A Dynamic Firing Speculation to Speedup Distributed Symbolic State-space Generation *

Ming-Ying Chung and Gianfranco Ciardo
University of California, Riverside
Department of Computer Science and Engineering
Riverside, CA 92521 USA
{chung, ciardo}@cs.ucr.edu

Abstract

The saturation *strategy for symbolic state-space generation is very effective for globally-asynchronous locally-synchronous discrete-state systems. Its inherently sequential nature, however, makes it difficult to parallelize on a NOW. An initial attempt that utilizes idle workstations to recognize event firing patterns and then speculatively compute firings conforming to these patterns is at times effective but can introduce large memory overheads. We suggest an implicit method to encode the firing history of decision diagram nodes, where patterns can be shared by nodes. By preserving the actual firing history efficiently and effectively, the speculation is more informed. Experiments show that our implicit encoding method not only reduces the memory requirements but also enables dynamic speculation schemes that further improve runtime.*

1. Introduction

Formal verification techniques such as model checking [12] are becoming widely used in industry for quality assurance, as they can be used to detect design errors early in the lifecycle. State-space generation is an essential, but very memory-intensive, step in model checking. Even though symbolic encodings based on *binary decision diagrams* (BDDs) [2] and *multiway decision diagrams* (MDDs) [17] help cope with the inherent *state-space explosion* of discrete-state systems, the analysis of a complex system may still rely heavily on the use of virtual memory. Approaches employing decision diagrams to encode the state space are said to be *symbolic*, to distinguish them from *explicit* approaches where states are discovered and stored one-by-one.

*Work supported in part by the National Science Foundation under grants CNS-0501747 and CNS-0501748.

Much research in this area has focused on parallel and distributed algorithms. For explicit state-space generation or model checking, [1, 22, 27] introduce algorithms that utilize the overall resources of a network of workstations (NOW). However, the size of the state space that can be handled by explicit approaches is usually much smaller than with symbolic approaches. For symbolic state-space generation or model checking, most works employ a *vertical* slicing scheme to parallelize BDD manipulations by decomposing boolean functions in breadth-first fashion and distributing the computation over a NOW [16, 19, 28]. This scheme allows algorithms to overlap the application of the next-state function to a set of states encoded by a decision diagram node (the so-called *image computation*), but no speedup information is reported. We detail this slicing scheme in Sect. 5.

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. Thus, the memory required for the state-space encoding is also exclusively partitioned onto the workstations. Since the distributed state-space generation does not create any redundant work at all, synchronization is avoided. Furthermore, with the horizontal slicing scheme, communication is required only between neighbor workstations, thus peer-to-peer communication suffices and scalability is not an issue. Yet, the approach implies a tradeoff: to maintain canonicity of the MDD over a NOW, the distributed computation is sequentialized. At any point in time, all workstations except one are waiting either for work requests or for a reply from their neighbors. Thus, this approach appears to provide no easy opportunity for speedup.

In [6], we tackle this drawback by using workstations idle time to speculatively fire events on MDD nodes, hoping that many of of these firings will be needed later in the computation. To prevent unrestrained specula-

tion from squandering the overall NOW memory, we introduce the idea that workstations recognize *event firing patterns*, namely sequences of events that have been fired on MDD nodes, at runtime, then speculatively explore only firings conforming to these patterns.

However, storing and recognizing firing patterns to help workstations improve the accuracy of prediction requires a nontrivial memory overhead. In this paper, we introduce an implicit method to encode firing patterns into graphs, so that MDD nodes can share the encoding of these patterns. This encoding method can not only reduce the memory overhead required by pattern recognition, but, most importantly, it also provides better information about the evolution of each pattern, allowing for more accurate speculation.

Our paper is organized as follows. Sect. 2 gives the necessary background on state-space generation, decision diagrams, Kronecker encoding, and saturation. Sect. 3 details our implicit method for firing pattern encoding and introduces two dynamically adjustable speculation schemes. Sect. 4 shows experimental results. Sect. 5 compares our approach with other related work. Sect. 6 draws conclusions and discusses future research directions.

2. Background

A discrete-state model is a triple $(\widehat{\mathcal{S}}, \mathbf{s}^0, \mathcal{N})$, where $\widehat{\mathcal{S}}$ is the set of *potential states* of the model, $\mathbf{s}^0 \in \widehat{\mathcal{S}}$ 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 seek to build the *(reachable) state space* $\mathcal{S} \subseteq \widehat{\mathcal{S}}$, the smallest set containing $\mathbf{s}^0$ and closed with respect to $\mathcal{N}$: $\mathcal{S} = \{\mathbf{s}^0\} \cup \mathcal{N}(\mathbf{s}^0) \cup \mathcal{N}(\mathcal{N}(\mathbf{s}^0)) \cup \cdots = \mathcal{N}^*(\mathbf{s}^0)$, 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 [9]. We then use the mappings $\psi_k : \mathcal{S}_k \to \{0, 1, ..., n_k - 1\}$, 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 $\{0, 1, ..., n_k - 1\}$, 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:

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

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.

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* [7, 8]. 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 $(i_K, ..., i_1) \in \widehat{\mathcal{S}}$,

$$\mathcal{N}_e(i_K, ..., i_1) = \mathcal{N}_{K,e}(i_K) \times \cdots \times \mathcal{N}_{1,e}(i_1).$$

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 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.,

$$\mathbf{N}_{y,b} = \left\{ \begin{array}{l} 1 \to 0 \\ 2 \to 1 \end{array} \right\} \text{ means } \mathbf{N}_{y,b} = \left[\begin{array}{ccc} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{array} \right]$$

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* [7, 20] and employ *saturation* [8]. 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 $\mathbf{s}^0$ and saturate its nodes bottom up, the root r will encode $\mathcal{S} = \mathcal{N}^*(\mathbf{s}^0)$ at the end, because: (1) $\mathcal{N}^*(\mathbf{s}^0) \supseteq \mathcal{B}(r) \supseteq \{\mathbf{s}^0\}$, 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. Results in [8, 9, 10] consistently show that saturation outperforms breadth-first 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

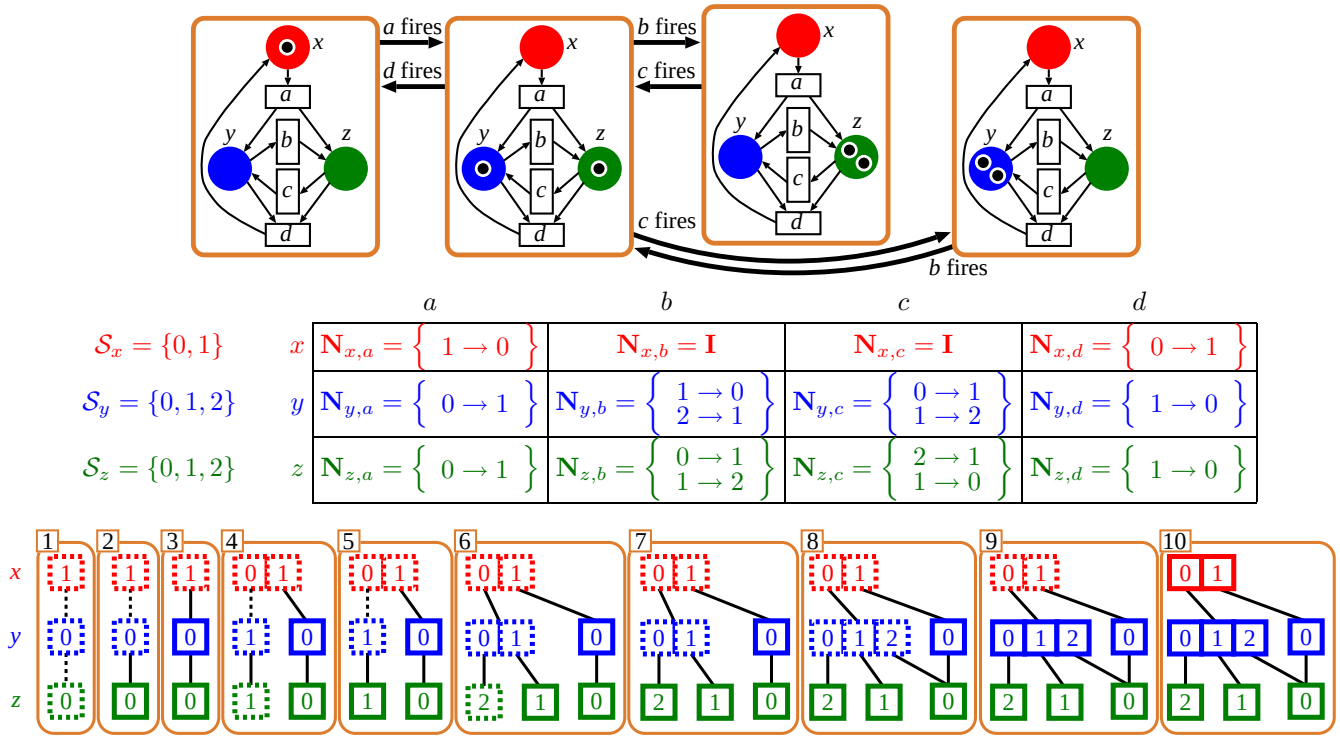

Figure 1. Reachability graph (top), $\mathcal{S}_x, \mathcal{S}_y, \mathcal{S}_z$, and $\mathcal{N}$ (middle), evolution of the MDD (bottom).

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.

2.4. Distributed version of saturation

In [5], we present *SaturationNOW*, a message-passing algorithm that distributes the MDD nodes encoding the state space onto a NOW, to study large models where a single workstation would have to rely on virtual memory to perform reachability analysis. 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.

In addition, [5] introduces a nested memory load balancing approach to cope with dynamic memory requirements by reassigning MDD levels, i.e., changing $mybot_w$ and $mytop_{w-1}$ of two neighbors. Since memory load balancing requests can propagate, each workstation can effectively rely on the overall NOW memory, not just that in its neighbors, without the need for global synchronization or broadcasting.

2.5. Parallel version of saturation

Our horizontal slicing approach effectively partitions the MDD memory onto the workstations, but it also strictly sequentializes the distributed computation, making it a challenge to achieve a speedup. Thus, we introduced the idea of using idle workstation time to fire events e with $Top(e) > k$ on saturated MDD nodes at level k *a priori*, hoping to reduce the time required to saturate MDD nodes at levels above k [6].

An MDD node p at level k is saturated if all events e with $Top(e) = k$ have been fired exhaustively on p. However, an event f with $Top(f) = l > k \geq Bot(f)$

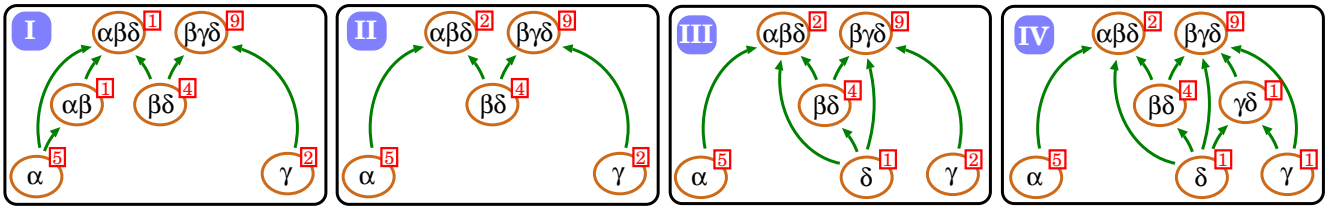

Figure 2. Updating the firing pattern graph.

may still need to be fired on p when saturating some MDD node q at level l. To reduce the time to saturate such hypothetical MDD node q, we speculatively create the (possibly disconnected) MDD node p' corresponding to the saturation of the result of firing f on p, and cache the result. Later on, firing f on p immediately returns the result p' found in the cache.

Unfortunately, we cannot know a priori whether such an event will be fired on a node p. A naïve speculative firing that lets an idle workstation compute *all* possible firings starting above the level k of each MDD node p it owns, $\mathcal{E}_{all}(p) = \{e \in \mathcal{E} : Top(e) > k \geq Bot(e)\}$, may require excessive memory. Thus, we explored an informed prediction based on firing patterns.

For each MDD node p at level k, let $\mathcal{E}_{patt}(p)$ be the set of events e that will be fired on p after p has been saturated, thus, $Top(e) > k$ and $\mathcal{E}_{patt}(p) \subseteq \mathcal{E}_{all}(p)$. We could then partition the MDD nodes at level k according to their patterns, i.e., MDD nodes p and q are in the same class iff $\mathcal{E}_{patt}(p) = \mathcal{E}_{patt}(q)$. Of course, $\mathcal{E}_{patt}(p)$ can be known only *a posteriori*, but it should be observed that most models exhibit clear firing patterns during saturation, i.e., most classes contain many MDD nodes and most patterns contain several events. The idea is to *speculate* the pattern of a given MDD node p based on the *history* of the events fired on p so far, $\mathcal{E}_{hist}(p) \subseteq \mathcal{E}_{patt}(p)$. Thus, if $\emptyset \subset \mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$, we can speculate that each $e \in \mathcal{E}_{hist}(q) \setminus \mathcal{E}_{hist}(p)$ will eventually need to be fired on p as well, which is true if $\mathcal{E}_{patt}(p) = \mathcal{E}_{patt}(q)$ at the end.

3. Implicit encoding method

To improve the efficiency and accuracy of speculation, we now explore how to encode the evolution of firing patterns for the MDD nodes. At the same time, we also seek to reduce the memory requirements to store this auxiliary information, which were already shown to be potentially substantial for the approach of [6]. The idea is to encode firing patterns *implicitly*, so that MDD nodes can share the encoding of the same patterns, while reducing overhead at same time.

3.1. Firing pattern graph

In our implicit encoding, firing patterns are stored in a *directed acyclic* graph, $G_k = (V_k, E_k)$ for $K \geq k \geq 1$. To distinguish the nodes of this graph from the MDD nodes, from now on, we refer to them as pattern graph (PG) nodes. Every PG node $v \in V_k$ represents a set of events that has been fired so far on one or more MDD nodes at level k after saturating them, $\mathcal{E}_v \subseteq \mathcal{E}_{>k} = \{e \in \mathcal{E} : Top(e) > k \geq Bot(e)\}$. Then, each MDD node p at level k does not store its own firing history $\mathcal{E}_{hist}(p)$ explicitly, it instead references the PG node $v \in V_{p.lvl}$ such that $\mathcal{E}_v$ is exactly the set of events that has been fired on p so far during state-space generation, i.e., $\mathcal{E}_{hist}(p) = \mathcal{E}_v$. Note that V_k contains only PG nodes corresponding to the current patterns of the current MDD nodes, thus, in practice, $|V_k| \ll 2^{|\mathcal{E}_{>k}|}$.

In addition, for any pair of distinct PG nodes u and v in V_k, there is an arc (u, v) in E_k iff $\mathcal{E}_u \subset \mathcal{E}_v$. In other words, the graph describes a hoped-for evolution of patterns during saturation for the MDD nodes at level k. In our implementation, every $v \in V_k$ maintains a set of PG node pointers, $v.parent = \{u \in V_k : (v, u) \in E_k\}$, and a reference counter $v.ref > 0$ recording the number of MDD nodes having firing pattern $\mathcal{E}_v$. If $v.ref$ becomes 0, PG node node v is not referenced by any MDD node, thus it can be removed. The time required to update the graphs is nontrivial, but workstations perform these updates only when idle.

Fig. 2 shows three examples of this graph updating. (I) shows the initial runtime snapshot of a firing pattern graph. (II) shows the updating due to firing event δ on the MDD node with current firing pattern $\{\alpha, \beta\}$. (III) shows the updating due to firing event δ on a newly created MDD node (not referencing any PG node). (IV) shows the updating due to firing event δ on one of the two MDD nodes with firing pattern $\{\gamma\}$.

3.2. Pattern graph based speculation

Obviously, applying our implicit method to the firing pattern encoding not only dramatically reduces the memory overhead introduced by pattern recognition,

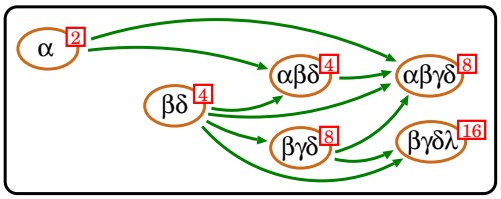

Figure 3. Firing pattern graph for dynamically adjustable speculation.

but, most importantly, it also records more information about the evolution of each pattern, so that the speculation on event firings has the potential to be more accurate.

Every path in the graph reveals a possible evolution of some pattern. Each outgoing arc of a PG node indicates one possible growth of the corresponding pattern. A basic event firing speculation for MDD node p at level k referencing PG node $v \in V_k$ consists of firing event $e = \mathcal{E}_u \setminus \mathcal{E}_v$ on p, for some $u \in v.parent$ such that $|\mathcal{E}_u \setminus \mathcal{E}_v| = 1$. There might be multiple ways to choose PG node u starting from v. More aggressive speculations may consider all such choices, or even consider PG nodes u such that $|\mathcal{E}_u \setminus \mathcal{E}_v| > 1$. The next section discusses different heuristics to use this graph.

Our goal is to maximize the usefulness of the speculative firing results while minimizing the space and time overhead. To this end, we dynamically adjust the speculation aggressiveness based on a runtime metric, the *speculation hit rate*, i.e., the fraction of speculative firing results put in the cache that is actually requested at least once later for actual (non speculative) firings originated from MDD nodes above. Each workstation records this hit rate dynamically for each MDD level and, if the hit rate increases, a workstation can adjust its speculation to become more aggressive at that level; on the other hand, if the hit rate is poor, a workstation can become more conservative in its speculation.

3.3. Pattern length based speculation

To perform a dynamically adjustable speculation through the firing pattern graph, each workstation can initialize a variable, *MaxDiff*, indicating the maximum number of events that can be fired speculatively to reach a PG node from another. Whenever a workstation is idle, for each MDD node p referencing PG node v, the workstation speculatively fires e on p for $e \in \mathcal{E}_u \setminus \mathcal{E}_v$, for any $u \in v.parent$ satisfying $|\mathcal{E}_u| - |\mathcal{E}_v| \leq MaxDiff$. Thus, a workstation can make speculation more aggressive by increasing its value of *MaxDiff*, and more conservative by reducing it.

For example, in Fig. 3, when $MaxDiff = 1$, a workstation managing MDD node p with firing pattern $\{\beta, \delta\}$ may fire events α and γ on p, since the firing patterns $\{\alpha, \beta, \delta\}$ and $\{\beta, \gamma, \delta\}$, both present in the firing pattern graph, differ from the firing pattern of p by just one more element. If $MaxDiff = 2$, the workstation may instead fire events α, γ, and λ on p, since now also the size of pattern $\{\beta, \gamma, \delta, \lambda\}$ is within *MaxDiff* of that of the firing pattern of p (pattern $\{\alpha, \beta, \gamma, \delta\}$ also qualifies, but it is already obtained as the union of patterns $\{\alpha, \beta, \delta\}$ and $\{\beta, \gamma, \delta\}$).

3.4. Weighted score based speculation

The previous heuristic uses the reference counter only to discard patterns when no MDD node references them. A finer speculation scheme could instead have workstations adjust the aggressiveness of prediction based on the value of the reference counters (higher count indicates more popular patterns), and the length of the involved patterns (if the target pattern differs by having many more elements, it is more expensive to reach and perhaps less likely to be a good guess). Thus, we define the *weighted score* from u to v as

$$Score(u, v) = v.ref/(|\mathcal{E}_v| - |\mathcal{E}_u|).$$

Then, a workstation calculates the score of each PG node pair (u, v) and performs the speculative firing of the events in $\mathcal{E}_v \setminus \mathcal{E}_u$ on the PG nodes having pattern $\mathcal{E}_u$ if $Score(u, v)$ meets some threshold value *MinScore*. Again, if the hit rate is good, the workstation can dynamically make speculation more or less aggressive by decreasing or increasing the value of *MinScore*.

For example, in Fig. 3, if $MinScore = 6$, a workstation managing MDD node p with firing pattern $\{\beta, \delta\}$ may fire events γ and λ on p, since, considering PG node $\{\beta, \gamma, \delta, \lambda\}$ with reference count 16, we have $16/(4-2) = 8 \geq MinScore$. If $MinScore = 4$, the workstation may instead fire α, γ and λ on p, since now, considering PG node $\{\alpha, \beta, \gamma\}$ with reference count 4, we have $4/(3-2) = 4 \geq MinScore$.

The workstations can initialize, and, especially, update, *MaxDiff* or *MinScore* differently from each other, depending on the range of MDD levels they own and on how aggressive they want to be based on the fraction of idle time and amount of memory they have.

4. Experimental results

We implemented both heuristic schemes in SMART-NOW [5], the MPICH-based distributed version of our tool SMART [11], and evaluated their performance by using saturation to generate the state space of the following models, all parametrized by a value N.

- *Flexible manufacturing system* [20] models a manufacturing system with three machines to process three different types of parts. N is the number of each type of parts.
- *Slotted ring network protocol* [24] models a protocol for local area networks. N is the number of nodes in the network.
- *Round robin mutex protocol* [14] models a round robin solution to the mutual exclusion problem. N is the number of processes involved.
- *Runway safety monitor* [26] models an avionics system monitoring T targets with S speeds on a grid represented as a $X \times Y \times Z$ grid.

We run our implementation on this four models using a cluster of Pentium IV 3GHz workstations with 512MB RAM each, connected by Gigabit Ethernet and running Red-Hat 9.0 Linux with MPI2 on TCP/IP. Table 1 shows runtimes, total memory requirements for W workstations, and the maximum memory requirements for any workstation, for sequential SMART (SEQ) [8] and the original SMART-NOW (DISTR) [5], and the percentage change w.r.t. DISTR for the naïve (NAÏVE) [6], history-based (HIST) [6], and the new pattern-length-adjusted (LENGTH) and weighted-score-adjusted (SCORE) speculative firing predictions; "*d*" means that dynamic memory load balancing is triggered, "*s*" means that, in addition, memory swapping occurs. For the LENGTH heuristic, we initialize *MaxDiff* to 2 and increase/decrease it by 2 whenever the speculation hit rate increases/decreases 5%. For the SCORE heuristic, we initialize *MinScore* as 100 and divide/multiply it by 2 whenever the speculation hit rate increases/decreases 5%. For each model, the size of the state space is also reported.

Experiments on the flexible manufacturing system model show that both LENGTH and SCORE outperform HIST in terms of runtime and memory consumption. Thus, both approaches outperform the original DISTR in terms of runtime as well, more so (up to 50%) as the number of workstations W increases. Also, the memory overhead required to store patterns for either LENGTH or SCORE is much less than for HIST.

The experiments on the slotted ring network protocol show a best-case example for HIST. While all heuristic speculative approaches improve over DISTR, neither LENGTH nor SCORE outperforms HIST in terms of runtime, since the firing patterns within this model is somewhat regular in comparison to other models. The time invested in organizing these relatively regular firing patterns into graphs actually slows down the parallel computation, since less time is spent on speculative firing. However, the new implicit encoding method nevertheless reduces the memory overhead required by pattern recognition.

The experiments on the round robin mutex protocol show a worst-case example for the speculative approach, as no useful firing pattern exists. However, at least, the memory overhead of LENGTH and SCORE is much smaller than for HIST because of the efficient encoding of patterns. Even more importantly, both LENGTH and SCORE approaches exhibit only a small worsening of their runtime, showing that their heuristics make them refrain from performing too many useless speculation, while this is not the case for HIST.

Finally, the experiment on the last model, a real system being developed by National Aeronautics and Space Administration (NASA) [26], shows that both LENGTH and SCORE approaches again outperform the HIST approach in terms of both runtime or memory consumption, suggesting that our implicit encoding method and dynamically adjustable speculation schemes work well on realistic models.

5. Related work

The symbolic approach is attractive because it allows decision diagram nodes to share not only state encodings but also intermediate results, during symbolic state-space generation. Since intermediate results are cached with respect to each decision diagram node and since these nodes are canonical, image computation greatly benefits when there is a high hit rate on these caches. In other words, the more state encodings and intermediate results are shared, the greater efficiency symbolic approaches exhibit with respect to explicit ones. However, this makes parallelizing symbolic state-space generation extremely challenging. Researchers working on distributed symbolic state-space generation over a NOW have merely focused on effectively utilizing the overall memory rather than the overall computational power.

[16, 19, 28] employ a vertical slicing scheme to parallelize BDD manipulations by decomposing boolean functions in breadth-first fashion and distributing the image computation over a NOW. Each workstation w computes a single slice of the global image, $\mathcal{S}_w \subseteq \mathcal{S}$, and $\mathcal{S} = \bigcup_{1 \leq w \leq W} \mathcal{S}_w$. Fig. 4 shows an example of

W	**Time** (sec)					**Total Memory** (MB)					**Max Memory** (MB)				
	DISTR	NAÏVE	HIST	LENGTH	SCORE	DISTR	NAÏVE	HIST	LENGTH	SCORE	DISTR	NAÏVE	HIST	LENGTH	SCORE
Flexible manufacturing system $N = 300$ $\lvert\mathcal{S}\rvert = 3.64 \cdot 10^{27}$											SEQ completes in 55 sec using 241MB				
2	79	−8%	−8%	−8%	−11%	243	+12%	+24%	+3%	+5%	121	+26%	+52%	+31%	+38%
4	91	^{d}+67%	−9%	−13%	−20%	243	+102%	+30%	+11%	+14%	119	+205%	+50%	+27%	+29%
8	260	-	−30%	−44%	−50%	243	-	+42%	+16%	+22%	103	-	+47%	+21%	+28%
$N = 450$ $\lvert\mathcal{S}\rvert = 6.90 \cdot 10^{29}$											SEQ does not complete in 5 hrs using 512MB				
2	^{s}257	^{s}+12%	^{s}−14%	^{s}−10%	^{s}−14%	826	+16%	+5%	+4%	+4%	512	+15%	+7%	+2%	+4%
4	^{d}311	$>$ 5hrs	^{d}−18%	^{d}−29%	^{d}−30%	826	-	+33%	+9%	+17%	372	-	+6%	+2%	+5%
8	959	$>$ 5hrs	−25%	−34%	−38%	826	-	+61%	+24%	+39%	343	-	+6%	+5%	+8%
Slotted ring network protocol $N = 200$ $\lvert\mathcal{S}\rvert = 8.38 \cdot 10^{211}$											SEQ completes in 108 sec using 284MB				
2	119	−24%	−13%	−5%	−8%	286	+3%	+45%	+12%	+14%	197	+1%	+53%	+3%	+6%
4	139	−27%	−15%	−6%	−9%	286	+11%	+51%	+17%	+26%	127	+61%	+58%	+10%	+20%
8	182	−32%	−24%	−14%	−20%	286	+129%	+62%	+20%	+29%	69	+239%	+62%	+14%	+37%
$N = 300$ $\lvert\mathcal{S}\rvert = 8.38 \cdot 10^{211}$											SEQ does not complete in 5 hrs using 512MB				
2	^{s}552	^{s}+5%	^{s}−5%	^{s}−10%	^{s}−7%	962	+25%	+11%	+6%	+10%	562	+8%	+7%	+3%	+5%
4	^{d}490	$>$ 5hrs	^{d}−16%	^{d}−18%	^{d}−14%	962	-	+34%	+13%	+19%	352	-	+12%	+8%	+11%
8	564	$>$ 5hrs	−39%	−24%	−30%	962	-	+50%	+21%	+29%	252	-	+23%	+13%	+19%
Round robin mutex protocol $N = 800$ $\lvert\mathcal{S}\rvert = 1.20 \cdot 10^{196}$											SEQ completes in 27 sec using 290MB				
2	29	+37%	+6%	+0%	+0%	293	+110%	+85%	+21%	+27%	215	+52%	+63%	+15%	+22%
4	36	+33%	+8%	+0%	+0%	293	+348%	+109%	+28%	+37%	130	+186%	+65%	+18%	+23%
8	51	+33%	+5%	+0%	+0%	293	+807%	+148%	+36%	+49%	73	+433%	+73%	+21%	+25%
$N = 1100$ $\lvert\mathcal{S}\rvert = 3.36 \cdot 10^{334}$											SEQ does not complete in 5 hrs using 512MB				
2	^{d}65	^{s}+62%	^{s}+18%	^{d}+5%	^{d}+9%	794	+46%	+6%	+2%	+3%	379	+79%	+30%	+2%	+5%
4	47	^{s}+131%	^{d}+10%	+2%	+2%	794	+119%	+38%	+3%	+5%	265	+104%	+40%	+7%	+10%
8	56	^{d}+164%	+7%	+2%	+2%	794	+299%	+50%	+9%	+10%	173	+126%	+38%	+23%	+26%
Runway safety monitor $Z = 2$ $\lvert\mathcal{S}\rvert = 1.51 \cdot 10^{15}$											SEQ completes in 236 sec using 314MB				
2	731	$>$ 10hrs	−2%	−3%	−5%	332	-	+39%	+4%	+9%	191	-	+48%	+11%	+18%
4	938	$>$ 10hrs	−8%	−16%	−18%	332	-	+88%	+20%	+28%	190	-	+30%	+6%	+15%
8	1480	$>$ 10hrs	−22%	−27%	−31%	332	-	+128%	+67%	+90%	173	-	+13%	+6%	+10%
$Z = 3$ $\lvert\mathcal{S}\rvert = 5.07 \cdot 10^{15}$											SEQ does not complete in 10 hrs using 512MB				
2	^{s}11280	$>$ 10hrs	^{s}−1%	^{s}−2%	^{s}−3%	962	-	+10%	+3%	+4%	595	-	+16%	+2%	+4%
4	^{d}9762	$>$ 10hrs	^{d}−15%	^{d}−19%	^{d}−26%	962	-	+31%	+4%	+13%	371	-	+8%	+3%	+5%
8	^{d}14101	$>$ 10hrs	^{d}−17%	^{d}−29%	^{d}−31%	962	-	+58%	+8%	+35%	359	-	+6%	+3%	+4%

Table 1. Experimental results.

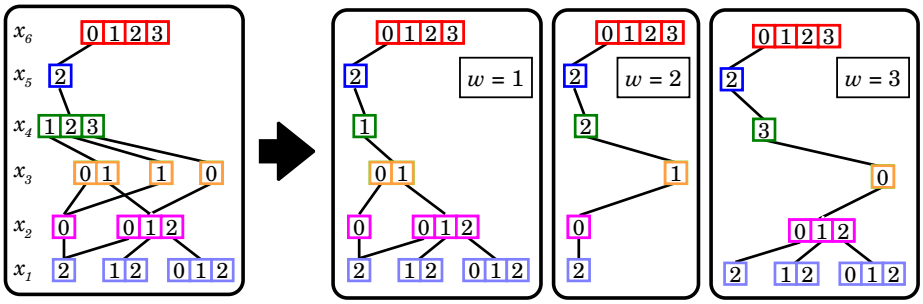

Figure 4. Vertically sliced MDD.

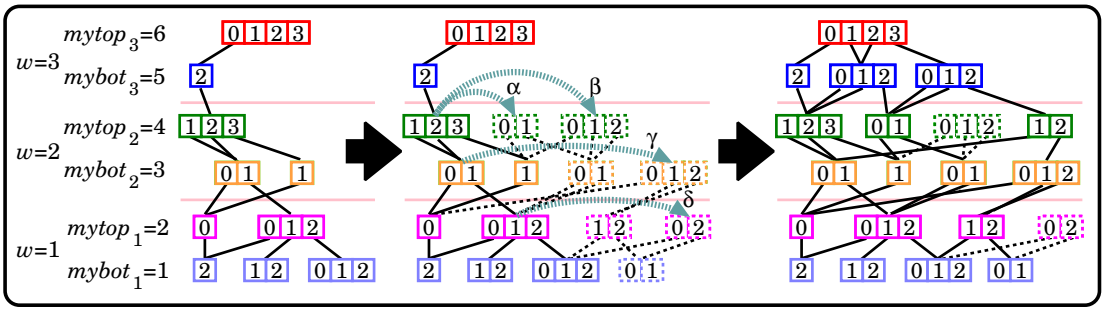

Figure 5. Speculative firing at work.

slicing a six-level MDD vertically into three portions according to the value of variable x_4. However, some MDD nodes at levels 6, 5, 2, and 1 are duplicated in multiple workstations. A poor slicing creates many additional decision diagram nodes, and it is generally agreed that finding a good slicing is not trivial [21].

Even though this slicing scheme allows algorithms to overlap the distributed image computation, occasional synchronization is required to minimize redundancy, which can harm the scalability of algorithms. To improve scalability, researchers have recently focused on workload distribution. [15] suggests to employ a host processor to manage the job queue for load-balance purposes, and to reduce the redundancy in the overlapped image computation by slicing it according to boolean functions that use an optimal choice of variables in order to minimize the peak number of BDD nodes required by the workstations. However, as far as we know, no speedup has been reported so far.

Instead, in our horizontal slicing scheme where each workstation owns a contiguous range of decision diagram levels, the distributed image computation does not create any redundant work at all, and global synchronization is avoided. Furthermore, the scheme requires only communication between neighbors, so scalability is not an issue, given the right hardware (e.g., a ring interconnection). Largely independent speculative computing on event firing can then be used to speedup the computation, while the pattern recognition approach we introduced is effective at avoiding excessive speculation in pathological models.

Of course, just like the redundant work introduced by vertical slicing, our approach also can introduce some useless work. More precisely, even though the portion of the MDD reachable from the root remains canonical, additional disconnected MDD nodes can be generated. Fig. 5 shows how speculative firing can create useless MDD nodes at work, where the "dashed" boxes indicated the disconnected MDD nodes. In the middle of the figure, workstations 2 and 1 have predicted and computed firings for α and β at level 4, γ at level 3, and δ at level 2. On the right, the MDD nodes resulting from firing α or γ are now connected, as they were actually needed and found in the cache. However, two disconnected MDD nodes remain, since the firing of γ and δ turned out to be useless.

Our implicit encoding method and dynamically adjustable speculation schemes further improve the accuracy of speculation (thus accelerate distributed state-space generation) and contain the growth of useless MDD nodes (thus reduce the memory overhead). While our approach at best achieves a speedup of 17% with respect to the best sequential implementation, it opens the door for greater speedups, while exhibiting near-optimal memory distribution and perfect node partitioning (no MDD node duplication) over a NOW.

6. Conclusions and future work

We presented an implicit method to efficiently encode the pattern of firings computed on the nodes of the decision diagram used to encode the state space of a discrete-state model, as it is being built. These patterns are then used by the workstations on a NOW to carry on informed speculative event firings on the decision diagram nodes, in the hope that they are needed later on in the computation. The implicit encoding can not only reduce the memory overhead required for pattern recognition, but also effectively record the evolution of each firing pattern, so that it is possible to dynamically adjust the speculation aggressiveness. Experiments show improvements on a realistic model.

We envision several possible extensions. First, having showed the potential of low-overhead speculative firing, we plan to apply the idea of speculative firing to the general problem of temporal logic model checking. In particular, while our idea is implemented for a saturation-style iteration, it is also applicable to the simpler breadth-first iteration needed by some of the CTL model checking algorithms, a fact we intend to explore. Second, we plan to develop heuristics to perform symbolic state-space generation with a variable number of workstations, so that the parallel computation uses only the least number of workstations needed, while still having a chance to obtain greater speedups.

References

[1] A. Bell and B. Haverkort. Sequential and distributed model checking of Petri nets. *STTT*, 7(1):43–60, 2005.

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

[3] 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.*, 12(3):203–222, 2000.

[4] J. Burch, E. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. Proc. *VLSI*, pp.49–58, 1991.

[5] M.-Y. Chung and G. Ciardo. Saturation NOW. Proc. *QEST*, pp.272–281, 2004.

[6] M.-Y. Chung and G. Ciardo. A pattern recognition approach for speculative firing prediction in distributed saturation state-space generation. Proc. *PDMC*, pp.65–79, 2005.

[7] G. Ciardo, G. Lüettgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. Proc. *ICATPN*, pp.103–122, 2000.

[8] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state-space generation. Proc. *TACAS*, pp.328–342, 2001.

[9] G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. Proc. *TACAS*, pp.379–393, 2003.

[10] G. Ciardo and R. Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. Proc. *CAV*, pp.40–53, 2003.

[11] G. Ciardo, R. Jones, A. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. *Performance Evaluation*, to appear.

[12] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. 1999.

[13] S. Gai, M. Rebaudengo, and M. Sonza Reorda. A data parallel algorithm for boolean function manipulation. Proc. *FMPSC*, pp.28–36, 1995.

[14] S. Graf, B. Steffen, and G. Lüttgen. Compositional minimisation of finite state systems using interface specification. *Formal Asp. of Comp.*, 8(5):607–616, 1996.

[15] O. Grumberg, T. Heyman, and A. Schuster. A work-efficient distributed algorithm for reachability analysis. Proc. *CAV*, pp.54–66, 2003.

[16] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits. Proc. *CAV*, pp.20–35, 2000

[17] T. Kam, T. Villa, R.K. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[18] S. Kimura and E. Clarke. A parallel algorithm for constructing binary decision diagrams. Proc. *ICCD*, pp.220–223, 1990.

[19] K. Milvang-Jensen and A. Hu. BDDNOW : A parallel BDD package. Proc. *FMCAD*, pp.501–507, 1998.

[20] A. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. Proc. *ICATPN*, pp.6–25, 1999.

[21] A. Narayan, A. Isles, J. Jain, R. Brayton, and A. Sangiovanni-Vincentelli. Reachability analysis using Partitioned-ROBDDs. Proc. *ICCAD*, pp.388–393, 1997.

[22] D. Nicol and G. Ciardo. Automated parallelization of discrete state-space generation. *J. Par. and Distr. Comp.*, 47:153–167, 1997.

[23] Y. Parasuram, E. Stabler, and S.-K. Chin. Parallel implementation of BDD algorithm using a distributed shared memory. Proc. *HICSS*, pp.16–25, 1994.

[24] E. Pastor, O. Roig, J. Cortadella, and R. Badia Petri net analysis using boolean manipulation. Proc. *ICATPN*, pp.416–435, 1994.

[25] R. Ranjan, J. Snaghavi, R. Brayton, and A. Sangiovanni-Vincentelli. Binary decision diagrams on network of workstations. Proc. *ICCD*, pp.356–364, 1996.

[26] R. Siminiceanu and G. Ciardo. Formal verification of the NASA Runway Safety Monitor. Proc. *AVoCS*, 2004.

[27] U. Stern and D. L. Dill. Parallelizing the Murϕ verifier. Proc. *CAV*, pp.256–267, 1997.

[28] T. Stornetta and F. Brewer. Implementation of an efficient parallel BDD package. Proc. *DAC*, pp.641–644, 1996.

[29] B. Yang and D. O'Hallaron. Parallel breadth-first BDD construction. Proc. *PPoPP*, pp.145–156, 1997.