A fine-grained fullness-guided chaining heuristic for symbolic reachability analysis*

Ming-Ying Chung, Gianfranco Ciardo, and Andy Jinqing Yu

Department of Computer Science and Engineering
University of California, Riverside
{chung, ciardo, jqyu}@cs.ucr.edu

Abstract. Chaining can reduce the number of iterations required for symbolic state-space generation and model-checking, especially in Petri nets and similar asynchronous systems, but requires considerable insight and is limited to a static ordering of the events in the high-level model. We introduce a two-step approach that is instead fine-grained and dynamically applied to the decision diagrams nodes. The first step, based on a precedence relation, is guaranteed to improve convergence, while the second one, based on a notion of node fullness, is heuristic. We apply our approach to traditional breadth-first and saturation state-space generation, and show that it is effective in both cases.

1 Introduction

BDD-based symbolic model checking [17] is one of the most successful techniques to verify industrial hardware and embedded software systems, and symbolic reachability analysis is a fundamental step in symbolic model checking. It is well-known that the peak number of BDD nodes is often much larger than the final number of BDD nodes for symbolic reachability analysis. In this paper, we propose a new *chaining* technique to reduce this peak number.

For asynchronous concurrent systems, such as distributed software, network protocols, and various classes of Petri nets, *chaining* [22] can reduce the peak memory usage and speed-up symbolic state-space generation by exploring events in a particularly favorable order. Chaining is normally applied as a modification of a strict breadth-first search (BFS), but it is also one of the factors behind the efficiency of the *saturation* algorithm [6]. As introduced, however, chaining is limited to finding a good order in which to apply the high-level model events during the symbolic iterations.

In this paper, we propose a general and effective heuristic that uses a partial-order relation and the concept of decision diagram node *fullness* to guide the chaining order, independent of the high-level formalism used to model the system. Our definition of node fullness is related to, but different from, the BDD node *density* defined in [20]. A detailed comparison can be found in Sect. 6.

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

Sect. 2 gives background on structured models, decision diagrams, BFS-based and saturation-based symbolic state-space generation, and chaining. Sect. 3 details our main contribution, where a fine-grained chaining is applied dynamically using the current structure of the decision diagram, rather than the model events. Sect. 4 describes the modified symbolic state-space generation algorithms incorporating our heuristic and gives implementation details. Sect. 5 provides numerical results on a suite of models showing that our heuristic reduces the runtime and memory requirements of both BFS-based and saturation-based algorithms. Sect. 6 compares the newly proposed chaining heuristics with some related work. Finally, Sect. 7 concludes with directions for future research.

2 Preliminaries

We consider a discrete-state model $(\widehat{\mathcal{S}}, \mathcal{S}^{init}, \mathcal{R})$, where $\widehat{\mathcal{S}}$ is a finite set of states, $\mathcal{S}^{init} \subseteq \widehat{\mathcal{S}}$ are the initial states, and $\mathcal{R} \subseteq \widehat{\mathcal{S}} \times \widehat{\mathcal{S}}$ is a transition relation. We assume the (*global*) model state to be a tuple of K *local state* variables, $(x_K, ..., x_1)$, where, for $K \geq l \geq 1$, $x_l \in \mathcal{S}_l = \{0, 1, ..., n_l - 1\}$, with $n_l > 0$, is the the l^{th} *local* state variable. Thus, $\widehat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$ and we write $\mathcal{R}(\mathbf{i}[K], ..., \mathbf{i}[1], \mathbf{j}[K], ..., \mathbf{j}[1])$, or $\mathcal{R}(\mathbf{i}, \mathbf{j})$, if the model can move from *current state* $\mathbf{i}$ to *next state* $\mathbf{j}$ in one step.

2.1 Symbolic encoding of state space and transition relation

State-space generation consists of building the smallest set of states $\mathcal{S} \subseteq \widehat{\mathcal{S}}$ satisfying $\mathcal{S} \supseteq \mathcal{S}^{init}$ and $\mathcal{S} \supseteq \mathit{Img}(\mathcal{S}, \mathcal{R})$, where the *image computation* function gives the set of successor states: $\mathit{Img}(\mathcal{S}, \mathcal{R}) = \{\mathbf{j} : \exists \mathbf{i} \in \mathcal{S}, \mathcal{R}(\mathbf{i}, \mathbf{j})\}$. Most symbolic approaches to store the state space encode x_l using $\lceil \log n_l \rceil$ boolean variables and a set of states $\mathcal{Z}$ using a BDD with $\sum_{K \geq l \geq 1} \lceil \log n_l \rceil$ levels.

We prefer to discuss our approach in terms of *ordered multi-way decision diagrams* (MDDs) [14], where each variable x_l is directly encoded in a single level, using a node with n_l outgoing edges. MDDs can be implemented directly, the approach taken in our tool SMART [3], or as an interface to BDDs [25].

Definition 1 An MDD over $\widehat{\mathcal{S}}$ is an acyclic edge-labeled multi-graph where:
- Each node p belongs to a *level* in $\{K, ..., 1, 0\}$, denoted $p.lvl$.
- There is a single *root* node $r^\star$.
- Level 0 can contain only the *terminal* nodes, $\mathbf{0}$ and $\mathbf{1}$.
- A node p at level $l > 0$ has n_l outgoing edges, labeled from 0 to $n_l - 1$. The edge labeled by $i \in \mathcal{S}_l$ points to node q, with $p.lvl > q.lvl$; we write $p[i] = q$.

Then, to ensure canonicity, *duplicate* nodes are forbidden:
- Given nodes p and q at level l, if $p[i] = q[i]$ for all $i \in \mathcal{S}_l$, then $p = q$,

and we must use either the *fully-reduced* rule [1] that forbids *redundant* nodes:
- No node p at level l can exist such that, $p[i] = q$ for all $i \in \mathcal{S}_l$,

or the *quasi-reduced* rule [15] that restricts arcs spanning multiple levels:
- The root is at level K.
- Given a node p at level l, $p[i].lvl$ is either $l - 1$ or 0, for all $i \in \mathcal{S}_l$. □

Definition 2 The set encoded by MDD node p at level k w.r.t. level $l \geq k$ is $\mathcal{B}(l, p) = \mathcal{S}_l \times \cdots \times \mathcal{S}_{k+1} \times \left(\bigcup_{i \in \mathcal{S}_k} \{i\} \times \mathcal{B}(k-1, p[i]) \right)$, where $\forall \mathcal{X} \subseteq \mathcal{S}_l \times \cdots \times \mathcal{S}_1$, $\mathcal{X} \times \mathcal{B}(0, \mathbf{0}) = \emptyset$ and $\mathcal{X} \times \mathcal{B}(0, \mathbf{1}) = \mathcal{X}$. If $l = k$, we write $\mathcal{B}(p)$ instead of $\mathcal{B}(k, p)$. □

MDDs vs. BDDs We use MDDs to implicitly encode the state space $\mathcal{S}$ and transition relation $\mathcal{R}$, instead of using $\lceil \log_2 \mathcal{S}_l \rceil$ bits for the local state variable x_l, and encoding $\mathcal{S}$ and $\mathcal{R}$ with BDDs. Compared with BDDs, MDDs have the disadvantage of resulting in larger and less shareable nodes when the variable domains $\mathcal{S}_l$ are very large (which is however not the case in our applications). On the other hand, MDDs have also advantages. First, many real-world models (e.g., non-safe Petri nets and software protocols) have variable domains with a priori unknown or very large upper bounds. These bounds must then be discovered "on the fly" during the symbolic iterations [10], and MDDs are preferable to BDDs when using this approach, due to the ease with which MDD nodes and variable domains can be extended. A second advantage, related to the present paper, is that our chaining heuristics applied to the MDD state variables more closely reflect structural information of the model behavior, which is instead spread on multiple levels in a BDD.

Most symbolic model checkers, e.g., NuSMV [11], generate the state space with BFS iterations, each consisting of an image computation. Set $\mathcal{X}^{[0]}$ is initialized to $\mathcal{S}^{init}$ and, after the d^{th} iteration, set $\mathcal{X}^{[d]}$ contains all the states at distance up to d from $\mathcal{S}^{init}$. When using MDDs, $\mathcal{X}^{[d]}$ is encoded as a K-level MDD and $\mathcal{R}$ as a $2K$-level MDD whose current and next state variables are normally interleaved for efficiency. We use this order too. Also, the transition relation is often conjunctively partitioned into a set of *conjuncts* or disjunctively partitioned into a set of *disjuncts* [2], stored as a set of MDDs that can share nodes instead of a single monolithic MDD. Heuristically, these partitionings have been shown to be effective for both synchronous and asynchronous systems.

In the following, we use the data-types mdd and mdd2 to indicate quasi-reduced MDDs encodings sets and relations, respectively, and, for readability, we let $\mathcal{X}$ indicate both a set and the root of the MDD encoding that set.

2.2 Disjunctive partition of $\mathcal{R}$ and chaining

Both asynchronous and synchronous behaviors may be present in many systems. We focus on *globally-asynchronous locally-synchronous* behaviors. Thus, we assume the high-level model specifies a set of asynchronous events $\mathcal{E}$, where each event $\alpha \in \mathcal{E}$ can be further specified as a set of small synchronous components.

For example, a guarded command language model specifies a set of commands of the form "$guard \rightarrow assignment_1 \parallel assignment_2 \parallel \cdots \parallel assignment_m$", where, whenever the boolean predicate *guard* evaluates to *true*, the m parallel atomic assignments can be executed concurrently (synchronously). Each command is an asynchronous events in the system and for each command, each assignment of the parallel assignments is a synchronous component of the event. Similarly, for Petri net models, the set of transitions in the net are the asynchronous events in the system and the firing of a transition synchronously updates all the places connected to it. We use extended Petri nets as the input formalism in SMART [3].

We encode the transition relation as $\mathcal{R} \equiv \bigvee_{\alpha \in \mathcal{E}} \mathcal{D}_\alpha$, where each disjunct $\mathcal{D}_\alpha$ corresponds to an asynchronous event α. Each $\mathcal{D}_\alpha$ is further conjunctively partitioned, where each conjunct $\mathcal{C}_{\alpha,l}$ represents a synchronous component of α, thus we can write $\mathcal{R} \equiv \bigvee_{\alpha \in \mathcal{E}} \mathcal{D}_\alpha \equiv \bigvee_{\alpha \in \mathcal{E}} (\bigwedge_l \mathcal{C}_{\alpha,l})$.

Chaining [22] was introduced to speed up symbolic BFS-based state-space generation and similar symbolic fixed-point computations for asynchronous systems. The idea of chaining is based on the observation that the number of symbolic iterations might be reduced if the effect of exploring various events on a given set of states is compounded sequentially. More precisely, in a strict BFS symbolic iteration, the set $\mathcal{X}^{[d]}$ of states at distance up to d from $\mathcal{S}^{init}$ is built in exactly d iterations starting from $\mathcal{X}^{[0]} = \mathcal{S}^{init}$. The d^{th} iteration applies the monolithic $\mathcal{R}$, or each disjunct $\mathcal{D}_\alpha$ corresponding to a distinct event α, to the set of states $\mathcal{X}^{[d-1]}$ reachable in up to $d-1$ steps. However, when we have the individual disjuncts $\mathcal{D}_\alpha$ at our disposal, we can instead apply them in an incremental fashion. If the event set is $\mathcal{E} = \{\alpha_1, \alpha_2, ..., \alpha_m\}$ and $\mathcal{Y}^{[d-1]}$ is the set of states found at the end of iteration d–1 with chaining, this approach computes

- $\mathcal{Y}^{[d;1]} \leftarrow \mathcal{Y}^{[d-1]} \cup \mathit{Img}(\mathcal{Y}^{[d-1]}, \mathcal{D}_{\alpha_1})$,
- $\mathcal{Y}^{[d;2]} \leftarrow \mathcal{Y}^{[d;1]} \cup \mathit{Img}(\mathcal{Y}^{[d;1]}, \mathcal{D}_{\alpha_2})$, and so on, until
- $\mathcal{Y}^{[d;m]} \leftarrow \mathcal{Y}^{[d;m-1]} \cup \mathit{Img}(\mathcal{Y}^{[d;m-1]}, \mathcal{D}_{\alpha_m})$, which becomes our next $\mathcal{Y}^{[d]}$.

Clearly, this will not discover states in strict BFS order, but it guarantees that the set of states discovered with chaining at the d^{th} (outer) iteration is at least as large as those discovered in strict BFS order: $\mathcal{Y}^{[d]} \supseteq \mathcal{X}^{[d]}$. Thus, chaining may reach the fixed point, i.e., compute $\mathcal{S}$, in fewer iterations. Of course, the efficiency of state-space generation is determined not just by the *number* of symbolic iterations, but also by their *cost*, which is strictly related to the number of nodes in the decision diagrams being manipulated. While chaining could in principle result in larger intermediate decision diagrams and even slow down the computation, in practice. the opposite is often true: chaining has been shown to be quite effective in many asynchronous models.

To maximize the effectiveness of chaining, however, we must employ some heuristic to decide the order in which events should be explored. For example, [22] uses a topological sort on the gates of a circuit modeled as a Petri net. The intuition is that, if firing Petri net transition α adds tokens to a place that is input to another Petri net transition β, then the corresponding disjunct $\mathcal{D}_\alpha$ should be applied before $\mathcal{D}_\beta$ within each iteration, as this increases the chances that β will be enabled, thus discover more states, in the larger set of states obtained by considering also the effect of α. If the Petri net has cycles, they need to be "opened" by arbitrarily picking one transition in the cycle to fire first, and then firing the remaining transitions in order.

A different, not model-based, chaining order heuristic can also be employed. Given an event α, define $\mathcal{V}_M(\alpha) = \{x_l : \exists \mathbf{i}, \mathbf{j} \in \widehat{\mathcal{S}}, \mathcal{D}_\alpha(\mathbf{i},\mathbf{j}) \wedge \mathbf{i}[l] \neq \mathbf{j}[l]\}$, and $\mathcal{V}_D(\alpha) = \{x_l : \exists \mathbf{i}, \mathbf{i}' \in \widehat{\mathcal{S}}, \forall k \neq l, \mathbf{i}[k] = \mathbf{i}'[k] \wedge \exists \mathbf{j} \in \widehat{\mathcal{S}}, \mathcal{D}_\alpha(\mathbf{i},\mathbf{j}) \wedge \nexists \mathbf{j}' \in \widehat{\mathcal{S}}, \mathcal{D}_\alpha(\mathbf{i}',\mathbf{j}')\}$, the variables that can be modified by α, or can disable, α, respectively. Letting

$$\mathit{Top}(\alpha) = \max\{l : x_l \in \mathcal{V}_M(\alpha) \cup \mathcal{V}_D(\alpha)\}, \mathit{Bot}(\alpha) = \min\{l : x_l \in \mathcal{V}_M(\alpha) \cup \mathcal{V}_D(\alpha)\},$$

we can then partition the events according to the value of *Top*, by defining the subsets of events $\mathcal{E}_l = \{\alpha : \mathit{Top}(\alpha) = l\}$, for $K \geq l \geq 1$ (some of these sets can be empty, of course). In [7] we observed that a chaining order that applies these

```
mdd $BfsChaining$( )
1 $\mathcal{S} \leftarrow \mathcal{S}^{init}$;
2 repeat
3     for $l = 1$ to $K$ do
4         foreach $\alpha \in \mathcal{E}_l$ do
5             $\mathcal{S} \leftarrow Union(\mathcal{S}, Image(\mathcal{S}, \mathcal{D}_\alpha))$
6 until $\mathcal{S}$ does not change;
7 return $\mathcal{S}$;
```

Fig. 1: Symbolic BFS-based state-space generation with chaining.

```
void $Saturation$( )
1 for $l = 1$ to $K$ do
2     foreach node $p$ at level $l$ in the MDD $\mathcal{S}^{init}$ do
3         $Saturate(p)$;                • Bottom-up sub-fixpoint computation by DD node
```

```
void $Saturate$( mdd $p$ )
1 $l \leftarrow p.lvl$;
2 repeat
3     choose $\alpha \in \mathcal{E}_l, i \in \mathcal{S}_l, j \in \mathcal{S}_l$ s.t. $p[i] \neq \mathbf{0}$ and $\mathcal{D}_\alpha[i][j] \neq \mathbf{0}$;
4     $p[j] \leftarrow Union(p[j], ImageSat(p[i], \mathcal{D}_\alpha[i][j]))$;
5 until $p$ does not change;
```

```
mdd $ImageSat$( mdd $q$, mdd2 $f$ )
1 if $q = \mathbf{0}$ or $f = \mathbf{0}$ then return $\mathbf{0}$;
2 $k \leftarrow q.lvl$;                • given our quasi-reduced form, $f.lvl = k$ as well
3 $s \leftarrow$ a new MDD node at level $k$ with all edges set to $\mathbf{0}$;
4 foreach $i \in \mathcal{S}_k, j \in \mathcal{S}_k$ s.t. $q[i] \neq \mathbf{0}$ and $f[i][j] \neq \mathbf{0}$ do
5     $s[j] \leftarrow Union(s[j], ImageSat(q[i], f[i][j]))$;
6 $Saturate(s)$;
7 return $s$.
```

Fig. 2: Saturation-based state-space generation.

subsets to the MDD in bottom-up fashion, as shown in Fig. 1, results in good speedups with respect to a strict BFS symbolic state-space generation.

Recognizing this *event locality* also lets us store $\mathcal{D}_\alpha$ with an MDD over just the current and next state variables having index k, for $Top(\alpha) \geq k \geq Bot(\alpha)$. Then, when computing the image of event α with $Top(\alpha) = l$, statement 5 in *BfsChaining* requires to access only MDD nodes at level l or below and to modify *in-place* [5] only MDD nodes at level l, without having to traverse the MDD from the root. Exploiting identity transformations in $\mathcal{D}_\alpha$ for variables strictly between $Top(\alpha)$ and $Bot(\alpha)$ is not as critical for the efficiency of the saturation approach, and therefore we do not discuss it in the rest of the paper, for simplicity's sake. However, it does contribute to the experimental results in Sect. 5.

2.3 Saturation algorithm

An MDD node p at level l is said to be *saturated* [6] if it encodes a fixed point:

$$\forall \alpha \in \mathcal{E}, Top(\alpha) \leq l,\ \mathcal{B}(K,p) \supseteq Img(\mathcal{B}(K,p), \mathcal{D}_\alpha).$$

To saturate node p once its descendants are saturated, we compute the effect of firing α on p for each α such that $Top(\alpha) = l$, recursively saturating any nodes at lower levels that might be created in the process, and add the result to $\mathcal{B}(p)$ using in-place updates. One advantage of this approach is that it stores only saturated nodes in the cache and unique table; these are the only "potentially useful" nodes, since nodes in the MDD encoding $\mathcal{S}$ are saturated by definition.

Fig. 2 shows the saturation algorithm in its most general form, as presented in [10]. Its fixed-point iterations constitute an extreme form of chaining. Saturation has been shown to reduce memory and runtime requirements by several orders of magnitude with respect to BFS-based algorithms when applied to asynchronous systems, for both state-space generation and CTL model-checking [8].

As an example, Fig. 3 shows a Petri net, and its equivalent guarded command language expression, modeling a gated-service queue with a limited pool of customers. New arrivals wait at the gate until it is opened, then all the waiting customers enter the service queue. Customers return to the pool after service. A state of the model can be represented as an evaluation of the integer variable vector (p, w, i), where p stands for *pool*, w for *wait* and i for *in-service.* Assuming a pool of two customers, the model has an initial state $(2, 0, 0)$ and six reachable states: $\mathcal{S} = \{(2, 0, 0), (1, 1, 0), (0, 2, 0), (1, 0, 1), (0, 0, 2), (0, 1, 1)\}$. Fig. 4 shows the execution of the saturation algorithm on this example. We use a for *arrive*, g for *gate*, and s for *service* to denote the transitions.

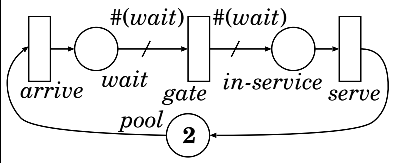
In Fig. 4, snapshot (a) shows the $2K$-level MDDs encoding the disjunctively partitioned transition relation. Snapshots from (b) to (k) show the evolution of the encoding of the state space, from the initial state to the final state space, where the key procedure calls are shown. For readability, node edges leading to terminal **0** are omitted. We denote each MDD node encoding the state space with a capital letter (A to I in the example), and color a node black after it becomes saturated. Not all procedure calls are shown, e.g., $\mathit{ImageSat}(C[1], \mathcal{D}_s[1][2])$ is called in snapshot (k) before node C becomes saturated, but it is not shown since no new nodes (states) are generated from the call.

3 Node-wise fine-grained chaining

Previously introduced chaining heuristics are *event-based*, thus *coarse-grained* (they define an order in which to explore the model-level events) and *static* (the order is derived from the high-level model prior to state-space generation). Our heuristic is instead *decision-diagram-node-based*, thus *fine-grained* (it defines the order of descent for the decision diagram nodes during image computation) and *dynamic* (the order is decided on a per-node basis during state-space generation). Such a dynamic policy has the potential to be more flexible and efficient than a static policy, but also the risk of higher runtime costs. In fact, Sect. 5 shows that our heuristic can achieve substantial improvements and has small overhead.

Before presenting our heuristic, we rewrite the transition relation by grouping the disjuncts according to the value of $\mathit{Top}(\alpha)$, i.e., $\mathcal{R} \equiv \bigvee_{K \geq l \geq 1} \mathcal{R}_l$, where $\mathcal{R}_l \equiv \bigvee_{\alpha \in \mathcal{E}_l} \mathcal{D}_\alpha \equiv \bigvee_{\alpha \in \mathcal{E}_l} (\bigwedge_j \mathcal{C}_{\alpha,j})$. Thus, $\mathcal{R}$ is described by a set of K MDDs where, for $K \geq l \geq 1$, the root $r_l^\star$ of the MDD encoding $\mathcal{R}_l$ is at level l (some of these MDDs can be empty).

Petri net model



Guarded command language model

initial state: $p = 2 \wedge w = 0 \wedge i = 0$;

$\mathcal{D}_a : p \geq 1 \rightarrow \{p' = p-1 \wedge w' = w+1 \wedge i' = i\}$;

$\mathcal{D}_s : i \geq 1 \rightarrow \{p' = p+1 \wedge w' = w \wedge i' = i-1\}$;

$\mathcal{D}_g : w \geq 1 \rightarrow \{p' = p \wedge w' = 0 \wedge i' = i+w\}$;

Fig. 3: A limited-arrival gated-service model with marking-dependent arc cardinalities.

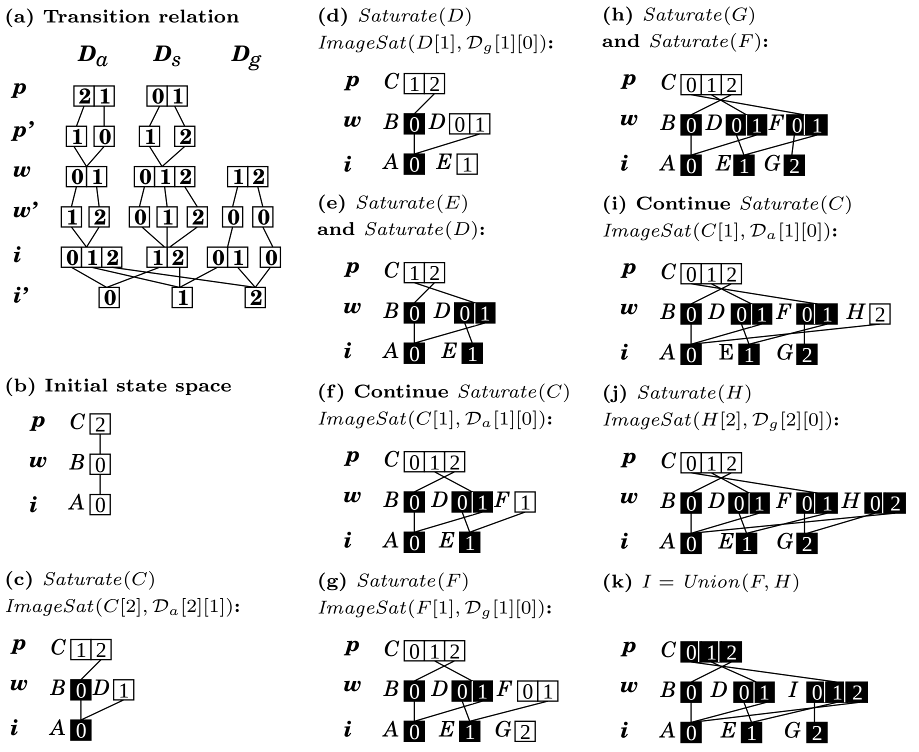

(a) Transition relation
D_a D_s D_g
p p' w w' i i'
(b) Initial state space
(c) Saturate(C) ImageSat(C[2], $\mathcal{D}_a$[2][1]):
(d) Saturate(D) ImageSat(D[1], $\mathcal{D}_g$[1][0]):
(e) Saturate(E) and Saturate(D):
(f) Continue Saturate(C) ImageSat(C[1], $\mathcal{D}_a$[1][0]):
(g) Saturate(F) ImageSat(F[1], $\mathcal{D}_g$[1][0]):
(h) Saturate(G) and Saturate(F):
(i) Continue Saturate(C) ImageSat(C[1], $\mathcal{D}_a$[1][0]):
(j) Saturate(H) ImageSat(H[2], $\mathcal{D}_g$[2][0]):
(k) I = Union(F, H)

Fig. 4: Saturation applied to the limited-arrival gate-service model.

We then seek good chaining by determining an order for the exploration of the children of a node p at level l and of $r^\star_l$, when computing $Image(p, r^\star_l)$ and using in-place updates. More precisely, we repeatedly choose a pair of local states $(i, j) \in \mathcal{S}_l \times \mathcal{S}_l$, compute $Image(p[i], r^\star_l[i][j])$, and use the result to update $p[j]$. Intuitively, pair (i', j') is preferred over (i'', j'') if there is a chance that updating j' can eventually "benefit" i'', i.e., increase $\mathcal{B}(p[i''])$, but no chance that updating j'' can benefit i'. Sect. 3.1 formalizes this concept by defining an equivalence relation on $\mathcal{S}_l$ that implies a partial order on the equivalence classes. This is the same rationale as for the original chaining heuristic, but at a much finer level. To further refine the order within each equivalence class, we use the "fullness" of the MDD nodes. Intuitively, if $p[i]$ encodes more substates than

$p[j]$, we want to compute $Image(p[i], r_l^\star[i][j])$ and use it to update $p[j]$ before computing $Image(p[j], r_l^\star[j][i])$ to update $p[i]$. The same is true if $p[i]$ and $p[j]$ encode the same number of substates, but $r_l^\star[i][j]$ encodes more transitions than $r_l^\star[j][i]$. Sect. 3.2 formalizes this idea by assigning a *score* to each pair (i, j). We stress that, while the first observation is based on logical conditions that are *guaranteed* to improve chaining, i.e., the best (total) chaining order must be compatible with the partial order, the second one is just a heuristic *likely* to improve chaining.

3.1 Partial-order-based chaining

Definition 3 Given node p at level l, $K \geq l \geq 1$, its *dynamic transition graph* is a directed graph $G_p = (\mathcal{S}_l, \mathcal{T}_p)$, where $\mathcal{T}_p = \{(i,j) : r_l^\star[i][j] \neq \mathbf{0} \wedge p[j] \neq \mathbf{1}\}$. $\square$

An edge $(i, j) \in \mathcal{T}_p$, if $p[i] \neq \mathbf{0}$, corresponds to an $Image(p[i], r_l^\star[i][j])$ that must eventually be computed when applying $r_l^\star$ to p. Obviously, if $p[i] = \mathbf{0}$ or $r_l^\star[i][j] = \mathbf{0}$, the image is $\mathbf{0}$, thus no computation is needed. Less obviously, we can avoid computation also when $p[j] = \mathbf{1}$, i.e., when $p[j]$ already encodes all possible substates; this new optimization could have been used in our original saturation algorithm [6], but was not (in other words, condition "$p[j] \neq \mathbf{1}$" should be added to the test in statement 3 of *Saturate* in Fig. 2).

Our heuristic chooses the pairs (i, j) respecting the partial order implied by graph: (i', j') is chosen before (i'', j'') if there is a path from j' to i'' but not from j'' to i', and $p[i'] \neq \mathbf{0}$. Note that computing $Image(p[i], r_l^\star[i][j])$ and using it to update $p[j]$ might change $p[j]$ from $\mathbf{0}$ to some other node, or it may make $p[j]$ become $\mathbf{1}$ (in which case we remove any of its incoming edges).

This first part of our heuristic considers the strongly connected components (SCCs) of the dynamic transition graph, and explores the edges according to their position in the resulting quotient graph. However, to discriminate between edges within the same SCCs, we need to refine our heuristic, which we do next.

3.2 Heuristic node-fullness-guided chaining

We define the *fullness* of a node as the ratio of the number of substates it encodes over the maximum number of substates it could encode, $n_{l:1} =_{\text{df}} \prod_{l \geq k \geq 1} n_k$.

Definition 4 The fullness of the terminal nodes is $\phi(\mathbf{0}) = 0$ and $\phi(\mathbf{1}) = 1$. The fullness of an MDD node p at level $l > 0$ is $\phi(p) = |\mathcal{B}(p)|/n_{l:1}$. $\square$

For models with boolean variables, the node fullness is the number of on-set minterms of the boolean function encoded by the BDD node over all the possible minterms. Our definition of node fullness is related to, but different from, the concept of node *density* proposed in [20]. We compare the two in Sect. 6.

Given p at level $l > 0$, we have $1/n_{l:1} \leq \phi(p) \leq 1$. If we store the value of ϕ with each node, or in a separate cache, we can compute it recursively bottom-up as $\phi(p) = \sum_{i \in \mathcal{S}_l} \phi(p[i])/n_l$. This definition can be applied also to the MDD encoding the transition relation $\mathcal{R}$ or of the disjuncts $\mathcal{D}_\alpha$. In practice, $\phi(p)$ is extremely small, and Sect. 4.1 addresses how to avoid floating point underflows.

To choose the next pair (i, j) to be explored when computing $\mathit{Image}(p, r_l^\star)$ if the partial order of Sect. 3.1 does not suffice, i.e., if there are edges (i', j') and (i'', j'') in the dynamic transition graph, with $p[i'] \neq \mathbf{0}$, $p[i''] \neq \mathbf{0}$, and paths from j' to i'' and from j'' to i', for each MDD node, we assign a *score* corresponding to each pair (i, j) and explore the pair with the highest score.

3.3 Scoring function based on probability

In this section, we restrict ourselves to a particular SCC of the dynamic transition graph. Using probabilistic arguments, we define the "score" of the pair (i, j) in node p at level l as $\sigma(p, i, j) = \phi(p[i]) \cdot \phi(r_l^\star[i][j]) \cdot (1 - \phi(p[j]))$. Then, in the image computation, we choose to explore next the pair with the highest score.

Assume that, for any i and j, the sets $\mathcal{B}(p[i])$, $\mathcal{B}(r_l^\star[i][j])$, and $\mathcal{B}(p[j])$ are independent and uniformly distributed random variables, i.e., that any of the $C_{n_{l-1:1}}^{|\mathcal{B}(p[i])|}$, $C_{n_{l-1:1}^2}^{|\mathcal{B}(r_l^\star[i][j])|}$, and $C_{n_{l-1:1}}^{|\mathcal{B}(p[j])|}$ possible choices for them are equally likely. The score of (i, j) could then be set to the expected fraction of new states found (ignoring the effect of saturating newly created nodes at lower levels): $\phi_{new} = E\left[|\mathcal{B}(\mathit{Image}(p[i], r_l^\star[i][j])) \setminus \mathcal{B}(p[j])|\right] / n_{l-1:1}$.

Consider the problem: given $A \in \mathbb{B}^n$, $R \in \mathbb{B}^{n \times m}$, and $B \in \mathbb{B}^m$, respectively with a, r, and b ones, compute $E[|\{j | B[j] = 0 \wedge \exists i, A[i] = 1 \wedge R[i, j] = 1\}|]/m$. Thanks to linearity, we can write this as $\rho(m - b)/m$, where ρ is the probability that, for a given j s.t. $B[j] = 0$, there is an i s.t. $A[i] = 1$ and $R[i, j] = 1$. We can compute the complementary probability $1 - \rho$ by observing that, if $A[i] = 0$, $R[i, j]$ can be either 0 or 1 but, if $A[i] = 1$, $R[i, j]$ must be 0. Thus, $1 - \rho$ is the probability that none of the ones in matrix R is in one of the "taboo" positions of column j corresponding to $A[i] = 1$, i.e., $1 - \rho = C_{nm-a}^r / C_{nm}^r$.

In our case, $n = m = n_{l-1:1}$, $a = |\mathcal{B}(p[i])| = n\phi(p[i])$, $r = |\mathcal{B}(r_l^\star[i][j])| = n^2\phi(r_l^\star[i][j])$, $b = |\mathcal{B}(p[j])| = n\phi(p[j])$, and ϕ_{new} is given by

$$\left[1 - \frac{C_{nm-a}^r}{C_{nm}^r}\right] \frac{m-b}{m} = \left[1 - \frac{(n^2 - n\phi(p[i]))!(n^2 - n^2\phi(r_l^\star[i][j]))!}{(n^2)!(n^2 - n^2\phi(r_l^\star[i][j]) - n\phi(p[i]))!}\right] [1 - \phi(p[j])].$$

Let $G = \phi(p[i])$, and $H = \phi(r_l^\star[i][j])$, then ϕ_{new} can be written as:

$$\begin{aligned}
\phi_{new} &= \left[1 - \frac{(n^2 - nG)!(n^2 - n^2H)!}{(n^2)!(n^2 - n^2H - nG)!}\right] [1 - \phi(p[j])] \\
&= \left[1 - \frac{(n^2 - n^2H)!\ /\ (n^2 - n^2H - nG)!}{(n^2)!\ /\ (n^2 - nG)!}\right] [1 - \phi(p[j])] \\
&= \left[1 - \frac{n^2 - n^2H}{n^2} \cdot \frac{n^2 - n^2H - 1}{n^2 - 1} \cdots \frac{n^2 - n^2H - (nG-1)}{n^2 - (nG-1)}\right] [1 - \phi(p[j])] \\
&= \left[1 - \left(1 - \frac{n^2H}{n^2}\right) \cdot \left(1 - \frac{n^2H}{n^2 - 1}\right) \cdots \left(1 - \frac{n^2H}{n^2 - (nG-1)}\right)\right] [1 - \phi(p[j])] \\
&\approx \left[1 - (1 - H)^{nG}\right] [1 - \phi(p[j])] && \text{(since } nG - 1 \ll n^2\text{)} \\
&\approx [1 - (1 - nGH + o(nGH))] [1 - \phi(p[j])] && \text{(assuming } nGH \ll 1\text{)} \\
&\approx [nGH] [1 - \phi(p[j])] && \text{(ignoring the higher order terms).}
\end{aligned}$$

We then use ϕ_{new}/n, i.e., $\phi(p[i]) \cdot \phi(r_l^\star[i][j]) \cdot (1 - \phi(p[j]))$, as the value of the scoring function $\sigma(p, i, j)$, since n is the same for all the nodes at level l. Observe that, just as ϕ_{new}, $\sigma(p, i, j)$ lies between 0 and 1, increases with $\phi(p[i])$ and $\phi(r_l^\star[i][j])$, decreases with $\phi(p[j])$, and evaluates to 0 when $p[i] = \mathbf{0}$, $r_l^\star[i][j] = \mathbf{0}$, or $p[j] = \mathbf{1}$, as it should be expected.

4 Fine-grained chaining symbolic state-space generation

Fig. 5 and 6 show the modified BFS-based and saturation-based symbolic state-space generation algorithms with our fullness-guided chaining. Unlike the algorithm of Fig. 1 that applies each event α with $Top(\alpha) = l$, for $l = 1, ..., K$, once, the one in Fig. 5 leaves level l only when further applications of $\mathcal{R}_l$ do not add states. To distinguish between the different aggressiveness in chaining, we indicate them as Top and $\mathsf{Top}^\star$. The modified saturation-based algorithm is as in Fig. 2, except that *SaturateFineGrainedChaining* replaces *Saturate*.

Both chaining algorithms use a dynamic transition graph $(\mathcal{S}_l, \mathcal{T}_p)$ to repeatedly explore transitions, corresponding to edges (i, j), until no new substates can be added. The edges of the graph are marked if they need to be explored. Function $InitTransGraph(p, r_l^\star)$ initializes this graph according to Def. 3 and marks any edge leaving a node i, if $p[i]$ encodes some substates. Function $UpdateTransGraph(p, j, \mathcal{T}_p)$ updates the set of edges according to Def. 3 after $p[j]$ has been modified, i.e., it removes any edge directed to node j, if $p[j] = 1$, and marks the edges leaving node j, since they need to be explored.

The algorithms call a function *ChooseEdge* which, given an mdd p, an $\mathsf{mdd2}$ $r_l^\star$, and a set of edges $\mathcal{T}_p$, returns the (marked) edge (i, j) to explore next, based on the partial order and scoring function heuristics. The marking of edges is essential since, once (i, j) has been explored, there is no need to explore it again unless $p[i]$ changes.

The quotient graph and score computations for our heuristic can be implemented efficiently. Using the execution profiler `gprof` on the benchmarks described in the next section, we can experimentally conclude that the total runtime overhead incurred by calling *ChooseEdge* is less than 1%. To achieve this low overhead, we initially build a static transition graph $(\mathcal{S}_l, \mathcal{T}_l)$ for each level l, based on the information in $r_l^\star$ alone, and compute its quotient graph. Then, when applying $r_l^\star$ to a node p during state-space generation, we build the dynamic graph transition as a subgraph of the static one (removing edges to any node j such that $p[j] = \mathbf{1}$) and its quotient graph as a refinement of the static one. We store the "marked" flag and the score in an adjacency matrix, restricted to the current SCC for efficiency. After a call $UpdateTransGraph(p, j, \mathcal{T}_p)$, the score of all edges incident to j is recomputed. The element with maximum score in each row of the adjacency matrix is recorded, so that *ChooseEdge* can efficiently find the maximum without searching the entire matrix.

4.1 Implementation details

One interesting issue is how to recognize the condition $\phi(p[j]) = 1$ appearing in our heuristic. As presented so far, we simply need to test whether $p[j] = \mathbf{1}$

```
set of (int,int) $InitTransGraph$(mdd $p$, mdd2 $r_l^\star$)
$\mathcal{T}_p \leftarrow \{(i,j) \in \mathcal{S}_l \times \mathcal{S}_l : r_l^\star[i][j] \neq \mathbf{0} \wedge p[j] \neq \mathbf{1}\}$;    • Definition 3
mark edges $\{(i,j) \in \mathcal{T}_p : p[i] \neq \mathbf{0}\}$;    • transitions to be explored
return $\mathcal{T}_p$;
```

```
set of (int,int) $UpdateTransGraph$(mdd $p$, int $j$, set of (int,int) $\mathcal{T}_p$)
mark edges $\{(j,h) \in \mathcal{T}_p : p[h] \neq \mathbf{1}\}$;    • edges leaving $j$ must be (re-)explored
if $p[j] = \mathbf{1}$ then    • no new substates can be added to $\mathcal{B}(p[j])$
    $\mathcal{T}_p \leftarrow \mathcal{T}_p \setminus \mathcal{S}_l \times \{j\}$;    • remove all edges directed toward node $j$
return $\mathcal{T}_p$;
```

```
mdd $BfsFineGrainedChaining$( )
$\mathcal{S} \leftarrow \mathcal{S}^{init}$;
repeat
    for $l = 1$ to $K$ do
        foreach node $p$ at level $l$ in the MDD of $\mathcal{S}$ do
            $\mathcal{T}_p \leftarrow InitTransGraph(p, r_l^\star)$;    • build graph $(\mathcal{S}_l, \mathcal{T}_p)$ and mark its edges
            while there is a marked edge in $\mathcal{T}_p$ do
                $(i,j) \leftarrow ChooseEdge(p, r_l^\star, \mathcal{T}_p)$;
                unmark edge $(i,j)$;
                $u \leftarrow Union(p[j], Image(p[i], r_l^\star[i][j]))$;    • image computation
                if $u \neq p[j]$ then    • new substates found
                    $p[j] \leftarrow u$;    • in-place update $p$ to add the new substates
                    $\mathcal{T}_p \leftarrow UpdateTransGraph(p, j, \mathcal{T}_p)$;
until no node in the MDD encoding $\mathcal{S}$ has changed;    • fixed-point computation
return $\mathcal{S}$;    • the reachable state space
```

Fig. 5: Fine-grained chaining variant of *BfsChaining*.

```
void $SaturateFineGrainedChaining$(mdd $p$)
$l \leftarrow p.lvl$;
$\mathcal{T}_p \leftarrow InitTransGraph(p, r_l^\star)$;    • build graph $(\mathcal{S}_l, \mathcal{T}_p)$ and mark its edges
while there is a marked edge in $\mathcal{T}_p$ do
    $(i,j) \leftarrow ChooseEdge(p, r_l^\star, \mathcal{T}_p)$;
    unmark edge $(i,j)$;
    $u \leftarrow Union(p[j], ImageSat(p[i], r_l^\star[i][j]))$;    • image computation
    if $u \neq p[j]$ then    • new substates found
        $p[j] \leftarrow u$;    • in-place update $p$ to add the new substates
        $\mathcal{T}_p \leftarrow UpdateTransGraph(p, j, \mathcal{T}_p)$;
```

Fig. 6: Fine-grained chaining variant of *Saturate* for the algorithm in Fig. 2.

but, in the past, we have proposed symbolic state-space generation algorithms where the actual ranges of the state variables are initially unknown, so that the sets $\mathcal{S}_l$, for $K \geq l \geq 1$, are built "on the fly" during the symbolic iterations [10]. In this case, only non-terminal nodes at level 1 can point to node **1**, since the meaning of $\mathcal{B}(l, \mathbf{1})$, and the value of $\phi(p)$, change every time one of the sets $\mathcal{S}_k$, for $l \geq k \geq 1$, changes. In this setting, it is best to store the absolute substate count $|\mathcal{B}(p)|$ instead of $\phi(p)$. Our tool SMART [3] provides an arbitrary precision integer state counting capability, but this is relatively expensive in terms of memory and time for large MDDs, so we store $|\mathcal{B}(p)|$ using a floating-point value, appropriately scaled. Then, however, recognizing that $p[j]$ encodes all (so far) possible substates, i.e., that $\phi(p[j]) = 1$, is feasible only if the value of $|\mathcal{B}(p[j])|$ is stored with enough precision that the comparison $|\mathcal{B}(p[j])| = n_{l-1:1}$ is reliable. Testing whether $\phi(p[i]) = 0$ or $\phi(r_l^\star[i][j]) = 0$, instead, is always possible, since it is equivalent to testing whether $p[i] = \mathbf{0}$ or $r_l^\star[i][j] = \mathbf{0}$, respectively, and edges to $\mathbf{0}$ from non-terminal nodes at any level can be present even if the variable ranges are not known a priori.

5 Experimental results

We now report results on a suite of asynchronous benchmarks parametrized by an integer N. We compare three different fine-grained chaining orders applied to the $\mathsf{Top}^\star$ and saturation algorithms: (1) a random order, (2) the order used in [7], where set of edges to explore is initialized using the order in which local states are discovered during symbolic state-space generation, then managed as a FIFO queue, and (3) the proposed fullness-guided order, respectively denoted $\mathsf{Top}_r^\star$, $\mathsf{Top}_d^\star$, $\mathsf{Top}_g^\star$, Sat_r, Sat_d, Sat_g, and compare them with (4) pure BFS without chaining and (5) Top, the coarse-grained chaining of Fig. 1. All algorithms are implemented in our tool SMART [3], run on a 3 Ghz Pentium IV workstation with 1GB memory. In all our experimental benchmarks, we use the best MDD variable order known to us, which is either obtained from the variable ordering heuristic described in [23] or derived manually in the high-level model.

The columns of Table 1 report the value of the parameter N, the size $|\mathcal{S}|$ of the state space, the runtime and peak number of MDD nodes for each approach, and the final number of MDD nodes. For each benchmark, we provide the number of state variables K as a function of the model parameter N, and give a reference where a detailed description of the model can be found. These benchmarks include network protocols (Aloha, Leader, Slot), generalized queueing/Petri net models (BQ, Kanban), and a speed-independent asynchronous circuit model (DME) from the NuSMV [11] distribution.

From this table, we see a separation by several orders of magnitude in terms of efficiency, with BFS at the lowest end, followed by Top, then $\mathsf{Top}^\star$, and finally Sat. Comparing different fine-grained chaining heuristics in the $\mathsf{Top}^\star$ and saturation algorithms, the chaining heuristic has significant impact on five out of the six benchmarks. In the DME benchmark, the “r”, “d”, and “g” orders result in exactly the same peak MDD nodes; as the runtimes are similar, this confirms that our fullness-guided heuristic has a small overhead.

Compared with random-order chaining, discovery-order chaining has worse performance for $\mathsf{Top}^\star$ in three cases and similar performance in the remaining four cases, while, for Sat, it has better performance in three cases, worse performance in one case, and similar performance in four cases. Compared with random-order or discovery-order chaining, the newly proposed fullness-guided chaining heuristic achieves better (by a factor of up to four) or similar runtime and memory consumption for both $\mathsf{Top}^\star$ and Sat in all benchmarks.

We conclude from our experiments that the newly proposed fullness-guided chaining heuristic is more efficient and stable than both discovery chaining order and random chaining order. In particular, the Sat_g algorithm is by far the most efficient among the eight algorithms we considered.

N	$\vert\mathcal{S}\vert$	Runtime (sec)								Peak Nodes (in thousands)								Final
		BFS	Top	Top$^\star_r$	Top$^\star_d$	Top$^\star_g$	Sat$_r$	Sat$_d$	Sat$_g$	BFS	Top	Top$^\star_r$	Top$^\star_d$	Top$^\star_g$	Sat$_r$	Sat$_d$	Sat$_g$	Nodes
Bounded open queuing network (BQ) [12]																		N is the number of customers and $K = 8$
20	$2.3{\cdot}10^7$	1,865	1,739	72	124	23	0.22	0.24	0.09	46	16	10	12	7	6	7	2	88
30	$2.4{\cdot}10^7$	-	-	996	1,667	217	0.7	0.73	0.28	-	-	24	30	18	13	15	5	128
200	$1.7{\cdot}10^{13}$	-	-	-	-	-	257	220	65	-	-	-	-	-	537	646	168	808
Aloha network protocol (Aloha) [4]																		N is the number of nodes in the network and $K = N + 3$
18	$2.6{\cdot}10^6$	21	20	10	20	9	0.07	0.09	0.05	6	6	5	6	5	5	6	5	551
24	$2.1{\cdot}10^8$	1,669	1,670	875	1,666	735	0.16	0.22	0.12	13	13	12	13	10	12	12	10	950
180	$1.4{\cdot}10^{56}$	-	-	-	-	-	76	103	53	-	-	-	-	-	4,410	4,925	3,954	49,232
Kanban manufacturing system (Kanban) [24]																		N is the number of each type of parts and $K = 16$
7	$4.1{\cdot}10^7$	2,016	899	3	4	3	0.04	0.03	0.03	38	6	2	3	2	0.94	1	0.92	107
15	$4.7{\cdot}10^{13}$	-	-	836	1,020	535	0.28	0.16	0.15	-	-	9	9	8	4	4	4	211
150	$1.4{\cdot}10^{21}$	-	-	-	-	-	396	121	90	-	-	-	-	-	338	364	309	1,966
Leader election protocol (Leader) [13]																		N is the number of processors in the ring and $K = 11N$
5	$5.9{\cdot}10^5$	218	139	48	46	45	3	3	2	1,495	977	376	373	372	121	123	108	18,401
6	$9.8{\cdot}10^6$	-	-	1,083	1,067	1,020	16	16	14	-	-	2,390	2,368	2,352	631	661	557	66,967
7	$1.3{\cdot}10^8$	-	-	-	-	-	53	50	45	-	-	-	-	-	1,895	1,872	1,594	142,412
Slotted ring network protocol (Slot) [19]																		N is the number of nodes in the network and $K = N$
7	$6.2{\cdot}10^6$	367	132	34	30	30	0.01	0.01	0.01	19	4	0.38	0.37	0.37	0.11	0.1	0.08	31
8	$6.8{\cdot}10^7$	-	1,622	378	347	347	0.01	0.01	0.01	-	7	0.95	0.89	0.89	0.18	0.2	0.16	40
200	$8.4{\cdot}10^{211}$	-	-	-	-	-	207	149	86	-	-	-	-	-	1,692	1,408	732	20,200
Distributed mutual exclusion circuit (DME) [16]																		N is the number of cells in the protocol and $K = 18N$
4	$7.5{\cdot}10^4$	309	113	3	3	3	0.01	0.01	0.01	751	228	18	18	18	3	3	3	1,422
5	$8.0{\cdot}10^5$	-	-	34	36	34	0.14	0.14	0.16	-	-	29	29	29	4	4	4	1,955
350	$8.8{\cdot}10^{324}$	-	-	-	-	-	15	15	16	-	-	-	-	-	391	391	391	185,840

Table 1: Experimental results ("-" indicates that the runtime is over 3600 seconds).

6 Related work

Ravi and Somenzi [20] defined the *density* of a BDD node p as the ratio of the number of substates encoded by p over the number of BDD nodes reachable from p, and used it to make decisions about which nodes to explore during state-space generation, with the goal of reducing memory consumption. In particular, their algorithm could ignore low-density nodes and explore just high-density ones, at the price of computing a (hopefully good) under-approximation of the exact state space. In contrast, our definition of node fullness does not take into account the number of decision diagram nodes needed to encode the node's function. Most importantly, we use it for a fundamentally different purpose. Instead of exploring only dense nodes and computing an under-approximation of the state space, we look for asynchronous transition from high-fullness nodes to low-fullness nodes with the goal of reducing the number of symbolic iterations required to reach the exact fixed-point via chaining.

In another related work, *hints* [21] were proposed with the same intent to guide the symbolic traversal of the state space and avoid intermediate BDD blow-ups in symbolic invariance checking and model checking. Hints are constraints which are added to the transition relation before the start of symbolic state-space exploration, and later removed from the transition relation to compute the exact solution, with the hope to avoid the peak memory consumption. However, this is orthogonal to our approach, since hints were designed to be dependent on the high-level model (as well as on the properties being verified), and either provided by model checker users [21] or automatically generated from the input model [26], while our approach lies in the symbolic back-end solver and is completely independent of the high-level model.

7 Conclusion and future work

We introduced a new approach to exploit *chaining* during symbolic state-space generation. Unlike previous heuristics that operate on a high-level model and decide in which order to explore events, ours considers low-level information extracted from the decision diagrams encoding the current state space and the transition relation, thus it is applicable to any globally-asynchronous locally-synchronous system, regardless of the formalism used to model it. We implemented our heuristic in both BFS-based and saturation-based algorithms, and experimentally demonstrated that runtimes and memory requirements can improve by a factor up to four. Having established the soundness of the approach, we plan in the future to investigate further refinements of the proposed heuristic.

References

1. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, Aug. 1986.
2. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. *Proc. Int. Conference on Very Large Scale Integration*, pages 49–58, Aug. 1991. IFIP Transactions, North-Holland.

3. G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. *Perf. Eval.*, 63:578-608, 2006.
4. G. Ciardo and Y. Lan. Faster discrete-event simulation through structural caching. *Proc. PMCCS*, pages 11–14, Sept. 2003.
5. G. Ciardo, Lüttgen, Gerald and G. Ciardo, G. Lüttgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. *Proc. ICATPN*, LNCS 1825, pages 103-122, Jun. 2000. springer.
6. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. *Proc. TACAS*, LNCS 2031, pages 328–342, Apr. 2001. Springer.
7. G. Ciardo, R. Marmorstein, and R. Siminiceanu. The saturation algorithm for symbolic state space exploration. *STTT*, 8(1):4-25, Feb. 2006.
8. G. Ciardo and R. Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. *Proc. CAV*, LNCS 2725, pages 40–53, July 2003. Springer.
9. G. Ciardo and K. S. Trivedi. A decomposition approach for stochastic reward net models. *Perf. Eval.*, 18(1):37–59, 1993.
10. G. Ciardo and A. J. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. *Proc. CHARME*, LNCS 3725, pages 146–161, Oct. 2005. Springer.
11. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. *Proc. CAV*, LNCS 1633, pages 495–499, 1999. Springer.
12. P. Fernandes, B. Plateau, and W. J. Stewart. Efficient descriptor-vector multiplication in stochastic automata networks. *J. ACM*, 45(3):381–414, 1998.
13. A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Proc. FOCS*, pages 150–158. IEEE Comp. Soc. Press, Oct. 1981.
14. T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.
15. S. Kimura and E. M. Clarke. A parallel algorithm for constructing binary decision diagrams. *Proc. ICCD*, pages 220–223, Sept. 1990. IEEE Comp. Soc. Press.
16. A. J. Martin. The design of a self-timed circuit for distributed mutual exclusion. *Proc. Chapel Hill Conference on VLSI*, pages 245–260, 1985.
17. K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.
18. A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. *Proc. ICATPN*, LNCS 1639, pages 6–25, June 1999. Springer.
19. E. Pastor, O. Roig, J. Cortadella, and R. Badia. Petri net analysis using boolean manipulation. *Proc. ICATPN*, LNCS 815, pages 416–435, June 1994. Springer.
20. K. Ravi and F. Somenzi. High-density reachability analysis. *Proc. ICCAD*, pages 154–158. IEEE Comp. Soc. Press, 1995.
21. K. Ravi and F. Somenzi. Hints to accelerate Symbolic Traversal. *Proc. CHARME*, pages 250–264 , 1999.
22. O. Roig, J. Cortadella, and E. Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri nets. *Proc. ICATPN*, LNCS 935, pages 374–391. Springer, June 1995.
23. R. Siminiceanu and G. Ciardo. New metrics for static variable ordering in decision diagrams. *Proc. TACAS*, LNCS 2031, pages 328–342. Springer, March 2006.
24. M. Tilgner, Y. Takahashi, and G. Ciardo. SNS 1.0: Synchronized Network Solver. *Proc. 1st Int. Workshop on Manuf. and Petri Nets*, pages 215–234, June 1996.
25. The VIS Group. VIS: A system for verification and synthesis. *Proc. CAV*, LNCS 1102, pages 428–432, Springer, July 1996.
26. D. Ward and F. Somenzi. Automatic Generation of Hints for Symbolic Traversal. *Proc. CHARME*, pages 207–221 , 2005.