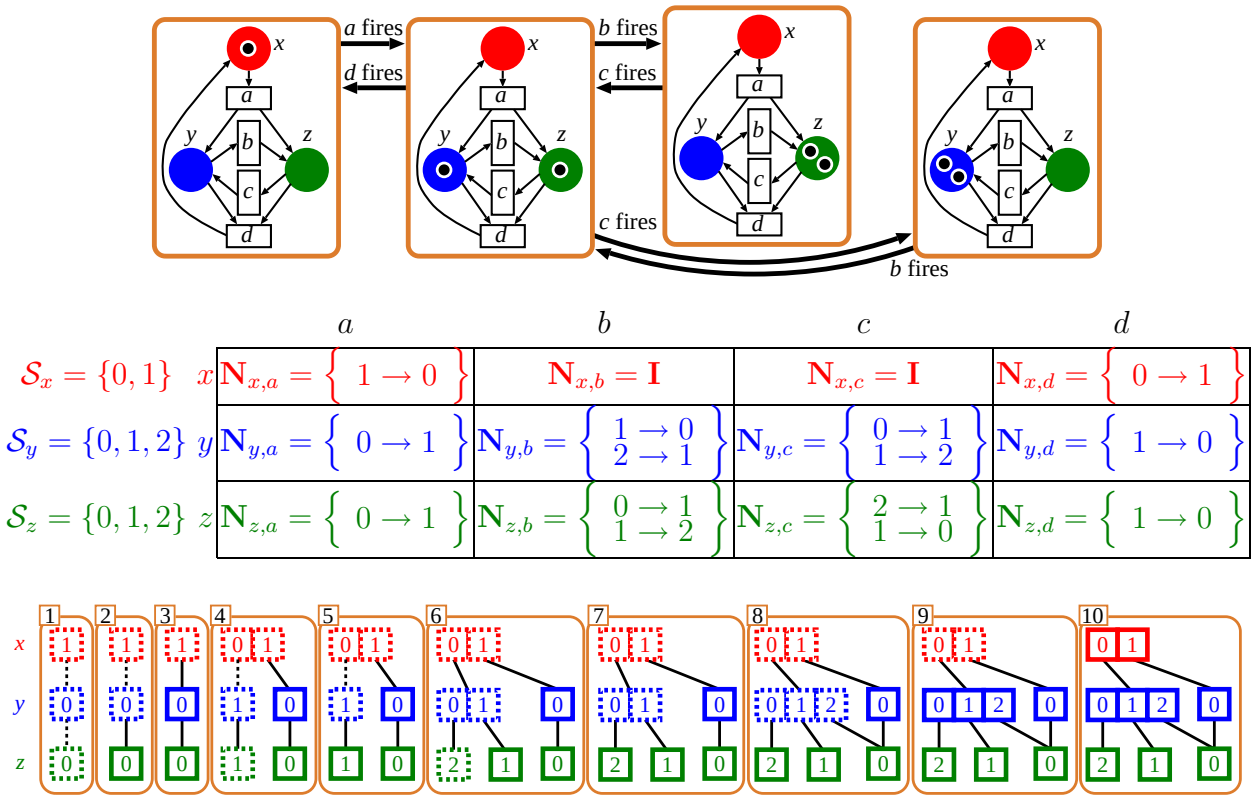
$$\mathcal{B}(p) \;=\; \begin{cases} \bigcup_{i_k \in \mathcal{S}_k} \{i_k\} \times \mathcal{B}(p[i_k]) & \text{if } \; p.lvl = k > 1 \\ \{i_1 : p[i_1] = \mathit{One}\} & \text{if } \; p.lvl = 1 \end{cases} .$$

For example, box 10 at the bottom of Fig. 1 shows a five-node MDD with $K = 3$ encoding four global states: (0,0,2), (0,1,1), (0,2,0), and (1,0,0). In our MDDs, arcs point down and their label is written in a box in the node from where the arc originates; the terminal nodes *Zero* and *One* and nodes p such that $\mathcal{B}(p) = \emptyset$, as well as any arc pointing to them, are omitted.

Compared with BDDs, MDDs have the disadvantage of resulting in larger and less shareable nodes when the variable domains $\mathcal{S}_k$ are very large. On the other hand, MDDs have several advantages. First, many real-world models (e.g., non-safe Petri nets and software protocols) have variable domains with a priori unknown or very large upper bounds. These bounds must then be discovered "on the fly" during the symbolic iterations [12, 14], and MDDs are preferable to BDDs when using this approach, due to the ease with which MDD nodes and variable domains can be extended. A second advantage, related to the present paper, is that our chaining heuristics applied to the MDD state variables more closely reflect structural information of the model behavior, which is instead spread on multiple levels in a BDD.

2.2 Symbolic encoding of $\mathcal{N}$

For $\mathcal{N}$, we adopt a Kronecker representation inspired by work on Markov chains [3], possible if the model is *Kronecker consistent* [10, 11]. Each $\mathcal{N}_e$ is conjunctively decomposed into K local next-state functions $\mathcal{N}_{k,e}$, for $K \geq k \geq 1$, satisfying, in any global state $(i_K, \ldots, i_1) \in \widehat{\mathcal{S}}$, $\mathcal{N}_e(i_K, \ldots, i_1) = \mathcal{N}_{K,e}(i_K) \times \cdots \times \mathcal{N}_{1,e}(i_1)$. Using $K \cdot |\mathcal{E}|$ matrices $\mathbf{N}_{k,e} \in \{0,1\}^{n_k \times n_k}$, with $\mathbf{N}_{k,e}[i_k, j_k] = 1 \Leftrightarrow j_k \in \mathcal{N}_{k,e}(i_k)$, we encode $\mathcal{N}_e$ as a (boolean) Kronecker product: $\mathbf{j} \in \mathcal{N}_e(\mathbf{i}) \Leftrightarrow \bigotimes_{K \geq k \geq 1} \mathbf{N}_{k,e}[i_k, j_k] = 1$, where a state $\mathbf{i}$ is interpreted as a *mixed-based* index



	a	b	c	d
$\mathcal{S}_x = \{0,1\}$ x	$\mathbf{N}_{x,a} = \left\{ 1 \to 0 \right\}$	$\mathbf{N}_{x,b} = \mathbf{I}$	$\mathbf{N}_{x,c} = \mathbf{I}$	$\mathbf{N}_{x,d} = \left\{ 0 \to 1 \right\}$
$\mathcal{S}_y = \{0,1,2\}$ y	$\mathbf{N}_{y,a} = \left\{ 0 \to 1 \right\}$	$\mathbf{N}_{y,b} = \left\{ \begin{smallmatrix} 1 \to 0 \\ 2 \to 1 \end{smallmatrix} \right\}$	$\mathbf{N}_{y,c} = \left\{ \begin{smallmatrix} 0 \to 1 \\ 1 \to 2 \end{smallmatrix} \right\}$	$\mathbf{N}_{y,d} = \left\{ 1 \to 0 \right\}$
$\mathcal{S}_z = \{0,1,2\}$ z	$\mathbf{N}_{z,a} = \left\{ 0 \to 1 \right\}$	$\mathbf{N}_{z,b} = \left\{ \begin{smallmatrix} 0 \to 1 \\ 1 \to 2 \end{smallmatrix} \right\}$	$\mathbf{N}_{z,c} = \left\{ \begin{smallmatrix} 2 \to 1 \\ 1 \to 0 \end{smallmatrix} \right\}$	$\mathbf{N}_{z,d} = \left\{ 1 \to 0 \right\}$

Figure 1: Reachability graph, $\mathcal{S}_x, \mathcal{S}_y, \mathcal{S}_z$, and $\mathcal{N}$, evolution of the MDD,

in $\widehat{\mathcal{S}}$ and $\bigotimes$ indicates the Kronecker product of matrices. The $\mathbf{N}_{k,e}$ matrices are extremely sparse, for standard Petri nets, each row contains at most one nonzero entry.

For example, the middle of Fig. 1 shows the Kronecker encoding of $\mathcal{N}$ according to events (a, b, c, d) and levels (x, y, z), listing only the nonzero entries, e.g.,

$$\mathbf{N}_{y,b} = \left\{ \begin{smallmatrix} 1 \to 0 \\ 2 \to 1 \end{smallmatrix} \right\} \quad \text{means} \quad \mathbf{N}_{y,b} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

and $\mathbf{N}_{y,b}[1,0] = 1$ indicates that if the local state at level y is 1, event b is locally enabled and firing b, if globally possible, moves the local state from 1 to 0.

2.3 Saturation-based iteration strategy

In addition to efficiently representing $\mathcal{N}$, the Kronecker encoding allows us to recognize *event locality* [10, 22] and employ *saturation* algorithm [11]. 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 for which $\mathbf{N}_{k,e} \neq \mathbf{I}$. An MDD node p at level k is said to be *saturated* if it is a fixed point with respect to all $\mathcal{N}_e$ such that $Top(e) \leq k$, i.e., $\mathcal{S}_K \times \cdots \times \mathcal{S}_{k+1} \times \mathcal{B}(p) \supseteq \mathcal{N}_{\leq k}(\mathcal{S}_K \times \cdots \times \mathcal{S}_{k+1} \times \mathcal{B}(p))$, where $\mathcal{N}_{\leq k} = \bigcup_{e:Top(e)\leq k} \mathcal{N}_e$. To saturate MDD node p once all its descendants have been saturated, we *update it in place* so that it encodes also any state in $\mathcal{N}_{k,e} \times \cdots \times \mathcal{N}_{1,e}(\mathcal{B}(p))$, for any event e such that $Top(e) = k$. This can create new MDD nodes at levels below k, which are saturated immediately, prior to completing the saturation of p.

If we start with the MDD encoding the initial state $\mathbf{s}^{init}$ and saturate its nodes bottom up, the root r will encode $\mathcal{S} = \mathcal{N}^*(\mathbf{s}^{init})$ at the end, because: (1) $\mathcal{N}^*(\mathbf{s}^{init}) \supseteq \mathcal{B}(r) \supseteq \{\mathbf{s}^{init}\}$, since we only add states, and only through legal event firings, and (2) $\mathcal{B}(r) \supseteq \mathcal{N}_{\leq K}(\mathcal{B}(r)) = \mathcal{N}(\mathcal{B}(r))$, since r is saturated.

The reachability graph of a three-place Petri net is shown at the top of Fig. 1. A global state is described by the local state of place x, y, and z, in that order, and we index local states by the number of tokens in the corresponding place. Three global states, (0,1,1), (0,0,2), and (0,2,0), are reachable from the initial state (1,0,0). The three local state spaces and the Kronecker description of $\mathcal{N}$ are shown in the middle of Fig. 1. The list of nonzero entries for matrix $\mathbf{N}_{y,b}$, for example, indicates that firing event b decreases the number of tokens in place y, either from 2 to 1 or from 1 to 0; it also indicates that b is disabled when place y contains 0 tokens, as no transition is listed from local state 0. The saturation-based state-space generation of this model is shown at the bottom of Fig. 1, where solid MDD nodes are saturated and dashed MDD nodes are not.

1 **Initial configuration :** Set up the MDD encoding the initial global state (1,0,0).

2 **Saturate node** $\boxed{0}$ **at level** z **:** No action is required, since there is no event with $Top(event) = z$. The node is saturated by definition.

3 **Saturate node** $\boxed{0}$ **at level** y **:** $Top(b) = Top(c) = y$, but neither b nor c are enabled at both levels y and z, Thus, no firing is possible, and the node is saturated.

4 **Saturate node** $\boxed{1}$ **at level** x **:** $Top(a) = x$ and a is enabled for all levels, thus event a must be fired on the node. Since, by firing event a, local state 1 is reachable from 0 for both levels y and z, node $\boxed{1}$ at level y and node $\boxed{1}$ at level z, are created (not yet saturated), This also implies that a new global state, (0,1,1), is discovered.

5 **Saturate node** $\boxed{1}$ **at level** z **:** Again, no action is required as the node is saturated by definition.

6 **Saturate node** $\boxed{1}$ **at level** y **:** $Top(b) = y$ and b is enabled for all levels, thus event b must be fired on the node. Since, by firing event b, local state 0 is reached from 1 at level y and local state 2 is reached from 1 at level z, node $\boxed{1}$ at level y is extended to $\boxed{01}$ and node $\boxed{2}$ at level z is created. This also implies that a new global state, (0,0,2), is discovered.

7 **Saturate node** $\boxed{2}$ **at level** z **:** Again, no action is required, as the node is saturated by definition.

8 **Saturate node** $\boxed{01}$ **at level** y **:** $Top(c) = y$ and c is enabled for all levels, thus event c must be fired on the node. Since, by firing event c, local state 2 is reachable from 1 at level y and local state 0 is reachable from 1 at level z, node $\boxed{01}$ at level y is extended to $\boxed{012}$ and node $\boxed{0}$ at level z, which has been created and saturated previously, is referenced. This also implies that a new global state, (0,2,0), is discovered.

9 **Saturate node** $\boxed{012}$ **at level** y **:** After exploring all possible firings, the node is saturated.

10 **Saturate node** $\boxed{01}$ **at level** x **:** Since no firing can find new global states, the root is saturated.

Saturation consists of many "lightweight" nested "local" fixed-point image computations and is completely different from the traditional breadth-first approach that employs a single "heavyweight" global fixed-point image computation. No matter whether the chaining idea is applied or not, results in [11, 12, 13, 14, 8] consistently show that the saturation approach outperforms the breadth-first approach of symbolic state-space generation by several orders of magnitude in both memory and time, making it arguably the most efficient state-space 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 approach would not offset the enormous speedups and memory reductions of saturation approach.

2.4 Saturation NOW

[5] described a message-passing algorithm, *Saturation NOW*, that distributes the MDD nodes encoding states over a NOW, to study large models where a single workstation would have to rely on virtual memory to explore the state space. On a NOW with $W \leq K$ workstations 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 $w = W$). Initially, we evenly allocate the K MDD levels to the W workstations accordingly, by assigning the ownership of levels $\lfloor w \cdot K/W \rfloor$ through $\lfloor (w-1) \cdot K/W \rfloor + 1$ to workstation w. Local variables $mytop_w$ and $mybot_w$ indicate the highest- and lowest-numbered levels owned by workstation w, respectively.

For distributed state-space generation, each workstation w first generates the Kronecker matrices $\mathbf{N}_{k,e}$ for those events and levels where $\mathbf{N}_{k,e} \neq \mathbf{I}$ and $mytop_w \geq k \geq mybot_w$, without any synchronization. Then, the sequential saturation algorithm begins, except that, when workstation $w > 1$ would normally issue a recursive call to level $mybot_w - 1$, it must instead send a request to perform this operation in workstation $w-1$ and wait for a reply. A linear organization of the workstations suffices, since each workstation only needs to communicate with its neighbors.

3 DISTRIBUTED GARBAGE COLLECTION

The implementation of garbage collection in SMART follows the cleanup procedure based on reference counts where each decision diagram node has a counter to record the number of references: the number of the incoming arcs to the node. A node's reference counter decreases by one whenever one of its parents node dereferences it. Whenever the reference counter of a node is becomes zero, any following cleanup invocation will delete the node and remove it from the corresponding unique table. Then, a garbage collection call will recycle the free space using level-based recycling pools. If the strict policy of garbage collection is used in SMART, any dereference may proceed recursively down to the bottom level of decision diagrams. Although a deeper recursive dereference can contribute to freeing up more memory, it can be much costly in terms of runtime as well. So, to relax this policy, SMART allows users to forbid the garbage collection to be triggered until the number of dead nodes reaches some given threshold.

However, updating the reference counters within the decision diagram during distributed state-space generation over a NOW requires a non-trivial amount of messages passing among workstations. We therefore prefer to skip the costly bookkeeping effort on maintaining up-to-date reference counters, in order to avoid the communication overhead. Yet, when the

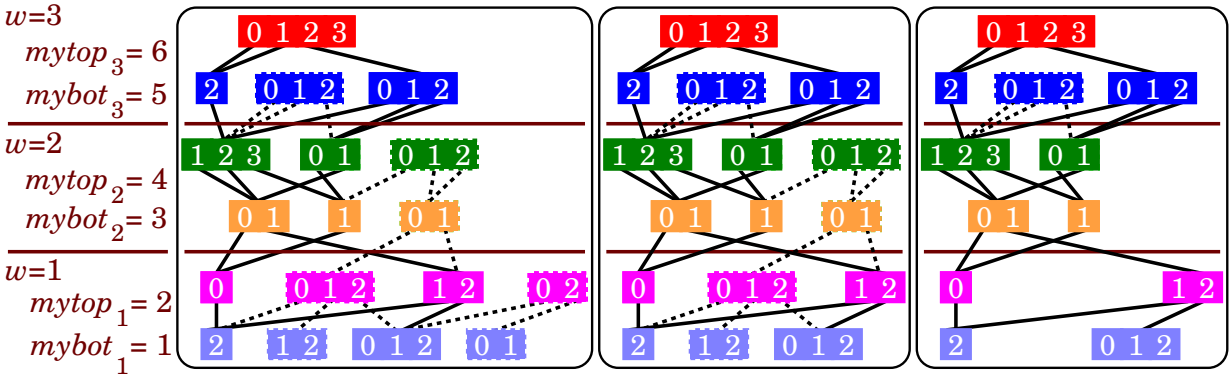


Figure 2: Garbage collection example

overall memory consumption reaches some given threshold, we perform distributed cleanup by freezing state-space exploration temporarily to deal with the disconnected nodes at runtime. Without the up-to-date reference counters, the distributed cleanup issued at the kth level needs to perform a scan at the previous level, reading through all k+1th nodes' outgoing arcs, to determine the referencing information of the kth nodes. To overlap the distributed cleanup and referencing information retrieval, it makes sense to clean up several consecutive levels at a time in a top-down fashion. Thus, our distributed garbage collection triggers a series of distributed cleanups starting at the kth level to recycle disconnected nodes at any level equal to or lower than the k^{th}. In general, the higher level the distributed cleanup is invoked at, the more communication overhead may be introduced.

The left of Fig. 2 shows a runtime snapshot of a six-level decision diagram distributed over three workstations where each workstation manages two levels of the MDD. The dashed boxes and lines indicate the disconnected nodes. The middle of Fig. 2 shows the decision diagram resulting when the two bottom workstations ($w = 2$ and 1) perform distributed cleanup on the decision diagram shown in the left of Fig. 2 starting at level 2. In this case, one node at level 2 and one node at level 1 have been cleaned out. The right of Fig. 2 shows the decision diagrams resulting when three workstations perform distributed cleanup on the same decision diagram but starting at level 4 instead. In this case, four nodes have been removed.

4 OPERATION CACHE POLICIES

The main reason why symbolic model checking can outperform the explicit approach is that the *implicit* state-space construction allows nodes to share not only their children roots of isomorphic decision diagrams (memory efficient) but also the computation corresponding to each of those children (time efficient). To efficiently share the computation during symbolic state-space generation, *hashing* is considered to be one of the most effective way to cache computed data dynamically. However, a good hashing can still create identical hash values for distinct entries. To cope with this so-called *hash-collision* issue, we use collision-tolerating hash tables: multiple entries having identical hash value are stored together using a linked list.

In detail, in each collision-tolerating hash table, we use a singly linked list to store each set of entries having identical hash value. For each level of decision diagrams, we use such

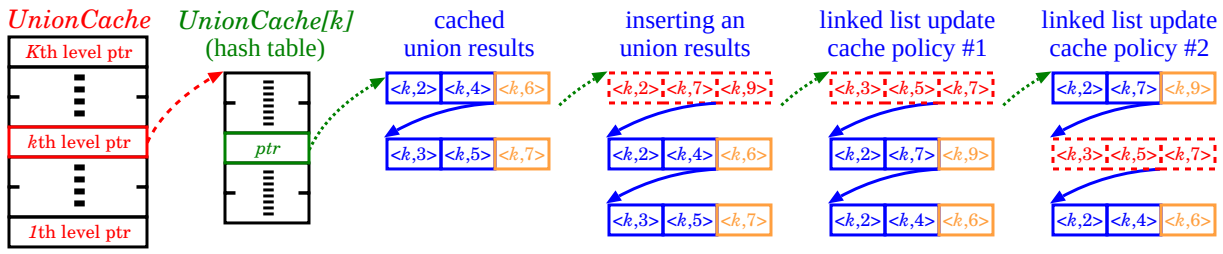


Figure 3: Union caches

a collision-tolerating hash table for each kind of operation (union, fire) and rehash each of the tables whenever the projected number of collisions becomes too large. Fig. 3 shows the data structure that we use to cache the results of union operation during distributed state-space generation. The fourth column of Fig. 3 shows an example that caching a newly computed result, the union of $\langle k, 2\rangle\langle k, 7\rangle \rightarrow \langle k, 9\rangle$, having same hash value as the the union of $\langle k, 2\rangle\langle k, 4\rangle$ and the union of $\langle k, 3\rangle\langle k, 5\rangle$. In this case, three entries are stored in the same linked list.

Yet, during symbolic state-space generation, the collision toleration described previously may hold up any cache-related operation and slowdown the overall computation, because any hash table lookup might end up searching in a linked list. An alternative way to ease the hash-collision issue is to perform rehashing: enlarging each hash table as needed to decrease the chance of hash collision. Yet, excessive rehashing can be very memory consuming as well. Thus, we tend to use a hybrid solution in our application.

4.1 Bounded-rehashing and bounded-collision caches

We develop a *bounded-rehashing* and *bounded-collision* cache policy to facilitate all cache-related operations.

In detail, for each collision-tolerating hash table, we keep track of *MaxCollision*, the maximal size of linked lists used in the table, indicating that the time to retrieve any entry (might end up searching in a linked list) is $\mathcal{O}(MaxCollision)$. Whenever some hash table's *MaxCollision* exceed the given threshold, *MaxAllowedCollision*, we rehash the table by doubling the size of the table. To prevent overusing memory for caching computed results, we prohibit rehashing a table if the size of the table has exceeded another given threshold, *RehashThreshold*. To preserve that the data retrieval time of the caches is asymptotically bounded, we restrict $MaxCollision \leq MaxAllowedCollision$: the size of linked lists used in any hash table is limited to *MaxAllowedCollision*. In other words, the maximal number of entries having identical hash value is always *MaxAllowedCollision* no matter whether the size of the hash table has exceeded *RehashThreshold* or not. So, the time to retrieve a cached entry is also $\mathcal{O}(MaxAllowedCollision)$.

After all, to bound the size of the linked lists once they are full, we always update linked lists by inserting the new element at the front and removing the last one. Also, the maximal values of *MaxAllowedCollision* and *RehashThreshold* allowed in a distributed program can be much larger than those used in a sequential program.

4.2 Policies of single-level caches

Since the number of elements allowed to be stored in each linked list is limited, deciding how to sift out the less useful element is an issue. An entry is said to be a *cache-hit* if it is retrieved by some hash table lookup. To take the usefulness of entries into account, we need some cache policy to preserve those frequently cache-hit entry which means to keep them in the first *MaxAllowedCollision* elements of the corresponding linked list.

LRU, a frequently used policy, treats any newly cache-hit entry the same as a newly cached one: moving every newly retrieved element to the front of the corresponding linked list. Yet, this policy treats every cache-hit entry equally no matter how often each of them has been retrieved. So, another idea is to switch the newly cache-hit entry with its previous one. The last two columns of Fig. 3 show how a linked list will be updated following this two policies after the result of the union of $\langle k, 3\rangle\langle k, 5\rangle \rightarrow \langle k, 7\rangle$ is cache-hit. After all, while using either of these two policies, the time to retrieve each cached entry is still $\mathcal{O}(MaxAllowedCollision)$.

5 RESULTS

5.1 Experiments of bounded rehashing scheme

We evaluated the rehashing heuristic by using the saturation algorithm to generate the state space of the following parameterized models.

- *Round robin mutex protocol* (**Robin**) [16] models the round robin solution of a mutual exclusion problem where N is the number of processes involved.
- *Flexible manufacturing system* (**FMS**) [22] models a manufacturing system with three machines to process three different types of parts where N is the number of each type of parts.
- *Slotted ring network protocol* (**Slot**) [23] models a local area network protocol where N is the number of nodes in the network.
- *Leader election protocol* (**Leader**) [19] models a protocol for designating a unique processor as the leader by sending messages along a unidirectional ring of N processors.
- *Aloha network protocol* (**Aloha**) [9] models a local area network protocol where N is the number of nodes in the network.
- *Kanban manufacturing system* (**Kanban**) [26] models a manufacturing system authorizing production based on the consumption at the downstream stations where N is the admission threshold to each machine.
- *Bounded open queuing network* (**BQ**) [15] models an open queuing network where the capacity of the queue is bounded by N.
- *Knights problem* (**Knight**) models the problem of determining how many non-attacking knights can be placed on an $N \times N$ chessboard.

Model	N	Cache		Time (sec)			Memory (mgb)			Rehash
		init	*max*	*HashI*	*HashM*	*DHash*	*HashI*	*HashM*	*DHash*	(times)
Robin	600	100	50K	41	34	37	326	555	349	2677
FMS	250	100	100K	174	55	56	103	240	108	62
Slot	150	100	100K	122	97	104	210	324	233	1302
Leader	7	100	100K	123	12	15	147	205	172	684
Aloha	70	100	500K	85	12	16	227	502	255	699
Kanban	60	100	1M	124	16	16	60	175	69	143
BQ	40	100	1M	102	58	60	97	120	106	43
Knight	6	100	1M	160	9	12	69	336	123	370
Queen	12	1000	1M	22	10	11	65	149	72	35
RIPS	14444	1000	10M	567	113	127	168	854	516	124

Table 1: Experimental results of rehashing.

- *Queens puzzle* (**Queen**) models the game of placing 8 queens on an $N \times N$ chessboard so that none of them can hit any other in one move.
- *Runway safety monitor* (**RIPS**) [24] models an avionics system monitoring T targets with S speeds on a grid represented as a $X \times Y \times Z$ grid.

Table 1 shows the experimental study performed on a Pentium IV 3GHz workstations with 1GB of RAM. The first two columns show the model names and the corresponding parameters used for this evaluation. The third and fourth columns indicate the initial size (*init*) of hash tables and the maximal size (*max*) allowed for rehashing. The next six columns present the time and memory consumption of three different approaches: fixed-size hashing using *init*, fixed-size hashing using *max*, dynamic hashing using *init* and then allowing to rehash until *max* is researched, denoted as *HashI*, *HashM*, and *DHash* respectively. The last column shows the number of rehashing has been performed while experimenting *DHash*.

In Table 1, we can see that *HashI* is always the most memory efficient approach and *HashM* is always the most time efficient one. However, in all cases, the memory consumption of *DHash* can be as low as *HashI*'s, while the runtime of *DHash* being very close to *HashM*'s Note that, the tradeoff in performing dynamical rehashing (shown in the last column of Table 1) is the runtime difference between *HashM* and *DHash* which is insignificant. Additionally, the high memory consumption of *DHash* implies that excessive unbounded rehashing can be expensive and unrealistic. In conclusion, the experiment shows that rehashing works well in many cases making the bounded rehashing idea more practical.

5.2 Experiments for the message-passing implementation

This experimental study of SMART and SMART$^{\text{MW}}$ on **Robin** and **Slot** was performed on Sciclone [1] cluster at the College of William and Mary consisting many different heterogeneous subclusters. We used the Whirlwind (homogeneous) subcluster which consists of 64 single-cpu Sun Fire V120 nodes (UltraSPARC IIi+ 650 MHz, 1 GB RAM) connecting by Myrinet and running Solaris 9 with LAM/MPI on TCP/IP. Three parameters selected for both models represents the small, medium, and large cases of symbolic state-space generation, where the sequential program requires $\approx$ 100MB, $\approx$ 500MB and $\approx$ 900MB to accomplish the tasks.

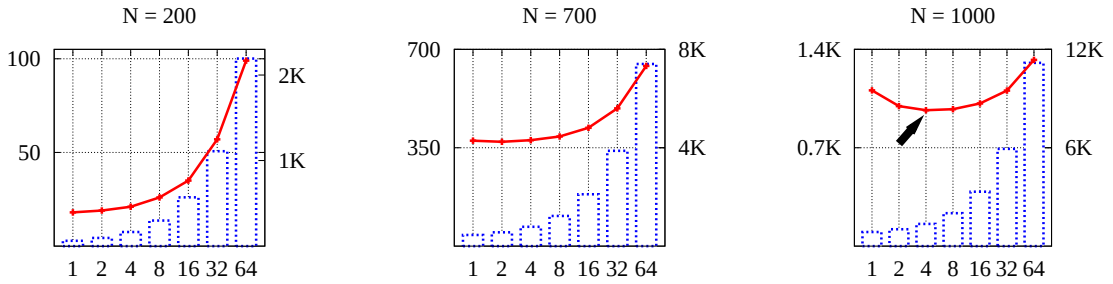


Figure 4: Experiments on **Robin**

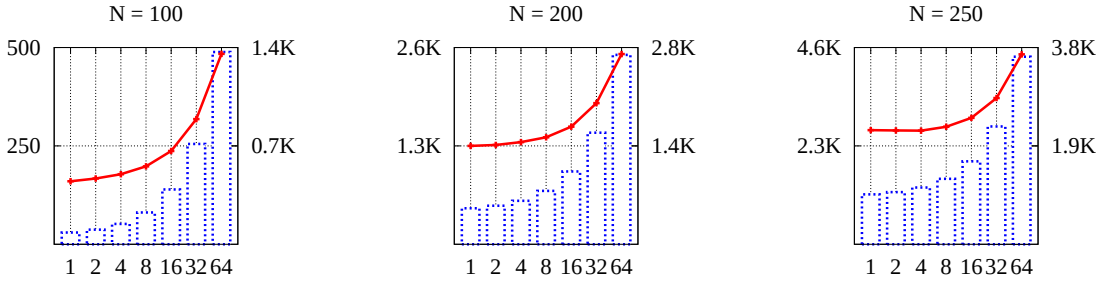


Figure 5: Experiments on **Slot**

For each test case, we run SMARTNOW on 1, 2, 4, 8, 16, 32, and 64 workstation(s) and record the runtime (the red pointed lines corresponding to the left axes) in seconds and the memory consumption (blue dashed boxes corresponding to the right axes) in megabytes. To see the actual overhead of the distributed approach, we use the memory information reported by the operating system. Also, we use fixed-size hashing to test all cases, even though, in comparison to SMART, running SMARTNOW on a NOW will have more memory for rehashing to accelerate the computation

Fig. 4 and 5 show that, 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. The message-passing overhead, while not huge, is not trivial either. In the small test cases, the runtime of SMARTNOW is few times larger than that of SMART. However, the difference diminishes as the model size grows, even way before memory swapping becomes an issue. Indeed, such huge differences arise even when comparing SMARTNOW with itself using different values of W. Considering the case of **ROBIN** $N = 1000$, the optimal number W_{opt} of workstations to use is 4 (indicated by a black arrow in the right of Fig. 4), where SMART rarely trigger memory swapping (862MB of local memory was used) and SMARTNOW does not use virtual memory at all (1026MB of NOW memory was used). In addition, while such W_{opt} 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. Considering the case of **ROBIN** $N = 1000$ again, the runtime penalty of using $W = 1 < W_{opt}$ is even higher than that of using $W = 32$. In the end, the experiment of the two cache policies merely show a small improvement in comparison to our original implementation.

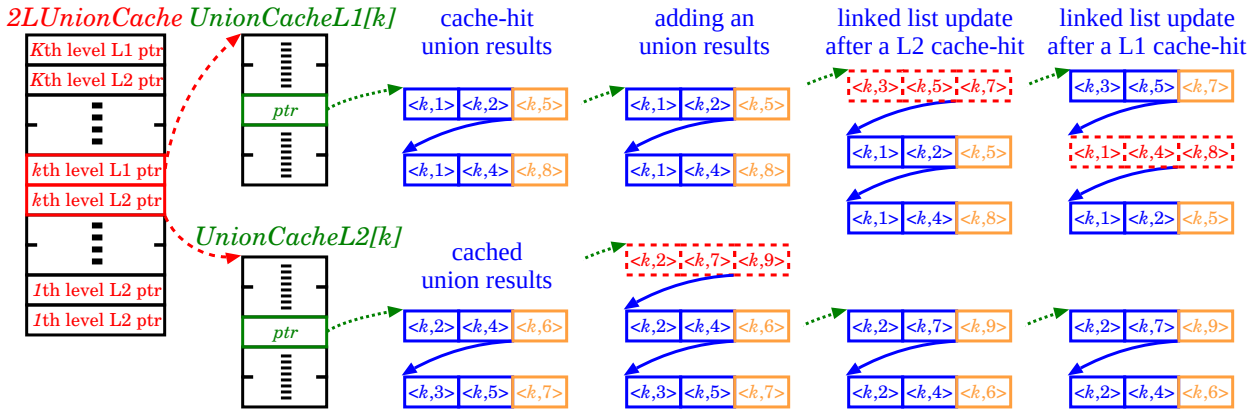


Figure 6: 2-levels union caches

6 CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

We designed and implemented a new version of the distributed symbolic state-space generator, SMART[DW], whose level-based node allocation scheme does achieve excellent memory distribution and scalability over NOWs. 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. Also, the cache heuristics does speedup our tool and makes the distributed approach more convincing.

Some future research directions are discussed below.

6.1 Two-level cache policies

Since the single-level cache policy mixes up the newly cached entries and the cache-hit entries, it might offset the idea of distinguishing the usefulness of each entry. In detail, the cache policy to distinguish the entries according to how frequent each of them has been retrieved makes more sense if the comparison is only among those cache-hit ones. Also, heavy element insertion might cancel out the usefulness of bookkeeping corresponding to some key. Thus, we suggest a two-level operation cache (L1 and L2) : L1 cache is used to store cache-hit entries; L2 cache is used to store newly cached entries. A newly computed result will be cached in L2 initially. Once some L2-cached entry is cache-hit, it will be moved to L1 cache. If an entry stored in L1 cache is cache-hit, its position will be changed. Fig. 6 shows the new structure we use to cache union operations. The three examples shown in the last three columns of Fig. 6 are: caching a newly computed result, the union of $\langle k,2\rangle\langle k,7\rangle$ is $\langle k,9\rangle$, sharing a key with the union of $\langle k,2\rangle\langle k,4\rangle$ and the union of $\langle k,3\rangle\langle k,5\rangle$; the update following an L2 cache hit on the result of the union of $\langle k,3\rangle\langle k,5\rangle$; the update following an L1 cache hit on the result of the union of $\langle k,1\rangle\langle k,4\rangle$. After all, the time to access each entry stored in this new hash table is still $\mathcal{O}(MaxAllowedCollision)$.

6.2 Parallel version of SMART[DW]

In [6], we introduced the idea of utilizing idle workstation time by firing events e with $Top(e) > k$ on saturated MDD nodes at level k *a priori*. Since we cannot know a priori whether such an event will need to be fired on p, and a naïve speculative scheme asking each idle workstation to compute *all* possible firings may require excessive memory, we introduce

the idea that workstations recognize *event firing patterns*, namely sequences of events that have been fired on MDD nodes so far, then speculatively explore only firings conforming to these patterns to prevent unrestrained speculation from squandering the overall NOW memory. Also, in [7], we explore how to encode the evolution of each firing pattern *implicitly*, so that MDD nodes can share the encoding of the same patterns. The implicit method for pattern encoding is not only for reducing the memory overhead of this prediction scheme but also for tuning the accuracy of speculation. In the near future, we plan to apply this speculative image computation idea to speedup the distributed reachability analysis on large scale systems on heterogeneous cluster (Myrinet connected single-chip multi-processing machines) using the combination of message passing (MPI) and shared memory (OpenMP) libraries.

REFERENCES

[1] Sciclone cluster project. http://www.compsci.wm.edu/sciclone/.

[2] Randy E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, August 1986.

[3] Peter Buchholz, Gianfranco Ciardo, Susanna Donatelli, and Peter Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. Comp.*, 12(3):203–222, 2000.

[4] Jerry R. Burch, Edmund M. Clarke, and David E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *Int. Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, August 1991. IFIP Transactions, North-Holland.

[5] Ming-Ying Chung and Gianfranco Ciardo. Saturation NOW. In *Proc. QEST*, pages 272–281, Enschede, The Netherlands, September 2004. IEEE Comp. Soc. Press.

[6] Ming-Ying Chung and Gianfranco Ciardo. A pattern recognition approach for speculative firing prediction in distributed saturation state-space generation. In Leucker Martin and Jaco van de Pol, editors, *Workshop on Parallel and Distributed Model Checking (PDMC)*, ENTCS, pages 65–79, Lisbon, Portugal, July 2005. Elsevier.

[7] Ming-Ying Chung and Gianfranco Ciardo. A dynamic firing speculation to speedup distributed symbolic state-space generation. In *Proc. International Parallel & Distributed Processing Symposium (IPDPS)*, Rhodes, Greece, April 2006. IEEE Computer Society.

[8] Ming-Ying Chung, Gianfranco Ciardo, and Andy Jinqing Yu. A fine-grained fullness-guided chaining heuristic for symbolic reachability analysis. In *Proc. International Symposium on Automated Technology for Verification and Analysis (ATVA)*, LNCS, Beijing, China, October 2006. Springer-Verlag.

[9] Gianfranco Ciardo and Yingjie Lan. Faster discrete-event simulation through structural caching. In *Proc. Sixth Int. Workshop on Performability Modeling of Computer and Communication Systems (PMCCS-6)*, pages 11–14, Monticello, IL, USA, September 2003.

[10] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In Mogens Nielsen and Dan Simpson, editors, *Proc. 21th Int. Conf. on Applications and Theory of Petri Nets*, LNCS 1825, pages 103–122, Aarhus, Denmark, June 2000. Springer-Verlag.

[11] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In Tiziana Margaria and Wang Yi, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2031, pages 328–342, Genova, Italy, April 2001. Springer-Verlag.

[12] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. Saturation unbound. In Hubert Garavel and John Hatcliff, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2619, pages 379–393, Warsaw, Poland, April 2003. Springer-Verlag.

[13] Gianfranco Ciardo and Radu Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. In Warren Hunt, Jr. and Fabio Somenzi, editors, *Computer Aided Verification (CAV'03)*, LNCS 2725, pages 40–53, Boulder, CO, USA, July 2003. Springer-Verlag.

[14] Gianfranco Ciardo and Andy Jinqing Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In Dominique Borrione and Wolfgang Paul, editors, *Proc. CHARME*, LNCS 3725, pages 146–161, Saarbrücken, Germany, October 2005. Springer-Verlag.

[15] Paulo Fernandes, Brigitte Plateau, and William J. Stewart. Efficient descriptor-vector multiplication in stochastic automata networks. *J. ACM*, 45(3):381–414, 1998.

[16] Susanne Graf, Bernhard Steffen, and Gerald Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Formal Asp. of Comp.*, 8(5):607–616, 1996.

[17] Orna Grumberg, Tamir Heyman, Nili Ifergan, and Assaf Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *Proc. CHARME*, LNCS 3725, pages 129–145, Saarbrücken, Germany, October 2005. Springer-Verlag.

[18] Tamir Heyman, Danny Geist, Orna Grumberg, and Assaf Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer-Aided Verification, 12th International Conference (CAV'00)*, volume 1855 of *LNCS*, pages 20–35, Chicago, IL, USA, 2000. Springer.

[19] Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. In *22th Annual Symp. on Foundations of Computer Science*, pages 150–158. IEEE Comp. Soc. Press, October 1981.

[20] Timothy Kam, Tiziano Villa, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[21] Kim Milvang-Jensen and Alan J. Hu. BDDNOW: A parallel BDD package. In H. Beilner and F. Bause, editors, *Formal Methods in Computer-Aided Design*, LNCS 1522, pages 501–507. Springer-Verlag, 1998.

[22] Andrew S. Miner and Gianfranco Ciardo. Efficient reachability set generation and storage using decision diagrams. In H.C.M. Kleijn and Susanna Donatelli, editors, *Proc. 20th Int. Conf. on Applications and Theory of Petri Nets*, LNCS 1639, pages 6–25, Williamsburg, VA, USA, June 1999. Springer-Verlag.

[23] Enric Pastor, Oriol Roig, Jordi Cortadella, and Rosa M. Badia. Petri net analysis using boolean manipulation. In Robert Valette, editor, *Proc. 15th Int. Conf. on Applications and Theory of Petri Nets*, LNCS 815, pages 416–435, Zaragoza, Spain, June 1994. Springer-Verlag.

[24] Radu Siminiceanu and Gianfranco Ciardo. Formal verification of the NASA Runway Safety Monitor. In Michael Huth, editor, *Proc. Fourth International Workshop on Automated Verification of Critical Systems (AVoCS'04)*, ENTCS, London, UK, September 2004. Elsevier.

[25] Anthony L. Stornetta. Implementation of an Efficient Parallel BDD Package. Master's thesis, University of California, Santa Barbara, 1995.

[26] Marco Tilgner, Yukio Takahashi, and Gianfranco Ciardo. SNS 1.0: Synchronized Network Solver. In *1st Int. Workshop on Manufacturing and Petri Nets*, pages 215–234, Osaka, Japan, June 1996.