

Speculative Image Computation for Distributed Symbolic Reachability Analysis*

Ming-Ying Chung	Gianfranco Ciardo
Verification Group Synopsys, Inc. Mountain View, CA 94085 mychung@synopsys.com	Department of Computer Science and Engineering University of California, Riverside Riverside, CA 92521 ciardo@cs.ucr.edu

Abstract

The Saturation-style fixpoint iteration strategy for symbolic reachability analysis is particularly effective for globally-asynchronous locally-synchronous discrete-state systems. However, its inherently sequential nature makes it difficult to parallelize Saturation on a NOW. We then propose the idea of using idle workstation time to perform speculative image computations. Since an unrestrained prediction may make excessive use of computational resources, we introduce a history-based approach to dynamically recognize image computation (event firing) patterns and explore only firings that conform to these patterns. In addition, we employ an implicit encoding for the patterns, so that the actual image computation history can be efficiently preserved. Experiments not only show that image speculation works on a realistic model, but also indicate that the use of an implicit encoding together with two heuristics results in a better informed speculation.

Keywords: reachability analysis, decision diagrams, image computation, distributed computing.

1 Introduction

Formal verification techniques such as model checking [14] 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, also called reachability analysis, is an essential, but very memory-intensive, step in model checking. Even though implicit encodings based on *binary decision diagrams* (BDDs) [2] and *multiway decision diagrams* (MDDs) [20] help cope with the inherent *state-space explosion*, the analysis of a complex discrete-state system may still rely 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.

To further increase the size of systems that can be studied, much formal verification research has recently focused on parallel and distributed algorithms. For explicit reachability analysis or model checking, [1, 24, 27] introduce algorithms that use the overall resources of a network of workstations (NOW). For symbolic reachability analysis or model checking, most works employ a *vertical* slicing scheme to parallelize BDD manipulations by spawning multiple image computations over the NOW, corresponding to distinct sets of paths through the BDD [19, 21, 28]. Algorithms following this scheme can overlap the application of the next-state function to sets of states encoded by different decision diagram node, that is, they can truly parallelize the *image computation*. We detail this slicing scheme in Section 6. In addition to these works on symbolic and explicit approaches, [15] develops a distributed approach for SAT-based bounded model checking, where the SAT problem is partitioned onto a NOW and clients can efficiently communicate with each others according to the partition topology.

*Work supported in part by the National Science Foundation under grant CNS-0619223 and ATM-0428880.

In [5], we instead encode the state space with an MDD [20] *horizontally* partitioned onto a NOW, so that each workstation exclusively owns a contiguous range of MDD levels. The memory required for the MDD state-space encoding is thus also exclusively partitioned onto the workstations. Since the distributed state-space generation does not create any redundant work, synchronization is avoided. Furthermore, with the horizontal slicing scheme, communication is required only between neighbor workstations, thus peer-to-peer communication suffices and good scalability is achieved. Yet, the approach implies a tradeoff: to maintain the MDD canonicity, the distributed computation is sequentialized. At any point in time, all workstations except one are waiting either for work requests from their neighbor above, or for a reply from their neighbor below. In other words, this approach uses memory optimally, but appears to provide no easy opportunity for speedup.

In this paper, we tackle this drawback by using workstations idle time to perform speculative image computation, in the hope that many of these anticipated results will be needed later in the computation. To prevent unrestrained prediction from squandering the overall NOW memory, we introduce a runtime heuristic where workstations recognize image computation patterns, namely sequences of events that have been fired on MDD nodes, then speculatively explore only firings conforming to these patterns.

As storing and recognizing the image computation patterns requires a nontrivial memory overhead, we develop an implicit method where MDD nodes can share the graphs encoding these patterns. This not only reduces the memory overhead but, most importantly, also provides better information regarding the evolution of each pattern, thus allows for more accurate speculation.

Our paper is organized as follows. Section 2 gives the necessary background on state-space generation, decision diagrams, Kronecker encoding, Saturation, and the distributed version of Saturation. Section 3 details our speculative image computation idea to accelerate distributed symbolic reachability analysis. Section 4 details our implicit method to dynamically encode image computation patterns, as well as two speculation heuristics that employ these encodings. Section 5 shows experimental results, while Section 6 compares our approach with related work. Finally, Section 7 draws conclusions and discusses future research directions.

2 Background

A discrete-state model is a triple $(\mathcal{S}, \mathbf{s}_{init}, \mathcal{N})$, where \mathcal{S} is the set of *potential states* of the model, $\mathbf{s}_{init} \in \mathcal{S}$ is the *initial state*, and $\mathcal{N}: \mathcal{S} \rightarrow 2^{\mathcal{S}}$ 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 $(\mathbf{i}_K, \dots, \mathbf{i}_1)$, where \mathbf{i}_k is the *local state* of submodel k , $K \geq k \geq 1$, and $\mathcal{S} = \mathcal{S}_K \times \dots \times \mathcal{S}_1$ is the cross-product of K *local state spaces*, which we assume finite. This allows us to use techniques aimed 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 $\mathcal{N}_e: \mathcal{S} \rightarrow 2^{\mathcal{S}}$ represents the next-state function associated with event e .

We seek to build the (*reachable*) *state space* $\mathcal{S}_{rch} \subseteq \mathcal{S}$, the smallest set containing \mathbf{s}_{init} and closed with respect to \mathcal{N} : $\mathcal{S}_{rch} = \{\mathbf{s}_{init}\} \cup \mathcal{N}(\mathbf{s}_{init}) \cup \mathcal{N}(\mathcal{N}(\mathbf{s}_{init})) \cup \dots = \mathcal{N}^*(\mathbf{s}_{init})$, where “ $*$ ” denotes reflexive and transitive closure and we let $\mathcal{N}(\mathcal{X}) = \bigcup_{\mathbf{i} \in \mathcal{X}} \mathcal{N}(\mathbf{i})$ for any $\mathcal{X} \subseteq \mathcal{S}$.

2.1 Symbolic encoding of \mathcal{S}_{rch}

In the sequel, we assume that the local state spaces \mathcal{S}_k are known a priori, although, they 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 $\psi_k: \mathcal{S}_k \rightarrow \{0, 1, \dots, n_k - 1\}$, 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 $\{0, 1, \dots, n_k - 1\}$, and encode any set $\mathcal{X} \subseteq \mathcal{S}$ in a (*quasi-reduced ordered*) *MDD* over \mathcal{S} . Formally, an MDD is a directed acyclic edge-labeled multi-graph where:

- Each node p belongs to a *level* $k \in \{K, \dots, 1, 0\}$, denoted $p.lvl$.

- 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 $k - 1$ or to the terminal node *Zero*; we write $p[i_k] = q$.
- No node at level $k > 0$ has all its n_k outgoing edges pointing to node *Zero*.
- Given nodes p and q at level $k > 0$, if $p[i_k] = q[i_k]$ for all $i_k \in \mathcal{S}_k$, then $p = q$, i.e., there are no *duplicates*.
- There is a single *root* node r with no incoming edges, and either r is at level K , or it is *Zero*.

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

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

For example, box 10 at the bottom of Figure 1 shows a five-node MDD with $K = 3$ encoding four states: (0,0,2), (0,1,1), (0,2,0), and (1,0,0), where edges point down and their label is written in a box in the node from where the arc originates; boxes and edges pointing to terminal node *Zero*, as well as edges pointing to terminal node *One*, as well as the terminal nodes themselves, are not shown.

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 state $(i_K, \dots, i_1) \in \mathcal{S}$, $\mathcal{N}_e(i_K, \dots, i_1) = \mathcal{N}_{K,e}(i_K) \times \dots \times \mathcal{N}_{1,e}(i_1)$. Then, 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 \mathcal{S} and \bigotimes indicates the Kronecker product of matrices. The $\mathbf{N}_{k,e}$ matrices are usually extremely sparse; for example, in the case of standard Petri nets, each row contains at most one nonzero entry.

2.3 Saturation-based fixpoint iteration strategy

In addition to efficiently representing \mathcal{N} , the Kronecker encoding allows us to recognize *event locality* [10, 22] and employ *Saturation* [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 fixpoint with respect to all \mathcal{N}_e such that $Top(e) \leq k$, i.e., $\mathcal{S}_K \times \dots \times \mathcal{S}_{k+1} \times \mathcal{B}(p) \supseteq \mathcal{N}_{\leq k}(\mathcal{S}_K \times \dots \times \mathcal{S}_{k+1} \times \mathcal{B}(p))$, where $\mathcal{N}_{\leq k} = \bigcup_{e: Top(e) \leq k} \mathcal{N}_e$. To saturate MDD node p after having saturated all its descendants, we *update it in place* so that it encodes also any state in $\mathcal{N}_{k,e} \times \dots \times \mathcal{N}_{1,e}(\mathcal{B}(p))$, for any event e with $Top(e) = k$. If this creates new MDD nodes below level k , they 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 $\mathcal{S}_{rch} = \mathcal{N}^*(\mathbf{s}_{init})$, 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 Figure 1. A 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 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} , according to events (a, b, c, d) and levels (x, y, z) , are shown in the middle of Figure 1. For each $\mathbf{N}_{k,e}$ we list only the nonzero entries, e.g.,

$$\mathbf{N}_{y,b} = \left\{ \begin{array}{l} 1 \rightarrow 0 \\ 2 \rightarrow 1 \end{array} \right\} \quad \text{means} \quad \mathbf{N}_{y,b} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

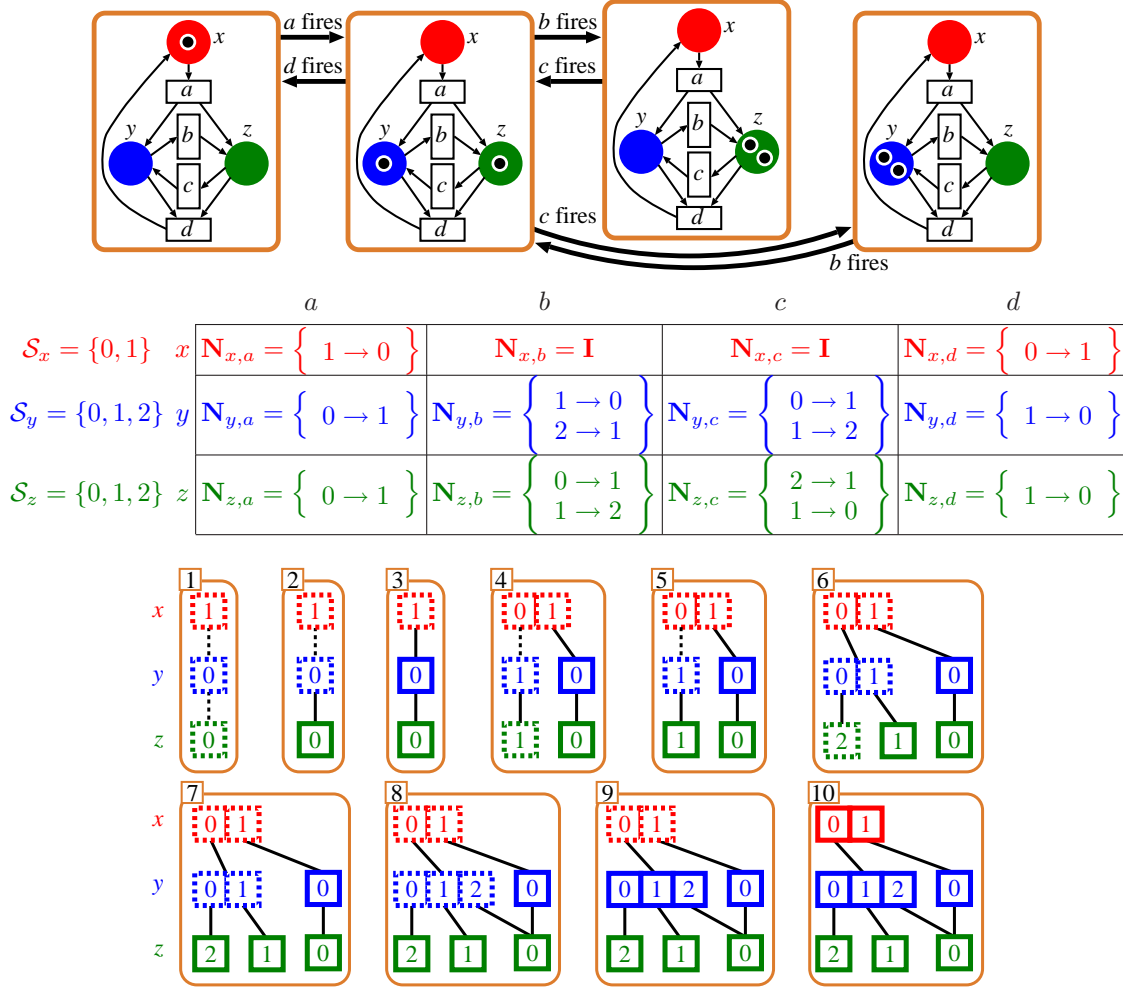


Figure 1: Reachability graph (top), \mathcal{S}_x , \mathcal{S}_y , \mathcal{S}_z , and \mathcal{N} (middle), evolution of the MDD (bottom).

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 reachability analysis of this model is shown at the bottom of Figure 1, where MDD nodes yet to be saturated are indicated with dashed lines.

- 1 **Initial configuration:** Set up the MDD encoding the initial state (1,0,0).
- 2 **Saturate node $\boxed{0}$ at level *z*:** No action is required, as there is no event *e* with $Top(e) = z$. 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 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 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 1 at level y is extended to 01 and node 2 at level z is created. This also implies that a new state, $(0,0,2)$, is discovered.
- 7 **Saturate node 2 at level z :** Again, no action is required, the node is saturated by definition.
- 8 **Saturate node 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 01 at level y is extended to 012 and node 0 at level z , which has been created and saturated previously, is referenced. This also implies that a new state, $(0,2,0)$, is discovered.
- 9 **Saturate node 012 at level y :** After exploring all possible firings, the node is saturated.
- 10 **Saturate node 01 at level x :** Since no firing can find new states, the root is saturated.

Saturation consists of many “lightweight” nested “local” fixpoint image computations and is completely different from the traditional breadth-first approach that employs a single “heavyweight” global fixpoint image computation. Results in [11, 12, 13] 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 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 presented *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 resort to virtual memory when performing 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: only pairwise communications and synchronization are needed.

3 Speculative Image Computation

Although SaturationNOW utilizes the overall NOW memory, its execution is strictly sequential; that is, it does not aim at exploiting or even just facilitating parallelism. The scheme corresponds to a sequentialization of the workstations, where most computations require a workstation to cooperate with its two neighbors, the workstations above and below it.

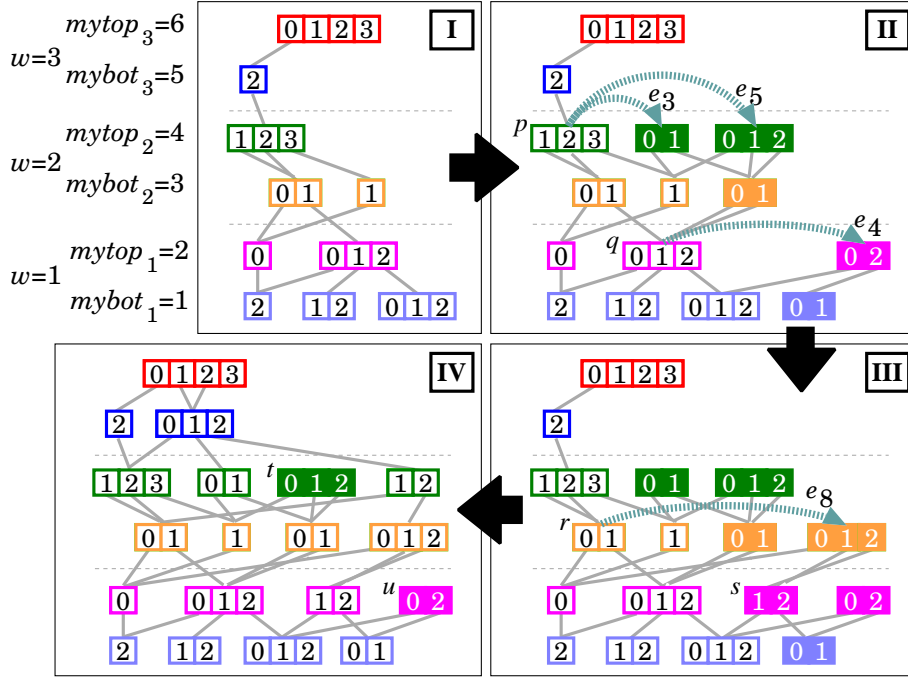


Figure 2: Speculative event firing.

We tackle this drawback, i.e., we improve the runtime of SaturationNOW, through the idea of using idle workstation time to perform speculative image computation: idle workstations speculatively fire events on decision diagram nodes, even if some of these event firings may never be needed. In a naïve approach, unrestrained speculation may cause an excessive increase in the memory consumption, to the point of being counterproductive. Thus, we, introduce a history-based approach to firing prediction that recognizes firing patterns and attempts only firings conforming to these patterns. Experiments show that our heuristic improves the runtime and has a small memory overhead.

Since event firing is an operation in Saturation (NOW) to perform image computation, we address our event-firing anticipation scheme as speculative image computation. Also, because our prediction performs at the decision diagram node level, we stress that this idea is applicable not only to SaturationNOW but also to other symbolic approaches that use decision diagrams.

3.1 Event firing speculation

We now explore the idea of using idle workstation time to fire events e with $Top(e) > k$ on saturated nodes p at level k *a priori*, in the hope to reduce the time required to saturate nodes reaching p .

As defined in Section 2.3, an MDD node p at level k is saturated if any event e with $Top(e) = k$ has been fired exhaustively on the node p . However, events e with $Top(e) = l > k \geq Bot(e)$ will still need to be fired on the node p if there is a path (i_l, \dots, i_{k+1}) from another MDD node q at level l to the MDD node p , such that the event e is locally enabled, i.e., $\mathcal{N}_{l,e}(i_l) \neq \emptyset, \dots, \mathcal{N}_{k+1,e}(i_{k+1}) \neq \emptyset$. To reduce the time required to saturate such a hypothetical node q , our speculation scheme creates the (possibly disconnected) MDD node p' corresponding to the saturation of the result of firing event e on node p , and caches the result. Later on, any firing of e on p will immediately return the result p' found in the cache. Figure 2 shows speculative event firing at work.

- (I) A runtime snapshot of an MDD distributed onto three workstations, where w_1, w_2 , and w_3 own MDD levels $\{1, 2\}, \{3, 4\}$, and $\{5, 6\}$, respectively.

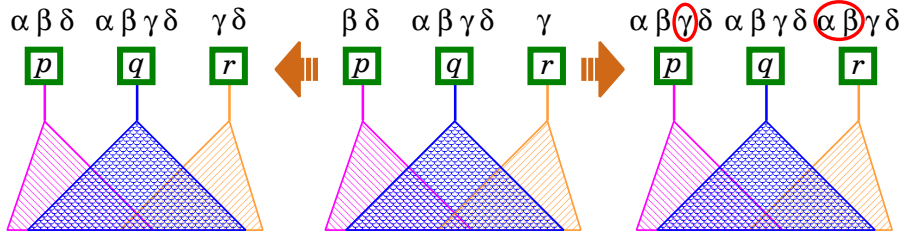


Figure 3: History-based approach to firing prediction.

- (II) Workstations 2 and 1 tentatively perform some speculative event firings individually: w_2 fires events e_3 and e_5 on node p at level 4; w_1 fires event e_4 on node q at level 2. Filled boxes represent the speculatively created MDD nodes. Such nodes are often not connected to the main MDD (in the figure, they are not reachable from the root node at level 6), but this is of course not always the case, as the result of the saturation of a speculative firing may happen to equal an existing saturated node.
- (III) Workstations 2 and 1 cooperate to perform a speculative event firing together: w_2 fires event e_8 on node r at level 3 and then requests w_1 to create node s at level 2
- (IV) Workstations 3 perform several real event firings, where the recursive invocations retrieve some of the speculatively created nodes from the cache and include them into the main MDD.

In part (IV) of Figure 2, node t at level 4 and node u at level 3 are disconnected (from the global image or the decision diagram). Some speculatively generated MDD nodes might never be used and just waste memory. Thus, our goal is minimizing the number of (eventually) disconnected MDD nodes.

We stress that the main MDD remains canonical, although there are additional disconnected nodes. Also, even when workstation w knows that event e satisfies $Top(e) > mytop_w = k \geq Bot(e) \geq mybot_w$, its decision to fire e on node p at level k can nevertheless end up requiring computation in workstation $w - 1$ below, since the result must be saturated, possibly causing work to propagate at levels below. In other words, as it is not known in advance whether the saturation of an event firing can be computed locally, consecutive idle workstations might need to perform speculative event firing together.

3.2 History-based approaches to speculative event firing

Since we do not know a priori whether event e will be fired on a node p at level k during state-space generation, the most naïve speculative event firing lets idle workstations exhaustively compute *all* possible firings starting *above* each node p of the MDD for \mathcal{S}_{rch} , i.e., it ideally would fire each event in $\mathcal{E}_{all}(p) = \{e : Top(e) > k \geq Bot(e)\}$ on p .

Obviously, this is effective only when the Kronecker encoding of \mathcal{N} is sparse, i.e., most $\mathbf{N}_{k,e}$ are identities, since, then, exhausting all possible firings over few events is relatively inexpensive in terms of time and space. However, for most models, this approach introduces too many nodes that never become connected to the MDD encoding the state space.

We now motivate a more informed prediction based on firing *patterns*. Given node p at level k , let $\mathcal{E}_{patt}(p) \subseteq \mathcal{E}_{all}(p)$ be the set of events e with $Top(e) > k$ that will be fired on p after it has been saturated. We can then partition the nodes at level k according to their patterns, i.e., nodes p and q are in the same class if and only if $\mathcal{E}_{patt}(p) = \mathcal{E}_{patt}(q)$. Unfortunately, $\mathcal{E}_{patt}(p)$ is only known *a posteriori*, but we have empirically observed that most models exhibit clear firing patterns during Saturation, i.e., most classes contain many nodes and most patterns contain several events.

Our goal is to *predict* the pattern of a given node p based on the *history* $\mathcal{E}_{hist}(p) \subseteq \mathcal{E}_{patt}(p)$ of the events fired on p so far. The key idea is that, if $\emptyset \subset \mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$, we can speculate that the events in $\{\mathcal{E}_{hist}(q) \setminus \mathcal{E}_{hist}(p)\}$ will eventually need to be fired on p as well, i.e., that $\mathcal{E}_{patt}(p) = \mathcal{E}_{patt}(q)$ at the end. Figure 3 shows an example where p , q , and r are saturated nodes at the same MDD

<pre> FirePredict (Requests : stack, Class : array) 1 while Requests ≠ ∅ do 2 (e, p) ← Pop(Requests); 3 Enqueue(e, E_hist(p)); 4 q ← Class_p.lvl[e]; 5 if E_hist(p) ⊃ E_hist(q) then 6 Class_p.lvl[e] ← p; 7 fire e on q and cache the result; 8 else if E_hist(p) ⊂ E_hist(q) then 9 foreach e' ∈ {E_hist(q) \ E_hist(p)} do 10 fire e' on p and cache the result; </pre>	
	<ul style="list-style-type: none"> • for each event firing that has been performed <ul style="list-style-type: none"> • record event firing history • find the representative node for e • E_hist(p) is a better pattern • let p become the new representative • predict event firings with the pattern

Figure 4: Firing prediction algorithm.

level. The middle of Figure 3 shows the current event firing history of these nodes at some point during runtime: $\mathcal{E}_{hist}(p) = \{\beta, \delta\}$, $\mathcal{E}_{hist}(q) = \{\alpha, \beta, \gamma, \delta\}$, and $\mathcal{E}_{hist}(r) = \{\gamma\}$. The left of Figure 3 shows the actual event firing history of these nodes after the state space is generated, i.e., their true patterns: $\mathcal{E}_{patt}(p) = \{\alpha, \beta, \delta\}$, $\mathcal{E}_{patt}(q) = \{\alpha, \beta, \gamma, \delta\}$, and $\mathcal{E}_{patt}(r) = \{\gamma, \delta\}$. Applying our history-based approach, instead, will result in the firings on the right of Figure 3: since $\mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$ and $\mathcal{E}_{hist}(r) \subset \mathcal{E}_{hist}(q)$, the workstation owning this MDD level will fire β and γ on p and α , β , and δ on q in advance, if it is idle. Thus, the useless firings of γ on p and of α and β on q will be speculatively computed (these are highlighted with circles in Figure 3).

Of course, we do not want to be too aggressive in our prediction. For example, $\mathcal{E}_{hist}(p) = \emptyset$ for all nodes p when we start keeping track of their history, i.e., after they have been saturated, thus $\mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$ holds for any node q with a non-empty history, but this is not a good reason to speculatively fire the events $\mathcal{E}_{hist}(q)$ on p . In fact, even once $\mathcal{E}_{hist}(p)$ contains some elements, it might still be that $\mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$ for several different nodes q whose histories have few common elements in addition to $\mathcal{E}_{hist}(p)$. If, for each of these nodes q , we fire each e in $\{\mathcal{E}_{hist}(q) \setminus \mathcal{E}_{hist}(p)\}$ on p , many of these predicted firings may be *useless*, i.e., they may not be actually requested. On the other hand, any prediction based on history is guaranteed to be *useful* in the rare case where the patterns of the nodes at level k are *disjoint*: i.e., if, for any two nodes p and q , either $\mathcal{E}_{patt}(p) = \mathcal{E}_{patt}(q)$ or $\mathcal{E}_{patt}(p) \cap \mathcal{E}_{patt}(q) = \emptyset$.

In the worst case, the overhead of the approach is the additional space for storing the firing history of each node and the time delay on responding to actual firing requests from above due to being busy on a firing prediction. Obviously, the pattern recognition algorithm is critical to the efficiency of this approach. Yet, a complex heuristic requiring much space and time will again decrease the improvement gained from speculative image computation.

3.3 An efficient implementation of our history-based approach

In addition to being useful, our heuristic also needs to be inexpensive in terms of memory and time overhead. Our technique, then, uses only a subset of the history and an efficient array-based method for prediction requiring $O(1)$ time per lookup and $O(K \cdot |\mathcal{E}|)$ memory overall. Each workstation w :

- Stores only the c (e.g., 10) most recent elements of \mathcal{E}_{hist} for each of its nodes.
- Maintains a list $Requests_w$ containing satisfied firing request (e, p) .
- Uses an array $Class_k$ of size $\{e : Top(e) > k \geq Bot(e)\}$ for each level k of the MDD it owns. An element $Class_k[e]$ is a pointer to a node, initially null.

Normally, workstation w is in *Saturation* mode: it computes the result of (real) firing requests (e, p) with $Top(e) > p.lvl = k$, and records (e, p) in $Requests_w$. When w becomes idle, it turns to *prediction* mode (speculative): it removes an element (e, p) from $Requests_w$, adds e to the (truncated)

history $\mathcal{E}_{hist}(p)$, and examines $Class_k(e)$. If $Class_k(e) = \text{null}$, it sets $Class_k(e)$ to p ; if $Class_k(e) = q$ and $\mathcal{E}_{hist}(p) \supset \mathcal{E}_{hist}(q)$, it sets $Class_k(e)$ to p , and speculatively fires the events in $\{\mathcal{E}_{hist}(p) \setminus \mathcal{E}_{hist}(q)\}$ on q ; if $Class_k(e) = q$ and $\mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$, it leaves $Class_k(e)$ unchanged and speculatively fires the events in $\{\mathcal{E}_{hist}(q) \setminus \mathcal{E}_{hist}(p)\}$ on p . Figure 4 shows the algorithm we have discussed. To minimize the latency for non-speculative requests, a (real) firing request from workstation $w + 1$ switches w back to *Saturation* mode, aborting any speculative event firing under way.

In other words, we use $Class_k(e)$ to predict which node has the best history among all nodes on which e has been fired so far, and use the history of this node as our speculation guide for any node on which e is subsequently fired. This heuristic may suffer from inversions: if $Class_k(e) = q$ and $\mathcal{E}_{hist}(p) \supset \mathcal{E}_{hist}(q)$ when e is fired on p , we set $Class_k(e)$ to p ; later on, further firings of q may result in $\mathcal{E}_{hist}(p) \subset \mathcal{E}_{hist}(q)$, but $Class_k(e)$ will never be set to q , since the firing of e on q is in the cache already and will never be requested again. Nevertheless, this heuristic has minimal time requirements for bookkeeping, especially in *Saturation* mode, and fast lookup times; its memory requirements are also low, since, the more workstations are idle, the faster $Requests_w$ is emptied, while $Class_k$ and the truncated history use less memory than the nodes of the MDD in practice. Section 5 shows that this heuristic can reduce runtime on large models. Furthermore, our approach can be relaxed: if we fire e on p , $Class_k(e) = q$, and $\mathcal{E}_{hist}(q) \cup \{f\} \supset \mathcal{E}_{hist}(p)$ but $f \notin \mathcal{E}_{hist}(q)$, we can still decide to speculatively fire $\{\mathcal{E}_{hist}(q) \setminus \mathcal{E}_{hist}(p)\}$ on p ; however, this aggressive approach often results in too many useless firings.

3.4 Space and time overhead of the pattern recognition approach

In terms of the space overhead, as long as the predefined value of c times the number of byte(s) needs to store an event index is close to the size of an MDD node, in the worst case, we actually pay the same size as the state-space construction for memorizing the event firing history of each node. Fortunately, our experiments show that, event firing patterns are usually few in comparison to the size of MDD nodes. Also, we can maintain the event firing history of each MDD node as a linked list to reduce memory consumption, since not every node has events fired on it after it is saturated. Moreover, each entry in $Requests_w$ is an integer pair and the number of entries in $Requests_w$ can grow and shrink at runtime. In fact, $Requests_w$ will be emptied frequently if a workstation is often idle. The size of array $Class_k$ is just $|\mathcal{E}|$ or, more precisely, $|\mathcal{E}'|$ where \mathcal{E}' is $\{e : e \in \mathcal{E} \text{ and } Top(e) > k \geq Bot(e)\}$, thus negligible compared to the memory requirements for state-space construction.

4 Graph-based image speculation

Section 3 describes our initial attempt to speedup *SaturationNOW*, which is only at times effective, and can introduce a non-trivial memory overhead. To improve the efficiency and accuracy of speculation, this section explores how to encode the evolution of firing patterns for the MDD nodes. At the same time, it also seeks to reduce the memory required to store this auxiliary information. 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.

4.1 Firing pattern graph

In our implicit encoding, firing patterns are stored in a directed acyclic graph, $G_k = (V_k, E_k)$ for every MDD level $K \geq k \geq 1$. 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 a non-empty set of MDD nodes at level k after saturating them, $\mathcal{E}_v \subseteq \mathcal{E}_{all} = \{e \in \mathcal{E} : Top(e) > k \geq Bot(e)\}$. 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 have 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 (nonempty) patterns of the current MDD nodes, thus, in practice, $|V_k| \ll 2^{|\mathcal{E}_{all}|}$.

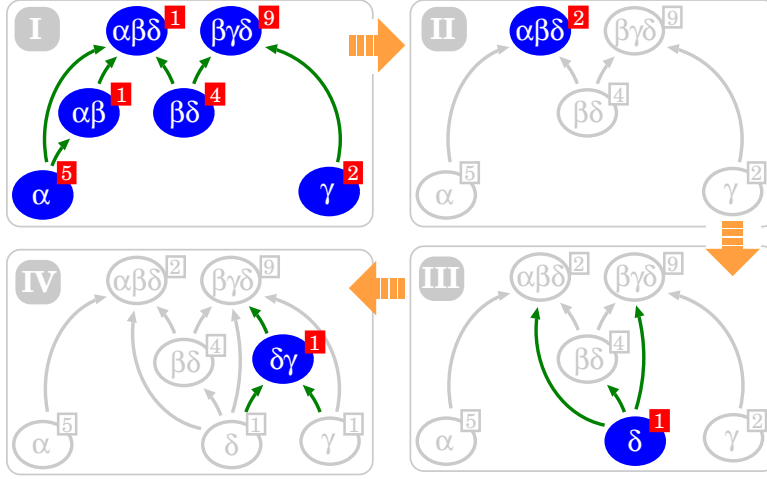


Figure 5: Updating a firing pattern graph.

In addition, for any pair of distinct PG nodes u and v in V_k , there is an arc (u, v) in E_k if and only if $\mathcal{E}_u \subset \mathcal{E}_v$. 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 v is not referenced by any MDD node, thus it is removed. The time required to update the graphs is nontrivial, but workstations perform these updates only when idle.

Figure 5 shows three examples of the pattern graph updating:

(I) Initial runtime snapshot of a firing pattern graph for an MDD level which has at least 22 nodes:

- five MDD nodes referencing the PG node representing the pattern $\{\alpha\}$,
- two MDD nodes referencing the PG node representing the pattern $\{\gamma\}$,
- one MDD node referencing the PG node representing the pattern $\{\alpha, \beta\}$,
- four MDD nodes referencing the PG node representing the pattern $\{\beta, \delta\}$,
- one MDD node referencing the PG node representing the pattern $\{\alpha, \beta, \delta\}$, and
- nine MDD nodes referencing the PG node representing the pattern $\{\beta, \gamma, \delta\}$.

(II) Update of the pattern graph after firing event δ on the only MDD node with firing pattern $\{\alpha, \beta\}$.

(III) Update after firing event δ on a newly created MDD node (not referencing any PG node).

(IV) Update after firing event δ on one of the two MDD nodes with firing pattern $\{\gamma\}$.

4.2 Pattern-graph-based image speculation

Obviously, applying our implicit method to the firing pattern encoding not only dramatically reduces the memory overhead introduced by pattern recognition, 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 a 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 , where $\{e\} = \mathcal{E}_u \setminus \mathcal{E}_v$ 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 a PG node

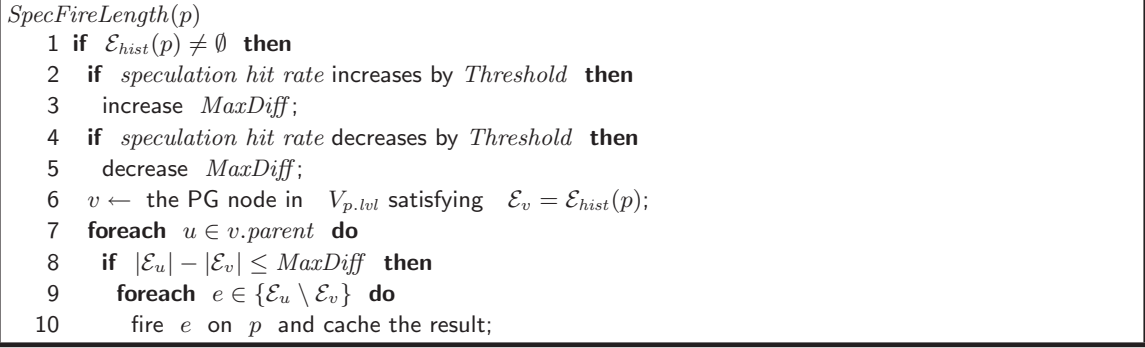


Figure 6: Algorithm to fine-tune the pattern-length-based scheme.

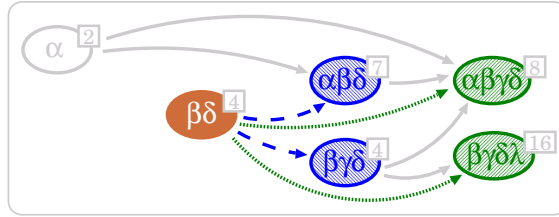


Figure 7: Pattern-length-based scheme.

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 and later requested at least once 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 by some user given threshold *Threshold*, the 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. We introduce two dynamically adjustable speculation schemes next.

4.2.1 Pattern-length-based scheme

To perform a dynamically adjustable speculation based on 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 dynamically make speculation more aggressive by increasing its value of *MaxDiff*, more conservative by reducing it. The algorithm is shown in Figure 6. The PG node v at line 6 of *SpecFireLength* exists and is unique for any given MDD node p .

For example, in Figure 7, when *MaxDiff* = 1, a workstation managing MDD node p with firing pattern $\{\beta, \delta\}$ may fire events α and γ on p (following the coarse-dashed lines), 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 having just one more element. If *MaxDiff* = 2, the workstation may in addition also fire λ on p (following the fine-dashed lines), since now also the size of pattern $\{\beta, \gamma, \delta, \lambda\}$ is within *MaxDiff* of that of the firing pattern of p . Note that pattern $\{\alpha, \beta, \gamma, \delta\}$ also qualifies, but it is already obtained as the union of patterns $\{\alpha, \beta, \delta\}$ and $\{\beta, \gamma, \delta\}$.

```

SpecFireScore(p)
1 if  $\mathcal{E}_{hist}(p) \neq \emptyset$  then
2   if speculation hit rate increases by Threshold then
3     decrease MinScore;
4   if speculation hit rate decreases by Threshold then
5     increase MinScore;
6    $v \leftarrow$  the node in  $V_{p,lvl}$  satisfying  $\mathcal{E}_v = \mathcal{E}_{hist}(p)$ ;
7   foreach  $u \in v.parent$  do
8     if  $Score(v, u) \geq MinScore$  then
9       foreach  $e \in \{\mathcal{E}_u \setminus \mathcal{E}_v\}$  do
10        fire  $e$  on  $p$  and cache the result;

```

Figure 8: Algorithm to fine-tune the weighted-score-based scheme.

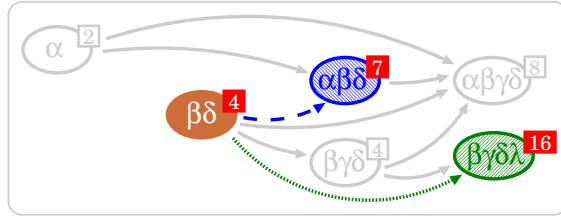


Figure 9: Weighted-score-based scheme.

4.2.2 Weighted-score-based scheme

The previous heuristic uses the reference counter only to discard patterns when no MDD node references them; that is, reference counters are only used to maintain the pattern graphs up-to-date. A finer speculation scheme could instead have workstations adjust the aggressiveness of prediction based on the value of the reference counters (higher counts indicate 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).

We thus define the *weighted score* from v to u as $Score(v, u) = \frac{u.ref}{|\mathcal{E}_u| - |\mathcal{E}_v|}$. Then, a workstation calculates the score of each PG edge (v, u) and performs the speculative firing of the events in $\{\mathcal{E}_u \setminus \mathcal{E}_v\}$ on the MDD nodes having pattern \mathcal{E}_v if $Score(v, u)$ meets or exceeds 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 $MinScore$. The algorithm is shown in Figure 8.

For example, in Figure 9, if $MinScore = 6$, a workstation managing MDD node p with firing pattern $\{\beta, \delta\}$ may fire events γ and λ on p (following the fine-dashed line), 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 in addition also fire α on p (following the coarse-dashed line), since now, considering PG node $\{\alpha, \beta, \delta\}$ with reference count 7, we have $7/(3 - 2) = 7 \geq MinScore$.

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.

5 Experimental results

We implemented both heuristic schemes in SMART^{QW} [5], the MPICH-based distributed version of our tool SMART [8], and evaluated their performance by using Saturation to generate the state space of the following models, all parameterized by a value N .

- *Flexible manufacturing system (FMS)* [22] models a manufacturing system with three machines to process three different types of parts. N is the number of each type of parts. $K = 19$, $|\mathcal{S}_k| = N + 1$ for all k except $|\mathcal{S}_{17}| = 4$, $|\mathcal{S}_{12}| = 3$, and $|\mathcal{S}_7| = 2$, $|\mathcal{E}| = 20$.
- *Slotted ring network protocol (Slot)* [25] models a protocol for local area networks. N is the number of nodes in the network. $K = N$, $|\mathcal{S}_k| = 10$ for all k , $|\mathcal{E}| = 3N$.
- *Runway safety monitor (RSM)* [26] models an avionics system monitoring T targets with S speeds on a grid represented as a $X \times Y \times Z$ grid. N is the value of Z , while $T = 2$, $S = 3$, $X = 3$, and $Y = 3$. $K = 5(T + 1)$, $|\mathcal{S}_{5+5i}| = 3$, $|\mathcal{S}_{4+5i}| = 14$, $|\mathcal{S}_{3+5i}| = 1 + X(10 + 6(S - 1))$, $|\mathcal{S}_{2+5i}| = 1 + Y(10 + 6(S - 1))$, $|\mathcal{S}_{1+5i}| = 1 + Z(10 + 6(S - 1))$, for $i = 0, \dots, T$, except $|\mathcal{S}_{4+5T}| = 7$, $|\mathcal{E}| = 49 + T(56 + (Y - 2)(31 + (X - 2)(13 + 4Z))) + 3(X - 2)(1 + YZ) + 2X + 5Y + 3Z$.
- *Aloha network protocol (Aloha)* [9] models the famous precursor to the Ethernet protocol. N is the number of nodes in the network. $K = N + 3$, $|\mathcal{S}_k| = 3$ for all k except $|\mathcal{S}_K| = 2$ and $|\mathcal{S}_{K-1}| = |\mathcal{S}_{K-2}| = N + 1$, $|\mathcal{E}| = 4N$.
- *Round robin mutex protocol (Robin)* [16] models a round robin solution to the mutual exclusion problem. N is the number of processes involved. $K = N + 1$, $|\mathcal{S}_k| = 10$ for all k except $|\mathcal{S}_1| = N + 1$, $|\mathcal{E}| = 5N$.
- *N queens placement puzzle (Queen)* models the classic problem of placing N chess queens on an $N \times N$ chessboard so that they cannot attack each other using the standard chess queen’s moves. $K = N$, $|\mathcal{S}_k| = N + 1$ for all k , $|\mathcal{E}| = N^2$.

We run our implementation on these 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 and total memory requirements using W workstations for sequential SMART (SEQ) [11] and the original SMART^{QW} (DISTR) [5], and the percentage change w.r.t. DISTR for the naïve (NAÏVE), the history-based (HIST) [6], and the pattern-length-adjusted (LENGTH) or weighted-score-adjusted (SCORE) speculative firing predictions [7]; “ d ” means that dynamic memory load balancing is triggered, “ s ” means that, in addition, memory swapping occurs. The last four columns show the overall hit rate of the speculative caches for the NAÏVE, HIST, LENGTH, and SCORE approaches. For the LENGTH heuristic, we initialize *MaxDiff* to 2 and increase/decrease it by 2 whenever the speculation hit rate increases/decreases by 5%. For the SCORE heuristic, we initialize *MinScore* as 100 and divide/multiply it by 2 whenever the speculation hit rate increases/decreases by 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. Furthermore, the overall speculative hit of LENGTH or SCORE is always higher 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 are 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 runway safety monitoring 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.

The experiments on the Aloha network protocol show an example where HIST, LENGTH, or SCORE can still outperform DISTR even though the overall hit-rate of their speculative caches is

W	Time (sec \vee +/- %)					Memory (MB \vee +/- %)					Spec. Cache (%)				
	DISTR	NAIVE	HIST	LENGTH	SCORE	DISTR	NAIVE	HIST	LENGTH	SCORE	NAIVE	HIST	LENGTH	SCORE	
FMS	$N=300$	$ \mathcal{S}_{rch} =3.64 \cdot 10^{27}$					SEQ completes in 55 sec using 241MB								
2	79	-8	-8	-8	-11	243	+12	+24	+3	+5	16	13	24	27	
4	91	$d+67$	-9	-13	-20	243	+102	+30	+11	+14	< 1	15	23	21	
8	260	-	-30	-44	-50	243	-	+42	+16	+22	-	9	19	15	
FMS	$N=450$	$ \mathcal{S}_{rch} =6.9 \cdot 10^{29}$					SEQ cannot complete in 5 hrs using 512MB								
2	$s257$	$s+12$	$s-14$	$s-10$	$s-14$	826	+16	+5	+4	+4	1	19	20	20	
4	$d311$	> 5hrs	$d-18$	$d-29$	$d-30$	826	-	+33	+9	+17	-	6	18	13	
8	959	> 5hrs	-25	-34	-38	826	-	+61	+24	+39	-	3	17	12	
SLOT	$N=200$	$ \mathcal{S}_{rch} =8.38 \cdot 10^{211}$					SEQ completes in 108 sec using 284MB								
2	119	-24	-13	-5	-8	286	+3	+45	+12	+14	22	13	17	16	
4	139	-27	-15	-6	-9	286	+11	+51	+17	+26	16	9	16	14	
8	182	-32	-24	-14	-20	286	+129	+62	+20	+29	< 1	9	15	11	
SLOT	$N=300$	$ \mathcal{S}_{rch} =8.38 \cdot 10^{211}$					SEQ cannot complete in 5 hrs using 512MB								
2	$s552$	$s+5$	$s-5$	$s-10$	$s-7$	962	+25	+11	+6	+10	2	9	13	10	
4	$d490$	> 5hrs	$d-16$	$d-18$	$d-14$	962	-	+34	+13	+19	-	7	12	7	
8	564	> 5hrs	-39	-24	-30	962	-	+50	+21	+29	-	5	10	8	
RSM	$Z=2$	$ \mathcal{S}_{rch} =1.51 \cdot 10^{15}$					SEQ completes in 236 sec using 314MB								
2	731	> 10hrs	-2	-3	-5	332	-	+39	+4	+9	-	5	10	7	
4	938	> 10hrs	-8	-16	-18	332	-	+88	+20	+28	-	4	11	9	
8	1480	> 10hrs	-22	-27	-31	332	-	+128	+67	+90	-	2	9	8	
RSM	$Z=3$	$ \mathcal{S}_{rch} =5.07 \cdot 10^{15}$					SEQ cannot complete in 10 hrs using 512MB								
2	$s11280$	> 10hrs	$s-1$	$s-2$	$s-3$	962	-	+10	+3	+4	-	5	9	6	
4	$d9762$	> 10hrs	$d-15$	$d-19$	$d-26$	962	-	+31	+4	+13	-	3	9	6	
8	$d14101$	> 10hrs	$d-17$	$d-29$	$d-31$	962	-	+58	+8	+35	-	3	8	5	
Aloha	$N=120$	$ \mathcal{S}_{rch} =8.1 \cdot 10^{37}$					SEQ completes in 292 sec using 202MB								
2	320	-5	-3	-2	-3	207	+55	+20	+11	+14	< 1	4	4	4	
4	443	-12	-8	-3	-9	207	+99	+51	+18	+23	< 1	5	4	5	
8	626	-21	-9	-5	-11	207	+217	+92	+47	+61	< 1	3	5	6	
Aloha	$N=200$	$ \mathcal{S}_{rch} =1.62 \cdot 10^{62}$					SEQ cannot complete in 5 hrs using 512MB								
2	$d4886$	$d+24$	$d-2$	$d-1$	$d-2$	855	+94	+50	+24	+29	< 1	1	3	2	
4	6902	$d+254$	$d-3$	-5	-6	855	+212	+104	+37	+52	< 1	< 1	2	2	
8	9430	> 10hrs	$d-7$	-8	-13	855	-	+126	+50	+68	-	< 1	2	5	
Robin	$N=800$	$ \mathcal{S}_{rch} =1.2 \cdot 10^{196}$					SEQ completes in 27 sec using 290MB								
2	29	+37	+6	+0	+0	293	+110	+85	+21	+27	< 1	< 1	< 1	< 1	
4	36	+33	+8	+0	+0	293	+348	+109	+28	+37	< 1	< 1	< 1	< 1	
8	51	+33	+5	+0	+0	293	+807	+148	+36	+49	< 1	< 1	< 1	< 1	
Robin	$N=1100$	$ \mathcal{S}_{rch} =3.36 \cdot 10^{334}$					SEQ cannot complete in 5 hrs using 512MB								
2	$d65$	$s+62$	$s+18$	$d+5$	$d+9$	794	+46	+6	+2	+3	< 1	< 1	< 1	< 1	
4	47	$s+131$	$d+10$	+2	+2	794	+119	+38	+3	+5	< 1	< 1	< 1	< 1	
8	56	$d+164$	+7	+2	+2	794	+299	+50	+9	+10	< 1	< 1	< 1	< 1	
Queen	$N=13$	$ \mathcal{S}_{rch} =4.67 \cdot 10^6$					SEQ completes in 597 sec using 193MB								
2	871	+175	+17	+0	+1	194	+291	+93	+12	+15	< 1	< 1	1	1	
4	1399	> 10hrs	+48	+0	+3	194	-	+166	+13	+21	-	< 1	< 1	< 1	
8	2204	> 10hrs	$d+52$	+2	+4	194	-	+195	+19	+24	-	< 1	< 1	< 1	
Queen	$N=14$	$ \mathcal{S}_{rch} =2.73 \cdot 10^7$					SEQ cannot complete in 10 hrs using 512MB								
2	$d14175$	> 10hrs	> 10hrs	$d+3$	$d+3$	907	-	-	+4	+5	-	-	< 1	< 1	
4	17206	> 10hrs	> 10hrs	+4	+6	907	-	-	+7	+11	-	-	< 1	< 1	
8	30074	> 10hrs	> 10hrs	+3	+6	907	-	-	+21	+30	-	-	< 1	< 1	

Table 1: Experimental results.

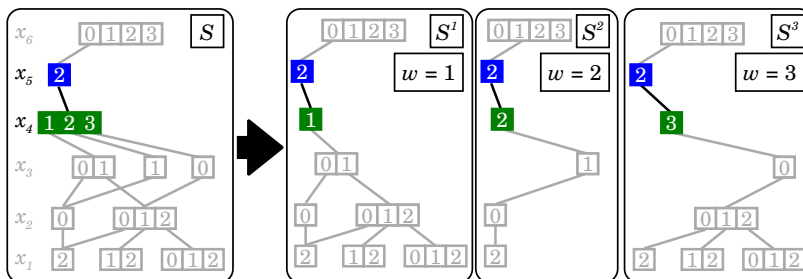


Figure 10: Vertical image slicing scheme — slicing variables.

relatively low. Furthermore, both LENGTH and SCORE can contain memory consumption without offsetting the speedup gained from speculation.

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 experiments on the last model, the N queen placement puzzle, show a worst-case example not only for our speculative schemes but for our horizontal image slicing approach. Because the nature of this model and its relatively small K , communication becomes the bottleneck (especially when W becomes larger) and none of those speculative schemes can help. In detail, many computations frequently traverse all levels of the decision diagram, K through 1, so most operations require abundant message passing. Yet, again, both LENGTH and SCORE restrain the distributed computation from performing too many useless speculations.

6 Related work

Most parallel or distributed work on symbolic reachability analysis employs a *vertical slicing* scheme to parallelize BDD manipulations, where image computation jobs are spawned and queued on the individual workstations of the NOW [19, 21, 29], allowing the algorithm to overlap image computation. Figure 10 shows how an MDDs can be decomposed into three portions where x_5 and x_4 are selected as slicing variables (the corresponding MDD nodes are colored in either blue or green). After performing image slicing, three paths originally going through x_5 and x_4 are now owned by w_1 , w_2 , and w_3 respectively:

- workstation w_1 owns any states satisfying $x_5 = 2$ and $x_4 = 1$,
- workstation w_2 owns any states satisfying $x_5 = 2$ and $x_4 = 2$,
- workstation w_3 owns any states satisfying $x_5 = 2$ and $x_4 = 3$,

Thus, the union of the sets of states encoded by the MDDs at each workstation (the three rectangular boxes shown at the right of Figure 10) equals to the set of states encoded by the original *global* image (the single rectangular box shown at the left of Figure 10): $S = S^1 \cup S^2 \cup S^3$. Also, it is (temporarily) true that each workstation *exclusively* owns a subset of states: $(S^1 \cap S^2 = \emptyset) \wedge (S^2 \cap S^3 = \emptyset) \wedge (S^1 \cap S^3 = \emptyset)$. However, this property is guaranteed to hold only immediately after performing the partition. As each workstation continues the fixpoint computation on its own set of states, different workstations may discover some of the same states. That is why this mechanism usually requires a master workstation to make sure that slave workstations do not frequently work on the same set of

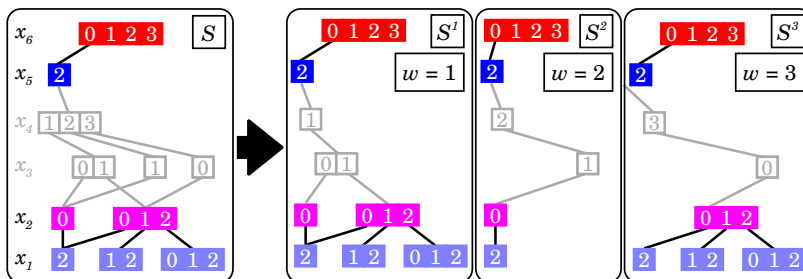


Figure 11: Vertical image slicing scheme — repeated MDD nodes.

states, by altering the list of slicing variables. Some degree of synchronization is required to reduce the impact of redundant image computation, and this may reduce the scalability of this approach.

In addition, even though it is true that, right after image slicing, the W workstations store disjoint sets of *states*, their MDDs could still store many identical MDD *nodes*. In Figure 11, all MDD nodes at levels corresponding to variables x_6 , x_5 , x_2 , and x_1 exist in multiple workstations: a partitioned state space does not imply a partitioned decision diagram, i.e., absence of node duplication. If the slicing choice is poor, many duplicate nodes may exist. In fact, it is generally agreed that finding a good slicing is not trivial [23].

To achieve scalability, researchers have recently focused on workload distribution over a NOW. [18] suggests employing a host processor to manage the job queue for load-balance purposes and to reduce redundancy in image computation, by slicing according to boolean functions that use an optimal choice of variables and minimize the peak number of decision diagram nodes required by any single workstation, thus the maximum workload.

To overcome the synchronization drawback, Grumberg et al. [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 the workload.

However, the idea of exploiting a vertical slicing approach for parallel symbolic reachability analysis might negate some important advantages of symbolic encodings: the sharing of partial state encodings and the caching of results. In other words, the MDD vertical slicing approach for symbolic reachability analysis works best when the set of states encoded by the MDD does not benefit much from symbolic encoding techniques as opposed to explicit ones. In other words, the best case for vertical slicing is when the MDD is a tree, since it is then easy to make a clean cut with no repeated nodes; however, this is also the worst case for decision diagrams, since no MDD node shares isomorphic subgraphs. The left of Figure 12 shows an MDD where exactly this situation arises: a 16-node MDD encoding only three (global) states.

Just like the redundant work introduced by vertical slicing, our approach (horizontal slicing plus image speculation) introduces useless work. More precisely, even though the MDD remains canonical, additional disconnected MDD nodes can be generated. Certainly, the vertical slicing approach can reorder the MDD variables to improve the node distribution, but the variable reordering operation is expensive and requires heavy synchronization. Instead, in our approach, each workstation can clean up disconnected MDD nodes at runtime without requiring any synchronization. Thus, our horizontal slicing scheme does not hurt scalability. Yet, there is a different limitation in our approach: the number of workstations used for distributed symbolic reachability analysis cannot exceed the number of MDD levels. To remove this limitation, we must allow multiple workstations to manipulate the same decision diagram level. Such an approach would require some (local) synchronization among the workstations cooperating on a given level, and needs to be investigated further. In addition, if the nature of some model requires most image computation to perform global traversals, as in the Queen model discussed in Section 5, communication becomes a serious bottleneck for our approach.

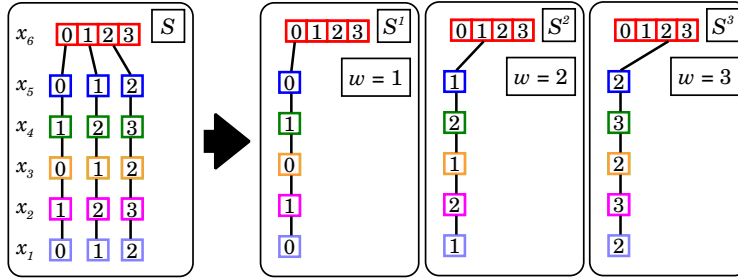


Figure 12: Best case for the vertical slicing scheme, worst case for using MDDs.

To summarize, vertical slicing approaches are best when there is some balanced way to distribute the image computation jobs, while horizontal slicing approaches combined with our image speculation heuristics are best when there are many state variables compared to the number of workstations and many image computation jobs can be performed somewhat locally; in fact, this is the type of models Saturation targets, globally-asynchronous locally-synchronous discrete-state systems. Even though neither approach has yet achieved a clear speedup with respect to the best sequential implementation, both open up new possibilities for speeding up symbolic reachability analysis.

7 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.

8 Acknowledgments

We wish to thank Radu Siminiceanu (National Institute of Aerospace) for suggesting the initial idea of speculative Saturation, Christian Shelton (University of California, Riverside) for the fruitful discussions on history-based heuristics, and the referees for their helpful suggestion.

References

- [1] Alexander Bell and Boudewijn R. Haverkort. Sequential and distributed model checking of Petri nets. *Software Tools for Technology Transfer*, 7(1):43–60, 2005.
- [2] Randy E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 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 Giuliana Franceschinis, Joost-Pieter Katoen, and Murray Woodside, editors, *Proc. Quantitative Evaluation of Systems (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 Arnold L. Rosenberg, editor, *Proc. International Parallel & Distributed Processing Symposium (IPDPS)*, Rhodes, Greece, April 2006. IEEE Comp. Soc. Press. (electronic proceeding).
- [8] Gianfranco Ciardo, Robert L. Jones, Andrew S. Miner, and Radu Siminiceanu. Logical and stochastic modeling with SMART. *Perf. Eval.*, 63:578–608, 2006.
- [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 Hatchiff, 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] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
- [15] Malay K. Ganai, Aarti Gupta, Zijiang Yang, and Pranav Ashar. Efficient distributed sat and sat-based distributed bounded model checking. In Daniel Geist and Enrico Tronci, editors, *Proc. Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of LNCS, pages 207–221, L'Aquila, Italy, October 2003. Springer-Verlag.
- [16] Susanne Graf, Bernhard Steffen, and Gerald Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Journal of Formal Aspects of Computing*, 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 Dominique Borrione and Wolfgang Paul, editors, *Proc. Correct Hardware Design and Verification Methods (CHARME)*, volume 3725 of *LNCS*, pages 129–145, Saarbrücken, Germany, October 2005. Springer-Verlag.
- [18] Orna Grumberg, Tamir Heyman, and Assaf Schuster. A work-efficient distributed algorithm for reachability analysis. In Jr. Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification (CAV)*, pages 54–66, Boulder, CO, USA, July 2003.
- [19] 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.
- [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 Ganesh Gopalakrishnan and Phillip J. Windley, editors, *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, volume 1522 of *LNCS*, pages 501–507, Palo Alto, California, USA, November 1998. Springer-Verlag.
- [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] Amit Narayan, Adrian J. Isles, Jawahar Jain, Masahiro Fujita, and Alberto L. Sangiovanni-Vincentelli. Reachability analysis using partitioned-ROBDDs. In Hiroto Yasuura, editor, *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 388–393, San Jose, CA, USA, November 1997. ACM and IEEE Computer Society Press.
- [24] David Nicol and Gianfranco Ciardo. Automated parallelization of discrete state-space generation. *J. Par. and Distr. Comp.*, 47:153–167, 1997.
- [25] Enric Pastor, Oriol Roig, Jordi Cortadella, and Rosa M. Badia. Petri net analysis using boolean manipulation. In Robert Valette, editor, *Proc. International Conference on Applications and Theory of Petri Nets (ICATPN)*, LNCS 815, pages 416–435, Zaragoza, Spain, June 1994. Springer-Verlag.
- [26] Radu Siminiceanu and Gianfranco Ciardo. Formal verification of the NASA Runway Safety Monitor. In Michael Huth, editor, *Proc. International Workshop on Automated Verification of Critical Systems (AVoCS)*, volume 128(6) of *ENTCS*, pages 179–194, London, UK, September 2004. Elsevier.
- [27] Ulrich Stern and David L. Dill. Parallelizing the mur ϕ verifier. In Orna Grumberg, editor, *Proc. International Conference on Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 256–278, Haifa, Israel, June 1997. Springer-Verlag.
- [28] Anthony L. Stornetta. Implementation of an Efficient Parallel BDD Package. Master's thesis, University of California, Santa Barbara, 1995.
- [29] Anthony L. Stornetta and Forrest Brewer. Implementation of an efficient parallel BDD package. In *Proc. Design Automation Conference (DAC)*, pages 641–644, Las Vegas, NV, June 1996. ACM Press.