Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

The saturation algorithm for symbolic state-space exploration[*]

Gianfranco Ciardo[1], Robert Marmorstein[2], Radu Siminiceanu[3]

[1] Department of Computer Science and Engineering, University of California, Riverside, Riverside, CA, 92521, USA; e-mail: ciardo@cs.ucr.edu
[2] College of William and Mary, Department of Computer Science, P.O. Box 8795, Williamsburg, VA 23187-8795, USA; e-mail: rmmarm@cs.wm.edu
[3] National Institute of Aerospace, 144 Research Drive, Hampton, VA 23666, USA; e-mail: radu@nianet.org

Received: date / Revised version: date

Abstract. We present various algorithms for generating the state space of an asynchronous system, based on the use of multi-way decision diagrams to encode sets and Kronecker operators on boolean matrices to encode the next-state function. The Kronecker encoding allows us to recognize and exploit the "locality of effect" that events might have on state variables. In turn, locality information suggests better iteration strategies aimed at minimizing peak memory consumption. In particular, we focus on the *saturation* strategy, which is completely different from traditional breadth-first symbolic approaches, and extend its applicability to models where the possible values of the state variables are not known a priori. The resulting algorithm merges "on-the-fly" explicit state-space generation of each submodel with symbolic state-space generation of the overall model.

Each algorithm we present is implemented in our tool SMART. This allows us to run fair and detailed comparisons between them, on a suite of representative models. Saturation, in particular, is shown to be many orders of magnitude more efficient in terms of memory and time with respect to traditional methods.

Key words: globally-asynchronous locally-synchronous systems – symbolic state-space generation – decision diagrams – Kronecker algebra – fixed-point iterations

1 Introduction

Safety-critical systems, such as flight guidance and early warning systems, require a high degree of assurance of design correctness. For these systems, failure can mean huge costs in productivity or even human life. Prevalent techniques for verifying safety and other properties rely heavily on extensive testing and debugging, which examine only a portion of the behaviors of the system. Since flaws can hide in the unexamined portions of the system, designers should instead employ exhaustive analysis techniques. Even for less critical applications, investments into formal specification and verification can pay off dramatically in increased reliability.

Most formal verification tools require that discrete-state models be expressed in *high-level* formalisms such as Petri nets [37], communicating sequential processes [31], and process algebras [3]. The algorithms implemented by these tools require knowledge of the reachable states of a system. State-space generation is an essential prerequisite to both model checking [22] algorithms and simpler liveness and safety queries. For example, a modeler might be interested in identifying deadlock states, and proves or disproves the existence of such states via an exhaustive search.

Traditional techniques for generating and storing the state space fall into two categories. Explicit techniques store each state individually, normally using some variant of breadth-first or depth-first search to discover the reachable states. Symbolic techniques instead represent states implicitly, using advanced data structures such as binary decision diagrams (BDD) [6] to encode and manipulate entire sets of states at once.

Unfortunately, both techniques are limited by the state explosion problem: as the complexity of a system increases, the memory and time required to store the combinatorially expanding state space can easily become excessive. Researchers have addressed this problem in several ways. Explicit techniques take advantage of partial order reduction [25,32,46], or symmetries [21] in the system to reduce the number of states that must be explored. These approaches explore and store only a "rep-

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

resentative" subset of the reachable states, but still provide the same capabilities as an exhaustive search.

Symbolic techniques have been successful in allowing the storage of very large state spaces [8] and providing truly exhaustive explorations, but they are limited by memory and time requirements as well. The major bottleneck is widely recognized to be the *peak size* of the BDDs, usually reached at some intermediate point during the exploration. To overcome these limitations, several directions of research have been proposed. The idea of a disjunctively-partitioned transition relation is natural for asynchronous systems [23,9], mostly because it allows the expression of a large, global transition relation as collection of smaller, local transition relations.

A great deal of effort has been put into finding flexible or adaptive strategies in applying the individual transitions, or clusters of transitions, to avoid the BDD explosion. The algorithms in [38,44] aim at reducing the number of iterations needed to reach all states, by producing a pipelining effect; using a static causality analysis, events can be ordered so that their firing increases the chance of finding new states at each step. Other approaches attempt a partial symbolic traversal on certain subsets of the newly reached states in the first stages and delay the exploration on the other portion, only to complete the full search in the end. The *high density* approach in [42] targets those subsets that have the most compact symbolic representation to advance the search in the early stages. The technique in [10] computes, in an initial learning phase, the *activity profiles* of each BDD node in the transition relation and uses this information to prune the decision diagrams of the inactive nodes. The *guided* search of [4] exploits user hints to simplify the transition relation, but this requires a priori knowledge of the system to predict the evolution of the BDD.

Another direction has focused on parallelizing symbolic algorithms [29,47]. For example, distributed algorithms for state-space generation are quite successful at making use of the overall memory available in a network of workstations, but much work is still needed to cope with communication latency and to devise work partitioning approaches that can provide nearly ideal speedup.

In this work, we "return to the basics" by analyzing some of the reasons current techniques suffer from memory and time limitations, and introduce new methods that greatly improve on these problems. These methods, including *Kronecker encoding of the next-state function* and *saturation-based iteration strategy* have been presented in our previous work [13–15,36], but have never been described in full. Here, we finally provide a detailed analysis of their properties and implementation.

The rest of this paper is organized as follows. Sect. 2 gives an overview of traditional state-space generation algorithms using decision diagrams and introduces our terminology. Sect. 3-4 describe our techniques and provide detailed pseudocode for them. Sect. 5 examines some important details of their implementation in our tool SMART [12]. Sect. 6 presents data comparing the memory and time requirements on a suite of models, for several alternative algorithms implemented in SMART, ranging from symbolic breath-first generation to our advanced saturation algorithm, and for the tool NuSMV. Finally, Sect. 7 summarizes the contributions of this work and discusses areas of future research.

2 State-space generation and decision diagrams

A discrete-state model is a triple $(\widehat{\mathcal{S}}, \mathbf{s}, \mathcal{N})$, where

- $\widehat{\mathcal{S}}$ is the set of *potential states* of the model, specifying the "type" of the states. We indicate states with lowercase boldface letters, e.g., $\mathbf{i}$.
- $\mathbf{s} \in \widehat{\mathcal{S}}$ is the *initial state* of the model. A set of initial states is often assumed in formal verification approaches; our method handles this case just as easily.
- $\mathcal{N} : \widehat{\mathcal{S}} \to 2^{\widehat{\mathcal{S}}}$ is the *next-state function* specifying the states reachable from each state in a single step.

As we target asynchronous systems, we partition $\mathcal{N}$ into a union of next-state functions [9]: $\mathcal{N}(\mathbf{i}) = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha(\mathbf{i})$, where $\mathcal{E}$ is a finite set of *events*, $\mathcal{N}_\alpha$ is the next-state function associated with event α, i.e., $\mathcal{N}_\alpha(\mathbf{i})$ is the set of states the system can enter when α occurs, or *fires*, in state $\mathbf{i}$. An event α is said to be *disabled* in $\mathbf{i}$ if $\mathcal{N}_\alpha(\mathbf{i}) = \emptyset$; otherwise, it is *enabled*.

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

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

where "$*$" denotes the reflexive and transitive closure, and $\mathcal{N}$ applies also to sets of states, $\mathcal{N}(\mathcal{X}) = \bigcup_{s \in \mathcal{X}} \mathcal{N}(s)$. Another way to define $\mathcal{S}$ is as the smallest fixed point of the equation

$$\mathcal{S} = \{\mathbf{s}\} \cup \mathcal{N}(\mathcal{S}).$$

Since $\mathcal{N}$ is the union of several functions $\mathcal{N}_\alpha$, we can build $\mathcal{S}$ by applying these functions in an arbitrary order, as long as we consider each event α often enough [24].

2.1 Explicit state-space generation

In traditional explicit state-space generation approaches, states are discovered one by one, see for example Fig. 1. Often, states are explored in breadth-first order, this is achieved by treating $\mathcal{U}$ as a *queue* of states. Statement 8 is performed as many times as there are state-to-state transitions between reachable states, thus an efficient search data structure, such as a hash table or a search tree, must be used to organize the states in $\mathcal{S}$.

Each state may require many bytes for its storage, so it is efficient to *index* states with a function $\psi : \widehat{\mathcal{S}} \to \mathbb{N} \cup \{\mathsf{null}\}$ such that $\psi(\mathbf{i}) = i \in \{0, ..., |\mathcal{S}|-1\}$ if $\mathbf{i} \in \mathcal{S}$

```
ExplicitSsGen($\mathbf{s}$ : state, $\mathcal{N}$ : stateset $\rightarrow 2^{stateset}$) : stateset
Compute $\mathcal{S} \subseteq \widehat{\mathcal{S}}$ starting from $\mathbf{s}$ by repeatedly applying $\mathcal{N}$.
 1 declare $\mathcal{S}$,$\mathcal{U}$:stateset,  $\psi$:stateset$\rightarrow\mathbb{N}\cup\{$null$\}$,  $\mathbf{i}$,$\mathbf{j}$:state;
 2 $\mathcal{S} \leftarrow \{\mathbf{s}\}$;                                   • known states
 3 $\mathcal{U} \leftarrow \{\mathbf{s}\}$;                         • unexplored known states
 4 $\psi(\mathbf{s}) \leftarrow 0$;                  • 0 is the index of the first state
 5 while $\mathcal{U} \neq \emptyset$ do                           • still states to explore
 6   choose a state $\mathbf{i}$ in $\mathcal{U}$ and remove it from $\mathcal{U}$;
 7   for each $\mathbf{j} \in \mathcal{N}(\mathbf{i})$ do
 8     if $\mathbf{j} \notin \mathcal{S}$ then                       • if $\mathbf{j}$ is a new state...
 9       $\psi(\mathbf{j}) \leftarrow |\mathcal{S}|$;           • ...assign the next index to it,...
10       $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathbf{j}\}$;                   • ...add it to $\mathcal{S}$, and...
11       $\mathcal{U} \leftarrow \mathcal{U} \cup \{\mathbf{j}\}$;             • ...remember to explore it later
12 return $\mathcal{S}$;
```

Fig. 1. Explicit state space exploration

and $\psi(\mathbf{i}) =$ null if $\mathbf{i} \notin \mathcal{S}$. A natural definition for ψ is to assign increasing indices to new states in the order they are found, starting from $\psi(\mathbf{s}) = 0$, as shown in Fig. 1.

Of course, any explicit method requires memory and time at least proportional to the number of reachable states, so it is feasible only when $\mathcal{S}$ contains at most 10^8 or perhaps 10^9 states, given today's workstation capabilities. Nevertheless, we briefly presented the explicit algorithm because it is employed by the symbolic algorithms we discuss.

2.2 Structured high-level models

A common characteristic of the high-level models we consider is that they can be partitioned, or *structured*, into interacting *submodels*. For a model composed of K submodels, a *global* system state $\mathbf{i}$ can be written as a K-tuple $(\mathbf{i}_K, \ldots, \mathbf{i}_1)$, where $\mathbf{i}_k$ is the *local* state of submodel k, for $K \geq k \geq 1$. Thus, the potential state space $\widehat{\mathcal{S}}$ is given by the cross-product $\mathcal{S}_K \times \cdots \times \mathcal{S}_1$ of K *local* state spaces, where $\mathcal{S}_k$ contains (at least) all the possible local states for submodel k. For example, the places of a Petri net can be partitioned into K subsets, so that the marking can be written as a vector of the K corresponding submarkings.

Partitioning a model into submodels enables us to use techniques targeted at exploiting system structure, in particular *symbolic* encodings based on decision diagrams. Returning to the issue of indexing, just as we map each reachable global state $\mathbf{i}$ to a natural number i in the explicit approach, we map local states as well. Assuming for now that each local state space $\mathcal{S}_k$ is known a priori, i.e., can be built considering the k^{th} submodel in isolation, we define K mappings $\psi_k : \mathcal{S}_k \rightarrow \{0, 1, \ldots, n_k - 1\}$, where n_k is the size of the local state space $\mathcal{S}_k$. Then, a global structured state $(\mathbf{i}_K, \ldots, \mathbf{i}_1)$ can be mapped to a vector $(i_K, \ldots, i_1)$ of indices. This was observed and exploited in [16], where we showed that, for an appropriate decomposition of the model into submodels, it is possible to store $\mathcal{S}$ explicitly using a "multi-level" data structure that requires only $O(\log n_1)$ bits per state, instead of $O(\sum_{K \geq k \geq 1} \log n_k)$. In the following, to stress the difference between states and their indices, we will consistently use lowercase non-bold letters for the latter. However, to keep notation simple, we use $\mathcal{S}_k$ to mean either the set of local states or of their indices, $\{\psi_k(\mathbf{i}_k) : \mathbf{i}_k \in \mathcal{S}_k\} = \{0, 1, \ldots, n_k - 1\}$, since there is a bijection between the two.

2.3 Decision diagrams

In the last decade, *symbolic* approaches have been used extensively to manipulate and store very large sets of states [8]. The key enabling technology is the use of *decision diagrams*, most often *reduced ordered binary decision diagrams* (RO)BDDs [6]. An N-variable BDD encodes a function $f : \{0,1\}^N \rightarrow \{0,1\}$. We can use $\lceil \log n_k \rceil$ boolean variables $x_{k,1}, \ldots, x_{k,\lceil \log n_k \rceil}$ to encode the k^{th} local state index i_k, for $K \geq k \geq 1$. Then, $N = \sum_{K \geq k \geq 1} \lceil \log n_k \rceil$ and the BDD can be used to store a set of states $\mathcal{X} \subseteq \mathcal{S}_K \times \cdots \times \mathcal{S}_1$ by encoding its indicator function $f_{\mathcal{X}}$, where

$$f_{\mathcal{X}}(x_{K,1}, \ldots, x_{K,\lceil \log n_K \rceil}, \ldots, x_{1,1}, \ldots, x_{1,\lceil \log n_1 \rceil}) = 1$$
$$\Leftrightarrow \ (\mathbf{i}_K, \ldots, \mathbf{i}_1) \in \mathcal{X}$$

and the integer value of $(x_{k,1}, \ldots, x_{k,\lceil \log n_k \rceil})$, interpreted as a binary string, is $\psi_k(\mathbf{i}_k)$, for $K \geq k \geq 1$.

Instead of giving more details on (RO)BDDs, we now turn to the version of decision diagrams employed by our approach, *quasi-reduced multi-way decision diagrams.* Multi-way decision diagrams (MDDs) [33] were introduced as a more natural way to encode the types of sets we are discussing, since they allow discrete variables to take values over arbitrary finite sets instead of requiring boolean variables. The quasi-reduced version of MDDs we now define is a natural combination of the multi-way extension with the quasi-reduced canonical format introduced for BDDs in [34].

Formally, a K-variable quasi-reduced ordered multi-way decision diagram, or MDD for short, is a directed acyclic edge-labeled multi-graph where:

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

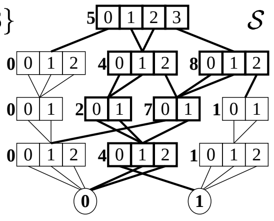
We can extend our arc notation to paths. The index of the node at level $l-1$ reached from a node $\langle k|p \rangle$

$\mathcal{S}_4 = \{0,1,2,3\}$

$\mathcal{S}_3 = \{0,1,2\}$

$\mathcal{S}_2 = \{0,1\}$

$\mathcal{S}_1 = \{0,1,2\}$



$\mathcal{S} = \{1000, 1010, 1100, 1110, 1210, 2000, 2010, 2100, 2110, 2210, 3010, 3110, 3200, 3201, 3202, 3210, 3211, 3212\}$

Fig. 2. A 4-variable MDD and the states $\mathcal{S}$ it encodes

through a sequence $(i_k, \ldots, i_l) \in \mathcal{S}_k \times \cdots \times \mathcal{S}_l$, for $K \geq k > l \geq 1$, is recursively defined as

$$\langle k|p\rangle[i_k, i_{k-1}, \ldots, i_l] = \langle k-1|\langle k|p\rangle[i_k]\rangle[i_{k-1}, \ldots, i_l].$$

The substates encoded by, or "below", $\langle k|p\rangle$ are then

$$\mathcal{B}(\langle k|p\rangle) = \{\beta \in \mathcal{S}_k \times \cdots \times \mathcal{S}_1 : \langle k|p\rangle[\beta] = 1\}$$

and those reaching, or "above", $\langle k|p\rangle$ are

$$\mathcal{A}(\langle k|p\rangle) = \{\alpha \in \mathcal{S}_K \times \cdots \times \mathcal{S}_{k+1} : \langle K|r\rangle[\alpha] = p\}.$$

For any node $\langle k|p\rangle$ on a path from the root $\langle K|r\rangle$ to $\langle 0|1\rangle$, the set of (global) states $\mathcal{A}(\langle k|p\rangle) \times \mathcal{B}(\langle k|p\rangle)$ is a subset of the set of (global) states encoded by the MDD, which we can write as $\mathcal{B}(\langle K|r\rangle)$ or $\mathcal{A}(\langle 0|1\rangle)$.

We reserve the indices 0 and 1 at each level k to encode the sets $\emptyset$ and $\mathcal{S}_k \times \cdots \times \mathcal{S}_1$, respectively. Such nodes do not need to be explicitly stored; this saves not only a small amount of memory, but also substantial execution time in the recursive manipulation algorithms. Fig. 2 shows a four-variable MDD and the set $\mathcal{S}$ it encodes (the node indices are in boldface and have been chosen arbitrarily, except for those with value 0 and 1). To see whether state (1,2,1,0) is encoded by this MDD, begin with the root node $\langle 4|5\rangle$ and follow the path: $\langle 4|5\rangle[1] = 4$, $\langle 3|4\rangle[2] = 7$, $\langle 2|7\rangle[1] = 4$, and $\langle 1|4\rangle[0] = 1$: since we reach the terminal node 1, the state is indeed encoded, i.e., $(1,2,1,0) \in \mathcal{B}(\langle 4|5\rangle)$. On the other hand, we can determine in a single operation that no state of the form $(0, i_3, i_2, i_1)$ is encoded by the MDD, since $\langle 4|5\rangle[0] = 0$ and, by convention, node $\langle 3|0\rangle$ is known to encode the empty set.

To illustrate how using quasi-reduced MDDs simplifies the manipulation algorithms, and how our convention on the meaning of the indices 0 and 1 improves efficiency, we show the pseudocode for performing a union operation on two MDDs, in Fig. 3. We use the types *level*, *index*, and *local* for MDD levels, MDD node indices within a level, and local state indices, respectively (in our implementation, these are simply integers in appropriate ranges). $\mathit{NewNode}(k)$ allocates a new MDD node $\langle k|x\rangle$ at level k, sets all its arcs $\langle k|x\rangle[i]$, for $i \in \mathcal{S}_k$, to 0, and returns its index x. $\mathit{SetArc}(k,p,i,y)$ sets $\langle k|p\rangle[i]$ to y; in addition, if the node $\langle k-1|x\rangle$ previously pointed to by $\langle k|p\rangle[i]$ does not have other incoming arcs, it is deleted. $\mathit{CheckIn}(k,p)$ searches the level-k *unique table* for a node

```
$\mathit{Union}(k : \mathit{level},\ p : \mathit{index},\ q : \mathit{index}) : \mathit{index}$
Return an index $u$ such that $\mathcal{B}(\langle k|u\rangle) = \mathcal{B}(\langle k|p\rangle) \cup \mathcal{B}(\langle k|q\rangle)$.

declare $u : \mathit{index},\ \ i : \mathit{local}$;
if $p = q$                      • $\mathcal{B}(\langle k|p\rangle) \cup \mathcal{B}(\langle k|p\rangle) = \mathcal{B}(\langle k|p\rangle)$
  return $p$;
else if $p = 1$ or $q = 1$      • exploit special meaning of $\langle k|1\rangle$
  return 1;
else if $p = 0$                 • exploit special meaning of $\langle k|0\rangle$
  return $q$;
else if $q = 0$                 • exploit special meaning of $\langle k|0\rangle$
  return $p$;
else if $k = 0$                 • terminal case, $p$ and $q$ are booleans
  return $p \vee q$;
if $\mathit{Cached}(\mathit{UNION}, k, p, q, u)$          • don't redo work
  return $u$;
$u \leftarrow \mathit{NewNode}(k)$;             • create a new node at level $k$
for $i = 0$ to $n_k - 1$
  $\mathit{SetArc}(k, u, i, \mathit{Union}(k-1, \langle k|p\rangle[i], \langle k|q\rangle[i]))$;
$u \leftarrow \mathit{CheckIn}(k, u)$;          • insert $\langle k|u\rangle$ in unique table
$\mathit{PutInCache}(\mathit{UNION}, k, p, q, u)$;    • remember result
return $u$;
```

Fig. 3. MDD union

$\langle k|q\rangle$ with the same n_k arc values (the hash key for the table) as p; if it finds such a node, it deletes the duplicate node $\langle k|p\rangle$ and returns q, otherwise it inserts $\langle k|p\rangle$ in the table and returns p. $\mathit{Cached}(\mathit{UNION}, k, p, q, u)$ searches the hash key "UNION, p, q" in the level-k *operation cache*, where UNION is an integer code from an enumerated type, like FIRE later on; if it finds the key, it sets u to the associated cached value and returns *true*, otherwise it returns *false*. $\mathit{PutInCache}(\mathit{UNION}, k, p, q, u)$ associates the value u to the key "UNION, k, p, q" in the level-k operation cache. In our implementation, we use dynamically-sized hash tables organized by levels for the caches, and dynamically-sized arrays to store nodes, so that $\langle k|p\rangle$ can be efficiently retrieved as the p^{th} entry of the k^{th} array (this index-based addressing of nodes is very efficient for quasi-reduced MDDs, as opposed to the more frequently used pointer-based addressing for fully-reduced MDDs).

2.4 MDDs for storing next-state functions

The next-state function $\mathcal{N}$ of the model can also be encoded using a decision diagram; since we need to represent transitions between states instead of single states, we need twice as many variables as the decision diagram encoding the state space.

In the case of MDDs, this means encoding the indicator function of the transition relation with a $2K$-variable MDD. Rather than numbering the variables from $2K$ down to 1, we distinguish them as "from" and "to" variables: we use the letters "i" and "j" for "from" and "to" local state or indices, unprimed and primed integers for "from" and "to" levels, and a double-bracket notation to distinguish these nodes from those of K-variable MDDs encoding sets of states. Thus, $\langle\langle 5'|p\rangle\rangle$ is a "to" node associated with the 5th submodel, and $\langle\langle 5'|p\rangle\rangle[j_5] = q$ means that the arc labeled with local state index j_5 from this node points to $\langle\langle 4|q\rangle\rangle$, a "from" node associated with the 4th submodel.

If the MDD encoding $\mathcal{N}$ is rooted at $\langle\langle K|t\rangle\rangle$, we have

$$(\mathbf{i}_K, \mathbf{j}_K, ..., \mathbf{i}_1, \mathbf{j}_1) \in \mathcal{B}(\langle\langle K|t\rangle\rangle) \Leftrightarrow (\mathbf{j}_K, ..., \mathbf{j}_1) \in \mathcal{N}(\mathbf{i}_K, ..., \mathbf{i}_1).$$

In principle, any permutation of the $2K$ variables could be used, but we have assumed that the variables are *interleaved*, that is, the "from" and "to" variables for a given level are next to each other[1]. There are at least two good reasons for this. First, it is widely acknowledged that interleaving is often the most efficient order in terms of nodes required to store the next-state function (of course, interleaving "defines" an order of the $2K$ variables only once the order of the K submodels, or the K variables of the MDDs storing sets of states, is fixed). Second, the symbolic state-space generation algorithms we consider next can be easily expressed using recursion if interleaving is used. Indeed, a very important third reason related to the above is presented in Sect. 3, where we argue that traditional symbolic approaches to represent $\mathcal{N}$ fail to exploit the large number of identity transformations (i.e., $i_k = i'_k$) present in the class of systems we address.

We conclude this section by recalling that, especially for asynchronous systems, it is often advantageous to store the next-state function as a *disjunction* [9]. If the events $\mathcal{E}$ are implied by the high-level model, as is clearly the case for Petri nets, where each *transition* α in the net corresponds to exactly one event in $\mathcal{E}$, we simply need to store each $\mathcal{N}_\alpha$ as a separate $2K$-variable MDD (of course, an MDD *forest* should be used, so that these MDDs can share common nodes [41]).

2.5 Breadth-first symbolic state-space generation

The simplest and most widely used symbolic state-space generation algorithm is shown in the top of Fig. 4. An alternative algorithm shown in the bottom of the same figure is conceptually simpler and closer to the definition of the state space $\mathcal{S}$ as a fixed-point. Both versions will halt after exactly as many steps as the maximum distance of any state from the initial state $\mathbf{s}$. The main computational difference is the cost of applying the next-state function; the traditional algorithm applies $\mathcal{N}$ *only to the unexplored states* $\mathcal{U}$ which, at iteration d, consists of all states at distance *exactly* d from $\mathbf{s}$, while the alternative algorithm applies $\mathcal{N}$ to *all known states*, that is, all states at distance *at most* d from $\mathbf{s}$.

While it would be wasteful to apply the next-state function to the same state more than once in an explicit setting, the cost of applying $\mathcal{N}$ to a set of states encoded as an MDD in a symbolic setting depends on the *number of nodes in the MDD*, not on the *number of states encoded by the MDD*, so it is not clear that the traditional algorithm is better, nor why the alternative version appears to have been neglected in the literature. Indeed, Sect. 6 reports experimental evidence suggesting that the set $\mathcal{U}$ of states having exactly distance d is often not a "nice" set to encode as an MDD (furthermore, the traditional version of the breadth-first generation algorithm incurs the additional cost of the set-difference operation required to compute the "truly new states" $\mathcal{X} \setminus \mathcal{S}$). However, our main motivation for discussing *AllBfSsGen* is that it introduces a different way of applying the next-state function, i.e., one that does not distinguish between new and old states. This is one of the characteristics of the algorithms we introduce in Sect. 3.

A second variation of symbolic breadth-first state-space generation, which introduces the idea of *chaining* [43], is shown in the top of Fig. 5. Its key advantage lies in that the states in $\mathcal{N}_\alpha(\mathcal{U})$, i.e., the (possibly) new states reachable in one firing of event α from the currently unexplored states $\mathcal{U}$, are added to $\mathcal{U}$ right away, instead of only after having explored each event in isolation from the same original $\mathcal{U}$. In other words, if $\mathbf{j}^{(1)} \in \mathcal{N}_{\alpha^{(1)}}(\mathbf{i})$, $\mathbf{j}^{(2)} \in \mathcal{N}_{\alpha^{(2)}}(\mathbf{j}^{(1)}), \ldots, \mathbf{j}^{(d)} \in \mathcal{N}_{\alpha^{(d)}}(\mathbf{j}^{(d-1)})$, and if the for-loop considers the events in the order $\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(d)}$, *ChSsGen* finds all the states in this path in a single iteration, while both *BfSsGen* and *AllBfSsGen* would require d iterations. Note that statement 6 in *ChSsGen*, just like statement 6 in *AllBfSsGen*, applies the next-state function to the same states more than once; however, it does so only to states in the set $\mathcal{U}$ being built; the difference between explored and unexplored states is still kept in the outer while-loop. An algorithm merging the ideas in *AllBfSsGen* and *ChSsGen* is shown at the bottom of Fig. 5: we believe that it is the most straightforward state-space generation algorithm when $\mathcal{N}$ is disjunctively partitioned into events, and the results of Sect. 6 show that its performance is competitive with respect to that of *BfSsGen*, *AllBfSsGen*, and *ChSsGen*.

ChSsGen and *AllChSsGen* are fundamentally different from *BfSsGen* and *AllBfSsGen* in one way: *they do not find states in strict breadth-first order*. This is a second characteristic of the algorithms we introduce in Sect. 3, where we also discuss the importance of the order in which events are considered[2].

2.6 Running example

As a running example, consider Fig. 6, showing a simplified producer-consumer system represented as a Petri

[1] We assume that level k is above level k', since we store the *forward* next-state function $\mathcal{N}$. However, symbolic model-checking [8] also makes use of the *backward* or *previous-state* function $\mathcal{N}^{-1}$ and, in this case, we would have level k' above level k.

[2] The paper that introduces chaining [43] uses a *topological sort* on the gates of the circuit, modeled as Petri net transitions.

```
BfSsGen(s : state, 𝒩 : stateset → 2^stateset) : stateset
Compute $\mathcal{S} \subseteq \widehat{\mathcal{S}}$ starting from s by repeatedly applying $\mathcal{N}$ in
breadth-first fashion. All sets are stored using MDDs.
1 declare $\mathcal{S},\mathcal{U},\mathcal{X}$ : stateset;
2 $\mathcal{S} \leftarrow \{\mathbf{s}\}$;                    • known states
3 $\mathcal{U} \leftarrow \{\mathbf{s}\}$;                    • unexplored known states
4 while $\mathcal{U} \neq \emptyset$ do                        • there are still unexplored states
5   $\mathcal{X} \leftarrow \mathcal{N}(\mathcal{U})$;         • possibly new states
6   $\mathcal{U} \leftarrow \mathcal{X} \setminus \mathcal{S}$; • truly new states
7   $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{U}$;
8 return $\mathcal{S}$;
```

```
AllBfSsGen(s : state, 𝒩 : stateset → 2^stateset) : stateset
1 declare $\mathcal{S},\mathcal{O}$ : stateset;
2 $\mathcal{S} \leftarrow \{\mathbf{s}\}$;                    • known states
3 $\mathcal{O} \leftarrow \emptyset$;                         • old states
4 while $\mathcal{O} \neq \mathcal{S}$ do                     • test for the fixed point
5   $\mathcal{O} \leftarrow \mathcal{S}$;                     • save old state space
6   $\mathcal{S} \leftarrow \mathcal{O} \cup \mathcal{N}(\mathcal{O})$; • apply $\mathcal{N}$ to entire state space
7 return $\mathcal{S}$;
```

Fig. 4. Two symbolic breadth-first generation algorithms

```
ChSsGen(s : state, 𝒩_{α∈ℰ} : stateset → 2^stateset) : stateset
Compute $\mathcal{S} \subseteq \widehat{\mathcal{S}}$ starting from s by incrementally applying
each $\mathcal{N}_\alpha$. All sets are stored using MDDs.
1 declare $\mathcal{S},\mathcal{U}$ : stateset;
2 $\mathcal{S} \leftarrow \{\mathbf{s}\}$;                    • known states
3 $\mathcal{U} \leftarrow \{\mathbf{s}\}$;                    • unexplored known states
4 while $\mathcal{U} \neq \emptyset$ do                        • there are still unexplored states
5   for each $\alpha \in \mathcal{E}$ do
6     $\mathcal{U} \leftarrow \mathcal{U} \cup \mathcal{N}_\alpha(\mathcal{U})$; • add to possibly new states
7   $\mathcal{U} \leftarrow \mathcal{U} \setminus \mathcal{S}$; • truly new states
8   $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{U}$;
9 return $\mathcal{S}$;
```

```
AllChSsGen(s:state, 𝒩_{α∈ℰ}:stateset → 2^stateset): stateset
1 declare $\mathcal{S},\mathcal{O}$ : stateset;
2 $\mathcal{S} \leftarrow \{\mathbf{s}\}$;                    • known states
3 $\mathcal{O} \leftarrow \emptyset$;                         • old states
4 while $\mathcal{O} \neq \mathcal{S}$ do                     • test for the fixed point
5   $\mathcal{O} \leftarrow \mathcal{S}$;                     • save old state space
6   for each $\alpha \in \mathcal{E}$ do
7     $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{N}_\alpha(\mathcal{S})$; • apply $\mathcal{N}_\alpha$ to entire state space
8 return $\mathcal{S}$;
```

Fig. 5. Symbolic breadth-first generation with chaining

net with one producer in place p anc one consumer in place s. The model has eight reachable (global) states:

$$\mathcal{S} = \{(p^1q^0r^0s^1t^0), (p^1q^0r^0s^0t^1), (p^0q^1r^1s^1t^0), (p^0q^1r^1s^0t^1), (p^1q^0r^1s^1t^0), (p^1q^0r^1s^0t^1), (p^0q^1r^0s^1t^0), (p^0q^1r^0s^0t^1)\}.$$

We decompose the model into the three submodels in Fig. 6, each corresponding to exactly one non-terminal level of the MDD. The reachable local states of the example are shown in Fig. 7. Submodel 1 has two local states, submodel 2 has four, and submodel 3 has two.

Figure 8 illustrates the MDD encodings of the sets of states $\mathcal{S}$ and $\mathcal{U}$ generated at each iteration of algorithm *BfSsGen*. The MDD encoding the initial state $\mathbf{s}$ is shown in frame (a).

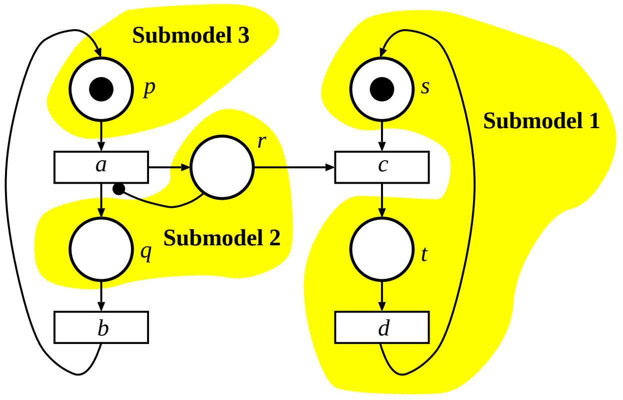


Fig. 6. A producer-consumer model and its submodels

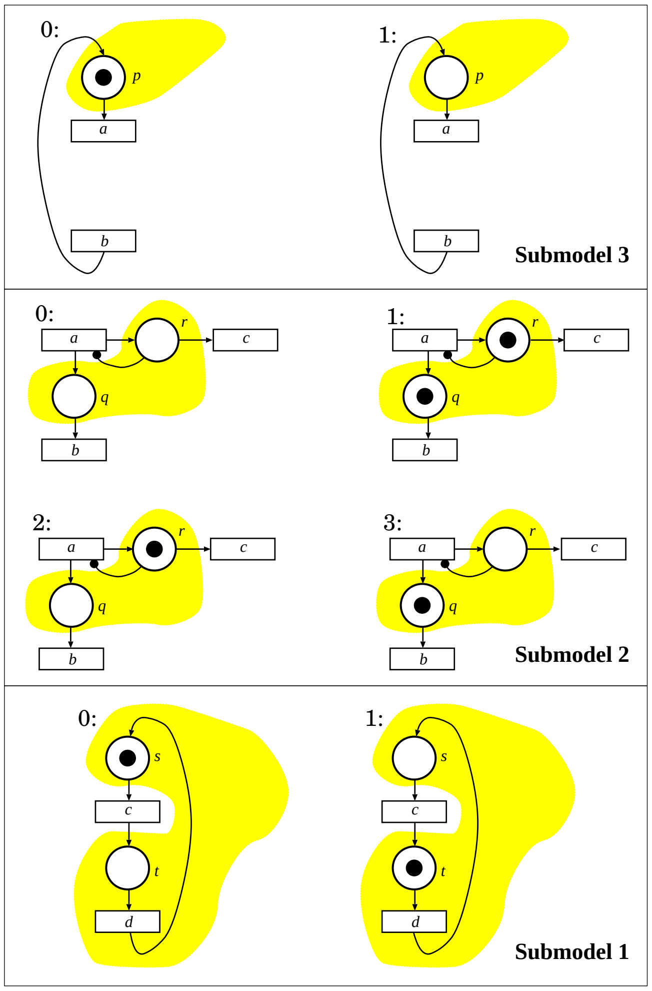


Fig. 7. The local states of each submodel

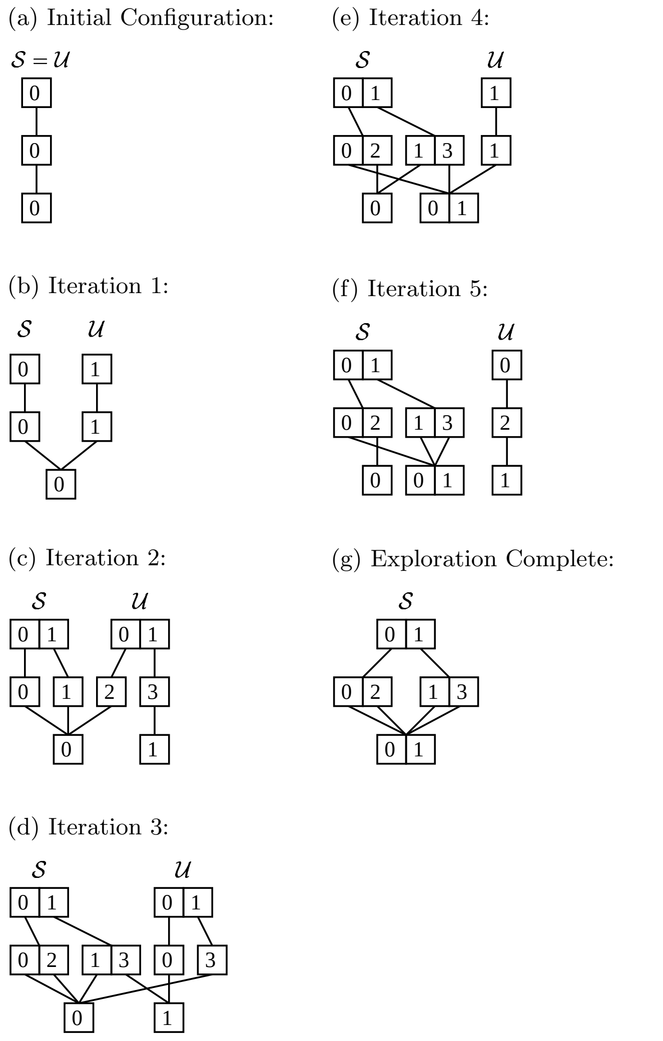


Fig. 8. Breadth-first generation of the state space

3 A new approach

The encoding of sets of states using MDDs instead of BDDs has no immediate advantage in itself, since the two data structures are very similar, except for the number of levels. However, the advantages of MDDs become apparent when we consider our encoding of the next-state function and how this encoding is exploited in fixed-point computations, which we discuss next.

3.1 Kronecker encoding of the next-state function

We adopt a representation of $\mathcal{N}$ inspired by work on Markov chains, where the infinitesimal generator matrix of a large continuous-time Markov chain is encoded through a *Kronecker algebra expression* [2] on a set of small matrices [7,40]. In our setting, this corresponds to a two-dimensional decomposition of $\mathcal{N}$: by events and by submodels. This is possible when the decomposition of the model into submodels is *Kronecker consistent*, that is, when we can find $K \cdot |\mathcal{E}|$ functions $\mathcal{N}_{k,\alpha} : \mathcal{S}_k \to 2^{\mathcal{S}_k}$, describing the (local) effect of a particular event α on a particular subsystem k. Formally, this means that, for each event $\alpha \in \mathcal{E}$ and (global) state $(i_K, \ldots, i_1) \in \widehat{\mathcal{S}}$,

$$\mathcal{N}_\alpha(i_K, \ldots, i_1) = \mathcal{N}_{K,\alpha}(i_K) \times \ldots \times \mathcal{N}_{1,\alpha}(i_1).$$

Thus, it must be possible to express the effect of firing an event α in a global state as the cross product of the local effects of α on each submodel.

The local next-state functions $\mathcal{N}_{k,\alpha}$ can be encoded as incidence matrices $\mathbf{N}_{k,\alpha} \in \{0,1\}^{n_k \times n_k}$, where

$$\mathbf{N}_{k,\alpha}[i_k, j_k] = 1 \;\Leftrightarrow\; j_k \in \mathcal{N}_{k,\alpha}(i_k).$$

Thus, event α is locally enabled in local state $i \in \mathcal{S}_k$ of the k^{th} submodel, $\mathcal{N}_{k,\alpha}(i) \neq \emptyset$, iff not all entries of the i^{th} row in $\mathbf{N}_{k,\alpha}$ are zero, $\mathbf{N}_{k,\alpha}[i,\cdot] \neq \mathbf{0}$.

The overall next-state function $\mathcal{N}$ is then encoded as the incidence matrix given by the (boolean) sum of Kronecker products

$$\mathbf{N} = \sum_{\alpha \in \mathcal{E}} \bigotimes_{K \geq k \geq 1} \mathbf{N}_{k,\alpha}$$

The $\mathbf{N}_{k,\alpha}$ matrices are extremely sparse (for Petri nets, each row contains at most one nonzero entry), and are indexed using the same mapping ψ_k used to index $\mathcal{S}_k$.

For some formalisms, such as ordinary Petri nets, the Kronecker consistency requirement is always satisfied, regardless of how the model is partitioned into submodels. In other formalisms, not all decompositions will be Kronecker consistent; however, we can always find a consistent decomposition by refining the events or coarsening the partition into submodels. Consider for example Fig. 9, where a model consists of two boolean variables x_1 and x_2 and the partition assigns each variable to a different submodel. If an event α swaps the values of x_1 and x_2 but is enabled only when $x_1 \neq x_2$, the partition is not Kronecker consistent. If it were, $\mathcal{N}_{2,\alpha}$ and $\mathcal{N}_{1,\alpha}$ would have to satisfy $1 \in \mathcal{N}_{2,\alpha}(0)$ and $1 \in \mathcal{N}_{1,\alpha}(0)$. However, Kronecker consistency would also imply that $(1,1) \in \mathcal{N}_{2,\alpha}(0) \times \mathcal{N}_{1,\alpha}(0) = \mathcal{N}_\alpha(0,0)$, while a transition from state $(0,0)$ to state $(1,1)$ should not be allowed.

To achieve Kronecker consistency, we can then either split event α into α' and α'', which allows us to treat the two enabling states, $(0,1)$ and $(1,0)$, separately, or merge the two levels into a single level with four possible local states.

3.2 An identity crisis

Event α is said to be *independent* of level k if $\mathbf{N}_{k,\alpha} = \mathbf{I}$, the identity matrix, that is, if its occurrence is not affected by, nor it affects, the local state of the k^{th} submodel. As we will see, our approach neither stores nor processes these identity matrices; this results in large savings especially when dealing with asynchronous systems, where most of the $\mathbf{N}_{k,\alpha}$ matrices are identities.

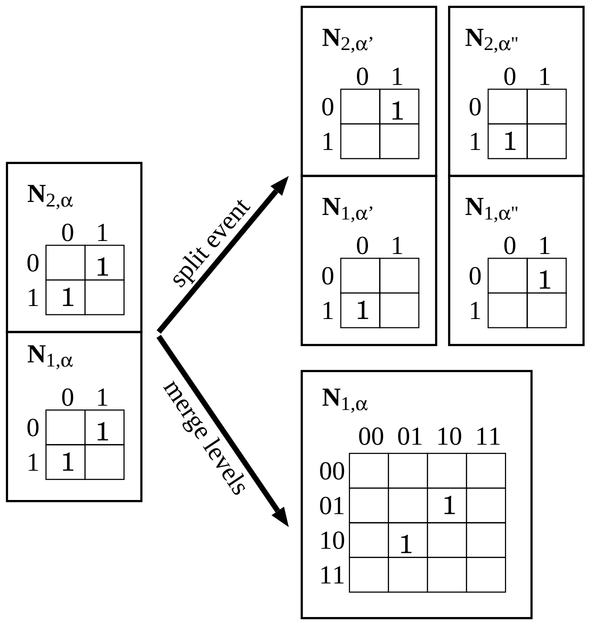


Fig. 9. Satisfying Kronecker-consistency requirements

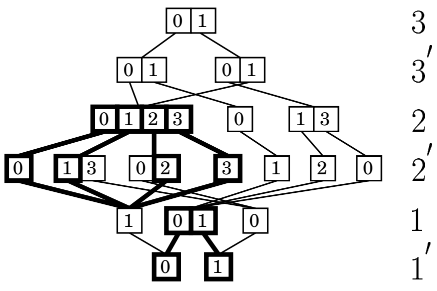
On the other hand, a decision diagram representation of $\mathcal{N}$ does not efficiently encode or exploit these identities. The variables (levels) not affected by an event α are still represented in the decision diagram by "identity patterns". These are wasteful both in terms of memory, as they require four nodes per variable along each corresponding path in the diagram, and in terms of time, as these additional nodes are processed during the image computation, just like any other node. The compactness of a BDD representation is characterized by the degree of merging of duplicate nodes and the number of redundant nodes that are eliminated and replaced with arcs skipping levels. However, when representing $\mathcal{N}$ with a BDD, an arc that skips levels k and k' (the "from" and "to' variables corresponding to the k^{th} submodel, respectively) means that, after an event fires, the k^{th} component can be either 0 or 1, regardless of whether it was 0 or 1 before the firing. The more natural behavior is instead the one where a 0 remains a 0 and a 1 remains a 1, the default in our Kronecker encoding.

Fig. 10 shows Kronecker, monolithic MDD, and partitioned MDD representations of the next state function for our running example. For the Kronecker representation, the total number of nonzero entries is nine and five out of twelve Kronecker matrices are identities. This is not a remarkable ratio because the model is small. For many asynchronous systems, however, the number of events grows linearly in K but most, if not all, events are described by a constant number (i.e., independent of K) of non-identity matrices; thus, while the Kronecker description potentially requires $O(K^2)$ matrices, $O(K)$ matrices suffice in practice, a huge improvement. On the other hand, the monolithic MDD encoding of $\mathcal{N}$, requires 19 nodes, while the partitioned MDD encoding requires

As Kronecker matrices:

	a	b	c	d
3	0:1	1:0	**I**	**I**
2	0:1	1:2,3:0	1:3,2:0	**I**
1	**I**	**I**	0:1	1:0

As a monolithic MDD:



As a disjunctions of MDDs:

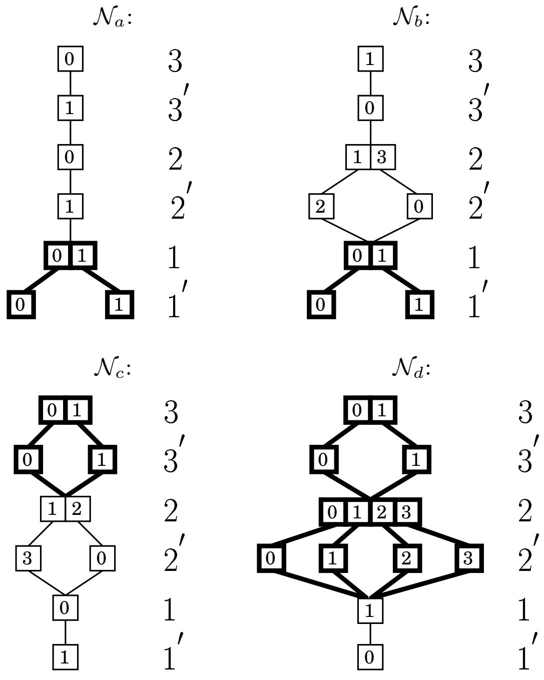


Fig. 10. Kronecker vs. MDD encoding of $\mathcal{N}$

8 nodes each for $\mathcal{N}_a$, $\mathcal{N}_b$, and $\mathcal{N}_c$, and 10 nodes for $\mathcal{N}_d$ (in total, only $8+8+8+10-3=8+31$ nodes are required, since the three nodes at levels 1 and $1'$ for $\mathcal{N}_a$ and $\mathcal{N}_b$ can be shared).

One reason for the large number of nodes in the MDD encoding is the explicit representation of identity transformations, corresponding to patterns where a "from" node $\langle k|p\rangle$ with n_k distinct children $\langle k'|q^{(i)}\rangle$, such that $\langle k'|q^{(0)}\rangle[0] = \cdots = \langle k'|q^{(n_k-1)}\rangle[n_k-1]$, shown with bold lines in the figure. In the partitioned MDD case, these patterns are clearly visible because all other arcs $\langle k'|q^{(i)}\rangle[j]$, for $j \neq i$, have value 0. In the monolithic MDD case, some of these patterns are harder to recognize because the level-k' nodes may have additional nonzero arcs, as is the case for the identity pattern at levels 2 and $2'$ in our figure, while other identity patterns are destroyed when performing the union of the next-state functions for multiple events, as is the case for the identity patterns at levels 3 and $3'$ of $\mathcal{N}_c$ and $\mathcal{N}_d$. However, these transformations are still encoded in the MDD, and they can occupy a non-negligible amount of memory.

3.3 Event locality

In large globally-asynchronous locally-synchronous systems, most events are independent of most levels. Thus, in addition to efficiently representing $\mathcal{N}$, the Kronecker encoding is also able to clearly evidence *event locality* [13,36], that is, identify which events affect which levels.

Let $Top(\alpha)$ and $Bot(\alpha)$ denote the highest and lowest levels on which α depends, that is:

$$Top(\alpha)=\max\{k:\mathbf{N}_{k,\alpha}\neq\mathbf{I}\},\ Bot(\alpha)=\min\{k:\mathbf{N}_{k,\alpha}\neq\mathbf{I}\}.$$

If events that are independent of all levels (hence do not affect the model's behavior) are ignored, these maximum and minimum levels exist, that is:

$$\forall\alpha\in\mathcal{E},\quad K\geq Top(\alpha)\geq Bot(\alpha)\geq 1.$$

We can then partition $\mathcal{E}$ into K classes,

$$\mathcal{E}_k=\{\alpha\in\mathcal{E}\ :\ Top(\alpha)=k\},\qquad \text{for } K\geq k\geq 1$$

where some $\mathcal{E}_k$ might be empty.

In [13], we showed how to use this locality information to avoid unnecessary work. When firing an event α, we do not start at the root of the MDD, since we know that all state components from K down to $Top(\alpha)+1$ are not going to change anyway. Rather, we "jump-in-the-middle" of the MDD by directly accessing each node at level $k=Top(\alpha)$ and we fire α in it and, recursively, on its descendants, up to nodes at level $l=Bot(\alpha)$ (of course, this requires that MDD nodes be organized and stored in such a way that provides easy access by level). Conceptually, for each node $\langle k|p\rangle$, we can compute a node $\langle k|f\rangle$ encoding the effect of firing α in it:

$$\mathcal{B}(\langle k|f\rangle)=\mathcal{N}_{k,\alpha}\times\cdots\times\mathcal{N}_{l,\alpha}\times\underbrace{\mathcal{N}_{l-1,\alpha}\times\cdots\times\mathcal{N}_{1,\alpha}}_{\text{identity functions}}(\mathcal{B}(\langle k|p\rangle)).$$

Then, we can *substitute* node $\langle k|p\rangle$ with a node $\langle k|u\rangle$, where u is the value returned by the call $Union(k,p,f)$, that is,

$$\mathcal{B}(\langle k|u\rangle)=\mathcal{B}(\langle k|p\rangle)\cup\mathcal{B}(\langle k|f\rangle).$$

This is correct because, if $\mathbf{i}\in\mathcal{S}$ and $\mathbf{j}\in\mathcal{N}_\alpha(\mathbf{i})$, we can conclude that $\mathbf{j}\in\mathcal{S}$ and, in particular, because, if $\mathbf{i}\in\mathcal{A}(\langle k|p\rangle)\times\mathcal{B}(\langle k|p\rangle)$, then $\mathbf{j}\in\mathcal{A}(\langle k|p\rangle)\times\mathcal{B}(\langle k|f\rangle)$ and substituting $\langle k|p\rangle$ with $\langle k|u\rangle$ will indeed add any such state $\mathbf{j}$.

The efficiency of jump-in-the-middle is due to two factors. First, we avoid work in nodes at levels from K to $k+1$. Second, any algorithm that fires α starting from the MDD root would reach $\langle k|p\rangle$ and attempt to fire α in it as many times as the number of incoming arcs pointing to $\langle k|p\rangle$ from level $k+1$ (of course, all but the first time the effect of this firing is retrieved from the operation cache, but the cost of these cache lookups is not negligible). With jump-in-the-middle, instead, we only need to consider $\langle k|p\rangle$ once.

One complication with this approach, however, is that substituting $\langle k|p\rangle$ with $\langle k|u\rangle$ requires us to *redirect* all incoming arcs so that they point to $\langle k|u\rangle$ instead of $\langle k|p\rangle$. Thus, in [13], we pair jump-in-the-middle with another improvement, *in-place updates* of MDD nodes, which not only avoids the explicit computation of node $\langle k|f\rangle$, but also greatly reduces the need to redirect arcs from nodes at level $k+1$.

3.4 In-place updates

The idea of substituting $\langle k|p\rangle$ with $\langle k|u\rangle$ we just discussed can be further improved by realizing that, instead of computing the result of firing α in $\langle k|p\rangle$ as a separate node $\langle k|f\rangle$ and then computing the union node $\langle k|u\rangle$, we can incrementally *modify* the arcs of node $\langle k|p\rangle$, so that, when we are done, they point to the same nodes that the arcs of node $\langle k|u\rangle$ would point, had we built it explicitly.

This is described in Fig. 11, which provides the pseudocode for functions *Fire* and *RecFire*. Function *Fire* is called to fire event α in nodes $\langle k|p\rangle$, with $k=Top(\alpha)$, and it updates its arcs $\langle k|p\rangle[j]$ in place. The recursive function *RecFire* is called to fire α on nodes $\langle l|q\rangle$, with $Top(\alpha)>l\geq Bot(\alpha)$, but it does not modify these nodes. Instead, it creates a node $\langle l|s\rangle$ that encodes the local effect of firing α on node $\langle l|q\rangle$ and its descendants. The calls $Cached(FIRE,l,\alpha,q,s)$ and $PutInCache(FIRE,l,\alpha,q,s)$ are exactly analogous to those used in *Union*, except that, when the keyword is *FIRE*, the search key is now an event, α and a node index, q, instead of the two nodes required for a union. It should be noted that only the firing of an event α on nodes at levels *below* $Top(\alpha)$ are cached. This is because, by updating node $\langle k|p\rangle$ in place, we keep changing (increasing) the set of states it encodes, thus the cached value for the effect of firing α in $\langle k|p\rangle$ becomes obsolete after every update. Note that, in a traditional approach without in-place updates, we need instead to cache this information merely only to deal with the situation when $\langle k|p\rangle$ has multiple incoming arcs; each in-place update in our case is equivalent to creating a new node (or a sequence of new nodes, if *Fire* uses pipelining) in the traditional approach, and thus there would be no cached value for such nodes, unless they are duplicates of existing nodes.

By using in-place updates, *Fire* achieves a chaining effect: if two local states i and i' "locally enable" event α, i.e., $\mathbf{N}_{k,\alpha}[i,\cdot]\neq\mathbf{0}$ and $\mathbf{N}_{k,\alpha}[i',\cdot]\neq\mathbf{0}$, if $\mathbf{N}_{k,\alpha}[i,i']=1$, and if i is picked before i' in statement 4, the effect of firing α in $\langle k|p\rangle[i]$ has already been incorporated in the updated node $\langle k|p\rangle$ that is, in the node pointed by $\langle k|p\rangle[j]$, by the time j is picked. Thus, $\langle k|p\rangle[j]$ will include the effect of firing α, zero, one, or two times. This *pipelining* can be carried even further by adding the last two statements in *Fire*, marked optional in the pseudocode: if we add the new local states j back to the set $\mathcal{L}$ of local states to explore (statement 12), the order in which the elements in $\mathcal{L}$ are picked does not matter, and the result is a "full" pipelining: event α is fired in the local states of $\langle k|p\rangle$ as long as doing so discovers new states. Therefore, the call $Fire(\alpha, k, p)$ updates $\langle k|p\rangle$ in-place so that it encodes $\mathcal{N}^*_\alpha(\mathcal{B}(\langle k|p\rangle))$, where, for notational convenience, we interpret the application of a global next-state function to a set of substates in the obvious way: $\mathcal{N}_\alpha(\mathcal{B}(\langle k|p\rangle))$ really means $\mathcal{N}_{k,\alpha} \times \cdots \times \mathcal{N}_{1,\alpha}(\mathcal{B}(\langle k|p\rangle))$.

In addition to creating and destroying fewer nodes, in-place updates have the advantage that the node $\langle k|p\rangle$ being updated is seldom deleted, so that its incoming arcs do not need to be redirected (in other words, if the value of $\langle k+1|q\rangle[i]$ was p before a call $Fire(\alpha, k, p)$, the value should remain p). Redirection is needed only when, after having been modified in place, node $\langle k|p\rangle$ becomes a duplicate of an existing node $\langle k|d\rangle$. In this case, $\langle k|p\rangle$ must be deleted and its incoming arcs must be redirected to $\langle k|d\rangle$ (in other words, we must set $\langle k+1|q\rangle[i]$ to d). In [13], we do so by using either *upstream arcs* or *forwarding arcs*. In the former case, we actually maintain pointers from each node $\langle k|p\rangle$ to any node $\langle k+1|q\rangle$ having pointers to it. In the latter case, we place an arc in the node $\langle k|p\rangle$ to be deleted, pointing to its active duplicate $\langle k|d\rangle$; $\langle k|p\rangle$ can then be actually deleted only once all the references to it have been forwarded to $\langle k|d\rangle$.

```
Fire(α : event, k : level, p : index) : index
Fire α in ⟨k|p⟩, where k = Top(α), using in-place updates.
When pipelining is used, return x such that B(⟨k|x⟩) = N*_{k,α}(B(⟨k|p⟩)).
 1 declare f, u : index,  i, j : local,  L : set of local;
 2 L ← {i_k ∈ S_k : ⟨k|p⟩[i_k] ≠ 0 ∧ N_{k,α}[i_k, ·] ≠ 0};
 3 while L ≠ ∅ do
 4   pick and remove i from L;
 5   f ← RecFire(α, k−1, ⟨k|p⟩[i]);
 6   if f ≠ 0 then
 7     foreach j s.t. N_{k,α}[i, j] = 1 do
 8       u ← Union(k−1, f, ⟨k|p⟩[j]);
 9       if u ≠ ⟨k|p⟩[j] then
10         ⟨k|p⟩[j] ← u;
11         if N_{k,α}[j, ·] ≠ 0 then          • optional: for pipelining
12           L ← L ∪ {j};                     • optional: for pipelining
13 p ← CheckIn(k, p);              • reinsert ⟨k|p⟩ in unique table
14 return p;
```

```
RecFire(α : event, l : level, q : index) : index
Recursively fire α in ⟨l|q⟩, where l < Top(α).
 1 declare f, u, s : index,  i, j : local,  L : set of local;
 2 if l < Bot(α) then return q;
 3 if Cached(FIRE, l, α, q, s) then return s;
 4 s ← NewNode(l);
 5 L ← {i_l ∈ S_l : ⟨l|q⟩[i_l] ≠ 0 ∧ N_{l,α}[i_l, ·] ≠ 0};
 6 while L ≠ ∅ do
 7   pick and remove i from L;
 8   f ← RecFire(α, l−1, ⟨l|q⟩[i]);
 9   if f ≠ 0 then
10     foreach j s.t. N_{l,α}[i, j] = 1 do
11       u ← Union(l−1, f, ⟨l|s⟩[j]);
12       if u ≠ ⟨l|s⟩[j] then
13         ⟨l|s⟩[j] ← u;
14 s ← CheckIn(l, s);                  • insert ⟨l|s⟩ in unique table
15 PutInCache(FIRE, l, α, q, s);                   • remember result
16 return s;
```

Fig. 11. Firing events with in-place-updates

3.5 Saturation

An important advantage of using decision diagrams to encode $\mathcal{N}$ lies in the ability to obtain an entire set of new states reachable from the current set through a single symbolic step. This approach works well in practice for the analysis of mostly synchronous systems. However, in many practical examples, the "state space explosion" phenomenon that hampers explicit techniques, reshapes itself in the form of a new obstacle for symbolic methods: "the BDD node explosion". This usually manifests midway through symbolic exploration, when the number of nodes in the decision diagrams, especially for levels in the middle section, grows extremely fast, before contracting to the final configuration, which is usually much more compact. The *peak size* of the decision diagram (measured in bytes or nodes) is frequently hundreds or thousands of times larger than the *final size*, placing enormous strain on the storage and computational resources.

A fundamental goal of any exploration strategy, then, should be a small peak-to-final size ratio. Attempts have been made in this direction, but most of them are still tied to the idea of a breadth-first search through the use of global iterations and have had limited success in reducing peak memory requirements [10,42,44]. Our approach starts by identifying nodes that contribute to the peak MDD size and are eventually eliminated, as opposed to nodes that are still (likely to be) present in the final MDD. To this end, we define a restriction of the next-state function to the set of events affecting level k and below:

$$\mathcal{N}_{\leq k} = \bigcup_{1 \leq l \leq k} \mathcal{N}_{\mathcal{E}_l} = \bigcup_{\alpha : Top(\alpha) \leq k} \mathcal{N}_\alpha$$

Then, we define the following fundamental idea.

Definition 1. An MDD node $\langle k|p\rangle$ at level k is said to be *saturated* if it represents a fixed point with respect to the firing of any event that affects only levels k and below: $\mathcal{B}(\langle k|p\rangle) = \mathcal{N}^*_{\leq k}(\mathcal{B}(\langle k|p\rangle))$.

Theorem 1. A node can appear in the final MDD representation of the state space $\mathcal{S}$ only if it is saturated.

Proof. By contradition, assume that the MDD encoding of $\mathcal{S}$, rooted at $\langle K|r\rangle$, contains a non–saturated node $\langle k|p\rangle$, i.e., $\mathcal{B}(\langle k|p\rangle)$ is a strict subset of $\mathcal{N}^*_{\leq k}(\mathcal{B}(\langle k|p\rangle))$. This means that $\mathcal{N}_{\leq k}(\mathcal{B}(\langle k|p\rangle))$ contains states not in $\mathcal{B}(\langle k|p\rangle)$, since, if $\mathcal{N}_{\leq k}(\mathcal{B}(\langle k|p\rangle)) \subseteq \mathcal{B}(\langle k|p\rangle)$, we would have $\mathcal{N}^t_{\leq k}(\mathcal{B}(\langle k|p\rangle)) \subseteq \mathcal{B}(\langle k|p\rangle)$ for all t, thus it could not be that $\mathcal{B}(\langle k|p\rangle) \subset \mathcal{N}^*_{\leq k}(\mathcal{B}(\langle k|p\rangle))$. In other words, there is an event $\alpha \in \cup_{l=1}^{k}\mathcal{E}_l$ such that its firing in a state $\mathbf{i} \in \mathcal{A}(\langle k|p\rangle) \times \mathcal{B}(\langle k|p\rangle)$ can lead to a state $\mathbf{j} = (i_K, ..., i_{k+1}, j_k, ..., j_1)$ such that $(j_k, ..., j_1) \notin \mathcal{B}(\langle k|p\rangle)$. However, this is a contradiction, since $\mathbf{j}$ is reachable, being reached by firing α in $\mathbf{i}$, a reachable state, but it is not encoded by the MDD, since $\langle K|r\rangle[i_K, \ldots, i_{k+1}] = \langle k|p\rangle$ and $(j_k, \ldots, j_1) \notin \mathcal{B}(\langle k|p\rangle)$. □

Note that being saturated is a necessary, but not sufficient, condition for a node to appear in the final MDD encoding of $\mathcal{S}$, and that it can be shown by contradiction that any MDD node reachable from a saturated node must be saturated as well. Also, since $\mathcal{N}_{\leq K}$ is simply $\mathcal{N}$, we can conclude that

$$\mathcal{B}(\langle K|r\rangle) = \mathcal{N}^*(\mathcal{B}(\langle K|r\rangle)),$$

when the root $\langle K|r\rangle$ of the MDD is saturated. Furthermore, if $\mathbf{s} \in \mathcal{B}(\langle K|r\rangle)$, we have $\mathcal{B}(\langle K|r\rangle) \supseteq \mathcal{N}^*(\mathbf{s})$. Thus, if we initialize the root node $\langle K|r\rangle$ so that it encodes exactly the initial state, $\mathcal{B}(\langle K|r\rangle) = \{\mathbf{s}\}$, and if we saturate $\langle K|r\rangle$ without ever adding any unreachable state in $\widehat{\mathcal{S}}\backslash\mathcal{S}$ to $\mathcal{B}(\langle K|r\rangle)$, the final result will satisfy $\mathcal{B}(\langle K|r\rangle) = \mathcal{S}$.

This is the idea behind our saturation algorithm [14]. We greedily transform unsaturated nodes into saturated ones in a way that minimizes the number of unsaturated nodes present in the MDD: when working on a node at level k, only one MDD node per level is unsaturated, at levels K through k. We fire events node-wise and exhaustively, instead of level-wise and just once per iteration, and in a specific *saturation order* that guarantees that, when saturating a node at level k, all nodes at levels below k are already saturated. In other words, starting from the MDD encoding the initial state, we first fire all events in $\mathcal{E}_1$ exhaustively in the node at level 1. Then, we saturate the node at level 2, by repeatedly firing all events in $\mathcal{E}_2$. If this creates nodes at level 1, they are immediately saturated, by firing all events in $\mathcal{E}_1$ on them. Then, we saturate the node at level 3, by exhaustively firing all events in $\mathcal{E}_3$. If this creates nodes at levels 2 or 1, we saturate them immediately, using the corresponding set of events, and so on. The algorithm halts when we have saturated the root node at level K.

While we assume a single initial state $\mathbf{s}$, thus a single unsaturated node per level in the initial MDD, it is trivial to extend the algorithm to the case where there is a set of initial states encoded by an arbitrary MDD; indeed, this is what we implement in our tool SMART [12]. Also in this case, the number of unsaturated nodes never increases beyond the number present at the beginning. Thus, except for those initially unsaturated nodes, only saturated nodes contribute to the peak size of the MDD. These saturated nodes are not guaranteed to be present in the final MDD, since they might become disconnected when arcs pointing to them are redirected to other saturated nodes encoding even larger sets, but they have at least a chance of being present in the final MDD, since they satisfy the necessary condition: they are saturated.

Pseudocode for the saturation algorithm is shown in Fig. 12. Just as in traditional symbolic state-space generation algorithms, we use a *unique table* to detect duplicate nodes, and *operation caches*, in particular a *union cache* and a *firing cache*, to speed-up computation. In our approach, however, only saturated nodes are checked in the unique table or referenced in the caches. In particular, the *Union* function of Fig. 3 behaves correctly without having to know whether the nodes it operates upon are saturated or not. This is because the node encoding the union of two saturated nodes is saturated by definition:

$$\mathcal{B}(\langle k|p\rangle) = \mathcal{N}^*_{\leq k}(\mathcal{B}(\langle k|p\rangle)) \;\wedge\; \mathcal{B}(\langle k|q\rangle) = \mathcal{N}^*_{\leq k}(\mathcal{B}(\langle k|q\rangle))$$
$$\Rightarrow\; \mathcal{B}(\langle k|p\rangle) \cup \mathcal{B}(\langle k|q\rangle) = \mathcal{N}^*_{\leq k}(\mathcal{B}(\langle k|p\rangle) \cup \mathcal{B}(\langle k|q\rangle)).$$

The saturation strategy has several important properties. It is very flexible in allowing a "chaotic" order of firing events, as dictated by the dynamics of creating new nodes. Moreover, all firings of events are very lightweight operations, with localized and chained/pipelined effects, as opposed to the monolithic heavyweight image computation of traditional breadth-first strategies. Storing and managing only saturated nodes has additional benefits. Many, if not most, of these nodes will still be present on the final MDD, while the unsaturated nodes are *guaranteed* not to be part of it. These properties lead to enormous time and memory savings, as illustrated in the results section: saturation is up to five orders of magnitude faster and up to three orders of magnitude less memory when compared to other state-of-the-art tools, such as NuSMV. The experimental studies for saturation also show that, at times, saturation is *optimal*, in the sense that the peak and final numbers of nodes differ by a small *constant*.

4 Building local state spaces on-the-fly

Symbolic analysis of unbounded discrete-event systems has been considered before. In most cases, the goal is the study of systems with infinite but *regular* state spaces. For example, the Queue BDDs of [26] allow one to model systems with a finite number of boolean variables plus one or more unbounded queues, as long as the contents of the queue can be represented by a deterministic finite automaton. The MONA system [30] implements *monadic second-order logic* and can be used to verify *parametric*

```
$Generate(\mathbf{s} : state,\ \mathcal{N} : \widehat{\mathcal{S}} \to 2^{\widehat{\mathcal{S}}}) : index$
Build an MDD rooted at $\langle K|r\rangle$ encoding $\mathcal{N}^*(\mathbf{s})$, return $r$.
1  declare $r, p : index,\quad k : level$;
2  $p \leftarrow 1$;
3  for $k = 1$ to $K$ do
4    $r \leftarrow NewNode(k)$;
5    $\langle k|r\rangle[0] \leftarrow p$;                • the index $\psi_k(\mathbf{s}_k)$ of is 0
6    $Saturate(k, r)$;
7    $CheckIn(k, r)$;
8    $p \leftarrow r$;
9  return $r$;

$Saturate(k : level,\ p : index)$
Update $\langle k|p\rangle$ in-place, to encode $\mathcal{N}^*_{\leq k}(\mathcal{B}(\langle k|p\rangle))$.
1  declare $e : event,\quad chng : bool$;
2  $chng \leftarrow true$;
3  while $chng$ do
4    $chng \leftarrow false$;
5    foreach $e \in \mathcal{E}_k$ do
6      $chng \leftarrow chng \vee SatFire(\alpha, k, p)$;

$SatFire(\alpha : event,\ k : level,\ p : index) : bool$
Update $\langle k|p\rangle$ in-place, to encode $\mathcal{N}^*_{\leq k-1}(\mathcal{N}^*_\alpha(\mathcal{B}(\langle k|p\rangle)))$.
1  declare $f, u : index,\quad i, j : local,\quad \mathcal{L}$ : set of $local$ $chng : bool$;
2  $chng \leftarrow false$;
3  $\mathcal{L} \leftarrow \{i_k \in \mathcal{S}_k : \langle k|p\rangle[i_k] \neq 0,\ \mathbf{N}_{k,\alpha}[i_k, \cdot] \neq \mathbf{0}\}$;
4  while $\mathcal{L} \neq \emptyset$ do
5    pick and remove $i$ from $\mathcal{L}$;
6    $f \leftarrow SatRecFire(\alpha, k-1, \langle k|p\rangle[i])$;
7    if $f \neq 0$ then
8      foreach $j$ s.t. $\mathbf{N}_{k,\alpha}[i, j] = 1$ do
9        $u \leftarrow Union(k-1, f, \langle k|p\rangle[j])$;
10       if $u \neq \langle k|p\rangle[j]$ then
11         $\langle k|p\rangle[j] \leftarrow u$;
12         $chng \leftarrow true$;
13         if $\mathbf{N}_{k,\alpha}[j, \cdot] \neq \mathbf{0}$ then
14           $\mathcal{L} \leftarrow \mathcal{L} \cup \{j\}$;
15 return $chng$;

$SatRecFire(\alpha : event,\ l : level,\ q : index) : index$
Build an MDD rooted at $\langle l|s\rangle$ encoding $\mathcal{N}^*_{\leq l}(\mathcal{N}_\alpha(\mathcal{B}(\langle l|q\rangle)))$,
return $s$.
1  declare $f, u, s : index,\ i, j : local,\ \mathcal{L}$ : set of $local$, $chng : bool$;
2  if $l < Bot(\alpha)$ then return $q$;
3  if $Cached(FIRE, l, \alpha, q, s)$ then return $s$;
4  $s \leftarrow NewNode(l)$; $chng \leftarrow false$;
5  $\mathcal{L} \leftarrow \{i_l \in \mathcal{S}_l : \langle l|q\rangle[i_l] \neq 0,\ \mathbf{N}_{l,\alpha}[i_l, \cdot] \neq \mathbf{0}\}$;
6  while $\mathcal{L} \neq \emptyset$ do
7    pick and remove $i$ from $\mathcal{L}$;
8    $f \leftarrow SatRecFire(\alpha, l-1, \langle l|q\rangle[i])$;
9    if $f \neq 0$ then
10     foreach $j$ s.t. $\mathbf{N}_{l,\alpha}[i, j] = 1$ do
11       $u \leftarrow Union(l-1, f, \langle l|s\rangle[j])$;
12       if $u \neq \langle l|s\rangle[j]$ then
13         $\langle l|s\rangle[j] \leftarrow u$;
14         $chng \leftarrow true$;
15 if $chng$ then $Saturate(l, s)$;
16 $CheckIn(l, s)$;
17 $PutInCache(FIRE, l, \alpha, q, s)$;
18 return $s$;
```

Fig. 12. *Generate*, *Saturate*, *SatFire*, and *SatRecFire*

systems without relying on a proof by induction. These types of approach can be generally classified under the umbrella of *regular model checking* [5].

Here, we target a different problem: the analysis of bounded models with unknown bounds of the state variables. Traditional symbolic state-space generation assumes that each local state space is known a priori. One could argue that this is reasonable for BDD-based methods, since each boolean variable takes simply values in $\{0,1\}$; however, this simply require a similar assumption, i.e, that we know how many boolean variables are needed to represent some system variable, such as an integer variable in a software module being modeled, or the number of tokens in a Petri net place. In other words, we need to know each $\mathcal{S}_k$, and in particular its size n_k, so that we can set up either the correct number $\lceil \log n_k \rceil$ of boolean variables for it, if we use a BDD, or the correct size n_k of the nodes at level k, if we use an MDD.

The algorithm we describe in this section can be used for symbolic state-space generation when all we know (or need to assume) initially about $\mathcal{S}_k$ is that it has a finite but unknown size. The main idea is to interleave symbolic generation of the global state space with explicit, "on-the-fly" generation of the *smallest* local state spaces (plus, possibly, a small *rim* of additional local states that are discarded at the end). Starting only with the knowledge of the initial state, the algorithm discovers new local states at each level. As n_k increases, the size of the MDD nodes at level k increases as well (if we used BDDs, the number of levels would have to increase instead).

Typical applications of this algorithm are found in modeling distributed software, one of the most challenging cases for symbolic methods. For this type of systems, even though the state spaces are finite, the highly "irregular" nature of the local spaces makes it difficult to apply regular model checking methods.

For a given high-level formalism, we can attempt to *pregenerate* the k^{th} local state space $\mathcal{S}_k$ with an explicit traversal of the local state-to-state transition graph of the k^{th} submodel, obtained by considering all the variables associated with that submodel and all the events affecting those variables. Unfortunately, this may create *spurious* local states. For example, if the two places of the Petri net in Fig. 13(a) are partitioned into two subsets, the corresponding subnets, (b) and (c), have unbounded local state spaces when considered in isolation. In subnet (b), transition u can keep adding tokens to place a since, without the input arc from b, u is always locally enabled. Hence, in isolation, a may contain arbitrarily many tokens. The same can be said for subnet (c). However, $\mathcal{S} = \{(1,0),(0,1)\}$, so we would ideally like to define $\mathcal{S}_2 = \mathcal{S}_1 = \{0,1\}$. This can be enforced by adding either inhibitor arcs, (d), or complementary places, (e). Consider now the Petri net of Fig. 13(f), partitioned into two subnets, one containing c, d, and e, the other containing f and g. The inhibitor

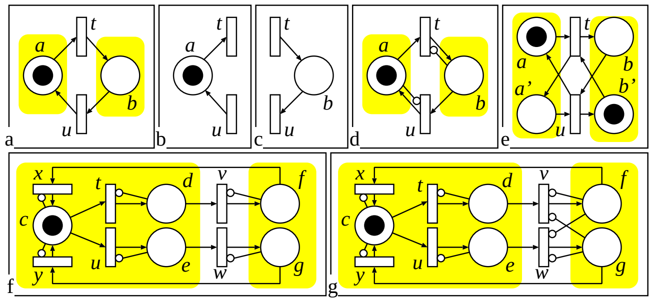


Fig. 13. Local state spaces built in isolation can be strict supersets of the actual ones

arcs shown avoid unbounded local state spaces in isolation, but they don't ensure that the local state spaces are as small as possible. For example, the local state space built in isolation for the subnet containing f and g is $\{(0,0),(1,0),(0,1),(1,1)\}$, while only the first three states are actually reachable in the overall net, since f and g can never contain a token at the same time. This is corrected in (g) by adding two more inhibitor arcs, from g to v and from f to w. An analogous problem exists for the other subnet as well, and correcting it with inhibitor arcs is even more cumbersome.

Thus, there are two problems with pregeneration: a local state space in isolation might be unbounded (causing pregeneration to fail) or it might contain spurious states (causing inefficiencies in the symbolic state-space generation, since n_k is larger that needed). Asking the modeler to cope with these problems by adding constraints to the original model (e.g., the inhibitor arcs in Fig. 13) is at best *burdensome*, since it requires a priori knowledge of $\mathcal{S}$, the output of state-space generation, and at worst *dangerous*, since adding the wrong constraint might hide undesirable behaviors present in the original model. This has been addressed before for explicit compositional approaches [27,35], but not, to the best of our knowledge, for symbolic approaches.

4.1 Local state spaces with unknown bounds

We now describe an *on-the-fly* algorithm to intertwine explicit generation of the local state spaces with symbolic generation of the global state space and, as a result, build the *smallest* local state spaces $\mathcal{S}_k$ needed to encode the correct global state space $\mathcal{S} \subseteq \widehat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$. Ideally, the additional time spent exploring local state spaces on-the-fly should be comparable to that spent in the pregeneration phase of our previous algorithm. This is indeed the case for our new algorithm, which incrementally discovers a set $\widehat{\mathcal{S}}_k$ of *locally-reachable* local states, of which a subset $\mathcal{S}_k$ is known to be also globally reachable. Our algorithm *confirms* that a local state $i_k \in \widehat{\mathcal{S}}_k$ is globally reachable when it appears as the k^{th} component of some state encoded by the MDD rooted at $\langle K|r\rangle$. Since *unconfirmed* local states in $\widehat{\mathcal{S}}_k \setminus \mathcal{S}_k$ are limited to a "rim" around the confirmed ones, and since unconfirmed states do not affect the size of the MDD nodes, there is only a small memory and time overhead in practice.

In this on-the-fly version of the saturation algorithm, the arcs of the MDD nodes are labeled only with confirmed states, while our Kronecker encoding of the next-state function must describe all possible transitions from confirmed local states to both confirmed and unconfirmed local states. Thus, we only explore *global symbolic firings* originating in confirmed states.

The on-the-fly algorithm follows exactly the same steps as the pregeneration one, except for the need to confirm local states. Rather than providing the entire pseudocode for this modified algorithm, we show procedure *Confirm* in Fig. 14, and list the three places where it should be called from the pregeneration pseudocode of Fig. 12. Between lines 3 and 4 of function *Generate* we add the statement

$Confirm(k, 0)$;

to confirm each initial local state, since we number local states starting at 0, that is, $\psi(\mathbf{s}_k) = 0$ for all submodels k. Between lines 10 and 11 of function *SatFire* we add the statement

if $j \notin \mathcal{S}_k$ then $Confirm(k, j)$;

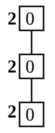
and between lines 12 and 13 of function *SatRecFire* we add the statement

if $j \notin \mathcal{S}_l$ then $Confirm(l, j)$;

to confirm each local state j (if not already confirmed) after it has been determined that a global state with j as the k^{th}, or l^{th}, component can be reached through a symbolic firing.

The firing of an event α in a local state i for node $\langle k|p\rangle$ may lead to a state $j \in \mathcal{S}_k$ or $j \in \widehat{\mathcal{S}}_k \setminus \mathcal{S}_k$. In the former case, j is already confirmed and row j of $\mathbf{N}_{k,\alpha}$ has been built, thus the local states reachable from j are already in $\widehat{\mathcal{S}}_k$. In the latter case, j is unconfirmed: it is locally, but not necessarily globally, reachable, thus it appears as a column index but has no corresponding row in $\mathbf{N}_{k,\alpha}$. Local state j will be confirmed if the global symbolic firing that used the entry $\mathbf{N}_{k,\alpha}[i,j]$ is actually possible, i.e., if α can fire in an MDD path from $Top(\alpha)$ to $Bot(\alpha)$ passing through node $\langle k|p\rangle$. Only when j is confirmed, the corresponding rows $\mathbf{N}_{k,\alpha}[j,\cdot]$ (for all events α that depend on k) are built, using one forward step of *explicit local reachability analysis*. This step must consult the description of the model itself, and thus works on actual submodel variables, not state indices. This is the only operation that may discover new unconfirmed local states. Thus, the on-the-fly algorithm uses "rectangular" Kronecker matrices over $\{0,1\}^{\mathcal{S}_k \times \widehat{\mathcal{S}}_k}$. In other words, only confirmed local states "know" their successors, confirmed or not, while unconfirmed states appear only in the columns of the Kronecker matrices.

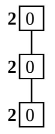
(a) Initial Configuration:



	a	b	c	d
3			**I**	**I**
2				**I**
1	**I**	**I**		

$\mathcal{S}_3 = \{p^1\} \equiv \{0\}$
$\mathcal{S}_2 = \{q^0r^0\} \equiv \{0\}$
$\mathcal{S}_1 = \{s^1t^0\} \equiv \{0\}$

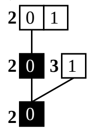
(b) Confirm initial $p^1 \equiv 0, q^0r^0 \equiv 0, s^1t^0 \equiv 0$:



	a	b	c	d
3	0:1	0:2	**I**	**I**
2	0:1			**I**
1	**I**	**I**	0:1	

$\mathcal{S}_3 = \{\underline{p^1}, p^0, p^2\} \equiv \{\underline{0}, 1, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, q^1r^1\} \equiv \{\underline{0}, 1\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, s^0t^1\} \equiv \{\underline{0}, 1\}$

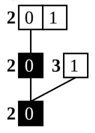
(c) Saturate $\langle 3|2\rangle$(Fire a):



	a	b	c	d
3	0:1	0:2	**I**	**I**
2	0:1			**I**
1	**I**	**I**	0:1	

$\mathcal{S}_3 = \{\underline{p^1}, p^0, p^2\} \equiv \{\underline{0}, 1, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, q^1r^1\} \equiv \{\underline{0}, 1\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, s^0t^1\} \equiv \{\underline{0}, 1\}$

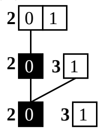
(d) Confirm $q^1r^1 \equiv 1$:



	a	b	c	d
3	0:1	0:2	**I**	**I**
2	0:1	1:2	1:3	**I**
1	**I**	**I**	0:1	

$\mathcal{S}_3 = \{\underline{p^1}, p^0, p^2\} \equiv \{\underline{0}, 1, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, \underline{q^1r^1}, q^0r^1, q^1r^0\} \equiv \{\underline{0}, \underline{1}, 2, 3\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, s^0t^1\} \equiv \{\underline{0}, 1\}$

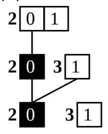
(e) Saturate $\langle 2|3\rangle$(Fire c):



	a	b	c	d
3	0:1	0:2	**I**	**I**
2	0:1	1:2	1:3	**I**
1	**I**	**I**	0:1	

$\mathcal{S}_3 = \{\underline{p^1}, p^0, p^2\} \equiv \{\underline{0}, 1, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, \underline{q^1r^1}, q^0r^1, q^1r^0\} \equiv \{\underline{0}, \underline{1}, 2, 3\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, \underline{s^0t^1}\} \equiv \{\underline{0}, 1\}$

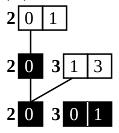
(f) Confirm $s^0t^1 \equiv 1$:



	a	b	c	d
3	0:1	0:2	**I**	**I**
2	0:1	1:2	1:3	**I**
1	**I**	**I**	0:1	1:0

$\mathcal{S}_3 = \{\underline{p^1}, p^0, p^2\} \equiv \{\underline{0}, 1, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, \underline{q^1r^1}, q^0r^1, q^1r^0\} \equiv \{\underline{0}, \underline{1}, 2, 3\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, \underline{s^0t^1}\} \equiv \{\underline{0}, \underline{1}\}$

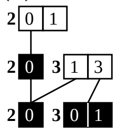
(g) Saturate $\langle 1|3\rangle$(Fire d):



	a	b	c	d
3	0:1	0:2	**I**	**I**
2	0:1	1:2	1:3	**I**
1	**I**	**I**	0:1	1:0

$\mathcal{S}_3 = \{\underline{p^1}, p^0, p^2\} \equiv \{\underline{0}, 1, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, \underline{q^1r^1}, q^0r^1, q^1r^0\} \equiv \{\underline{0}, \underline{1}, 2, 3\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, \underline{s^0t^1}\} \equiv \{\underline{0}, \underline{1}\}$

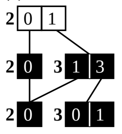
(h) Confirm $q^1r^0 \equiv 3$:



	a	b	c	d
3	0:1	0:2	**I**	**I**
2	0:1,3:4	1:2,3:0	1:3	**I**
1	**I**	**I**	0:1	1:0

$\mathcal{S}_3 = \{\underline{p^1}, p^0, p^2\} \equiv \{\underline{0}, 1, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, \underline{q^1r^1}, q^0r^1, \underline{q^1r^0}, q^2r^1\} \equiv \{\underline{0}, \underline{1}, 2, \underline{3}, 4\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, \underline{s^0t^1}\} \equiv \{\underline{0}, \underline{1}\}$

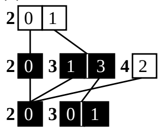
(i) Confirm $p^0 \equiv 1$:



	a	b	c	d
3	0:1	0:2,1:0	**I**	**I**
2	0:1,3:4	1:2,3:0	1:3	**I**
1	**I**	**I**	0:1	1:0

$\mathcal{S}_3 = \{\underline{p^1}, \underline{p^0}, p^2\} \equiv \{\underline{0}, \underline{1}, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, \underline{q^1r^1}, q^0r^1, \underline{q^1r^0}, q^2r^1\} \equiv \{\underline{0}, \underline{1}, 2, \underline{3}, 4\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, \underline{s^0t^1}\} \equiv \{\underline{0}, \underline{1}\}$

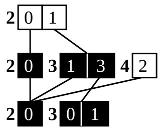
(j) Continue Saturating $\langle 3|2\rangle$(Fire b):



	a	b	c	d
3	0:1	0:2,1:0	**I**	**I**
2	0:1,3:4	1:2,3:0	1:3	**I**
1	**I**	**I**	0:1	1:0

$\mathcal{S}_3 = \{\underline{p^1}, \underline{p^0}, p^2\} \equiv \{\underline{0}, \underline{1}, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, \underline{q^1r^1}, q^0r^1, \underline{q^1r^0}, q^2r^1\} \equiv \{\underline{0}, \underline{1}, 2, \underline{3}, 4\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, \underline{s^0t^1}\} \equiv \{\underline{0}, \underline{1}\}$

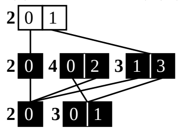
(k) Confirm $q^0r^1 \equiv 2$:



	a	b	c	d
3	0:1	0:2,1:0	**I**	**I**
2	0:1,3:4	1:2,3:0	1:3,2:0	**I**
1	**I**	**I**	0:1	1:0

$\mathcal{S}_3 = \{\underline{p^1}, \underline{p^0}, p^2\} \equiv \{\underline{0}, \underline{1}, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, \underline{q^1r^1}, \underline{q^0r^1}, \underline{q^1r^0}, q^2r^1\} \equiv \{\underline{0}, \underline{1}, \underline{2}, \underline{3}, 4\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, \underline{s^0t^1}\} \equiv \{\underline{0}, \underline{1}\}$

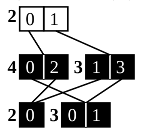
(l) Saturate $\langle 2|4\rangle$(Fire c):



	a	b	c	d
3	0:1	0:2,1:0	**I**	**I**
2	0:1,3:4	1:2,3:0	1:3,2:0	**I**
1	**I**	**I**	0:1	1:0

$\mathcal{S}_3 = \{\underline{p^1}, \underline{p^0}, p^2\} \equiv \{\underline{0}, \underline{1}, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, \underline{q^1r^1}, \underline{q^0r^1}, \underline{q^1r^0}, q^2r^1\} \equiv \{\underline{0}, \underline{1}, \underline{2}, \underline{3}, 4\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, \underline{s^0t^1}\} \equiv \{\underline{0}, \underline{1}\}$

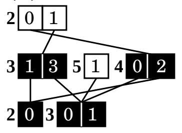
(m) Union $\langle 2|2\rangle$ and $\langle 2|4\rangle$:



	a	b	c	d
3	0:1	0:2,1:0	**I**	**I**
2	0:1,3:4	1:2,3:0	1:3,2:0	**I**
1	**I**	**I**	0:1	1:0

$\mathcal{S}_3 = \{\underline{p^1}, \underline{p^0}, p^2\} \equiv \{\underline{0}, \underline{1}, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, \underline{q^1r^1}, \underline{q^0r^1}, \underline{q^1r^0}, q^2r^1\} \equiv \{\underline{0}, \underline{1}, \underline{2}, \underline{3}, 4\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, \underline{s^0t^1}\} \equiv \{\underline{0}, \underline{1}\}$

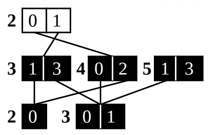
(n) Continue Saturating $\langle 3|2\rangle$(Fire a):



	a	b	c	d
3	0:1	0:2,1:0	**I**	**I**
2	0:1,3:4	1:2,3:0	1:3,2:0	**I**
1	**I**	**I**	0:1	1:0

$\mathcal{S}_3 = \{\underline{p^1}, \underline{p^0}, p^2\} \equiv \{\underline{0}, \underline{1}, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, \underline{q^1r^1}, \underline{q^0r^1}, \underline{q^1r^0}, q^2r^1\} \equiv \{\underline{0}, \underline{1}, \underline{2}, \underline{3}, 4\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, \underline{s^0t^1}\} \equiv \{\underline{0}, \underline{1}\}$

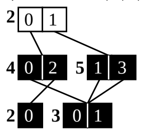
(o) Saturate $\langle 2|5\rangle$(Fire c):



	a	b	c	d
3	0:1	0:2,1:0	**I**	**I**
2	0:1,3:4	1:2,3:0	1:3,2:0	**I**
1	**I**	**I**	0:1	1:0

$\mathcal{S}_3 = \{\underline{p^1}, \underline{p^0}, p^2\} \equiv \{\underline{0}, \underline{1}, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, \underline{q^1r^1}, \underline{q^0r^1}, \underline{q^1r^0}, q^2r^1\} \equiv \{\underline{0}, \underline{1}, \underline{2}, \underline{3}, 4\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, \underline{s^0t^1}\} \equiv \{\underline{0}, \underline{1}\}$

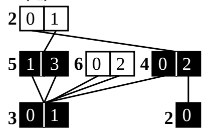
(p) Union $\langle 2|3\rangle$ and $\langle 2|5\rangle$:



	a	b	c	d
3	0:1	0:2,1:0	**I**	**I**
2	0:1,3:4	1:2,3:0	1:3,2:0	**I**
1	**I**	**I**	0:1	1:0

$\mathcal{S}_3 = \{\underline{p^1}, \underline{p^0}, p^2\} \equiv \{\underline{0}, \underline{1}, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, \underline{q^1r^1}, \underline{q^0r^1}, \underline{q^1r^0}, q^2r^1\} \equiv \{\underline{0}, \underline{1}, \underline{2}, \underline{3}, 4\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, \underline{s^0t^1}\} \equiv \{\underline{0}, \underline{1}\}$

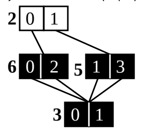
(q) Continue Saturating $\langle 3|2\rangle$(Fire b):



	a	b	c	d
3	0:1	0:2,1:0	**I**	**I**
2	0:1,3:4	1:2,3:0	1:3,2:0	**I**
1	**I**	**I**	0:1	1:0

$\mathcal{S}_3 = \{\underline{p^1}, \underline{p^0}, p^2\} \equiv \{\underline{0}, \underline{1}, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, \underline{q^1r^1}, \underline{q^0r^1}, \underline{q^1r^0}, q^2r^1\} \equiv \{\underline{0}, \underline{1}, \underline{2}, \underline{3}, 4\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, \underline{s^0t^1}\} \equiv \{\underline{0}, \underline{1}\}$

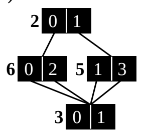
(r) Union $\langle 2|4\rangle$ and $\langle 2|6\rangle$:



	a	b	c	d
3	0:1	0:2,1:0	**I**	**I**
2	0:1,3:4	1:2,3:0	1:3,2:0	**I**
1	**I**	**I**	0:1	1:0

$\mathcal{S}_3 = \{\underline{p^1}, \underline{p^0}, p^2\} \equiv \{\underline{0}, \underline{1}, 2\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, \underline{q^1r^1}, \underline{q^0r^1}, \underline{q^1r^0}, q^2r^1\} \equiv \{\underline{0}, \underline{1}, \underline{2}, \underline{3}, 4\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, \underline{s^0t^1}\} \equiv \{\underline{0}, \underline{1}\}$

(s) Discard unreachable local states:



	a	b	c	d
3	0:1	1:0	**I**	**I**
2	0:1	1:2,3:0	1:3,2:0	**I**
1	**I**	**I**	0:1	1:0

$\mathcal{S}_3 = \{\underline{p^1}, \underline{p^0}\} \equiv \{\underline{0}, \underline{1}\}$
$\mathcal{S}_2 = \{\underline{q^0r^0}, \underline{q^1r^1}, \underline{q^0r^1}, \underline{q^1r^0}\} \equiv \{\underline{0}, \underline{1}, \underline{2}, \underline{3}\}$
$\mathcal{S}_1 = \{\underline{s^1t^0}, \underline{s^0t^1}\} \equiv \{\underline{0}, \underline{1}\}$

Fig. 15. Saturation by example

```
Confirm(k : level, i : local)
Add local state index i to $\mathcal{S}_k$ and build the corresponding
rows $\mathbf{N}_{k,\alpha}[i,\cdot]$ for all matrices $\mathbf{N}_{k,\alpha} \neq \mathbf{I}$.
 1 declare $\alpha$ : event,  j : local,  n : int,  $\mathbf{i}_k, \mathbf{j}_k$ : state;
 2 $\mathbf{i}_k \leftarrow \psi_k^{-1}(i)$;
 3 foreach $\alpha$ s.t. $\mathbf{N}_{k,\alpha} \neq \mathbf{I}$ do
 4   foreach $\mathbf{j}_k \in \mathcal{N}_{k,\alpha}(\mathbf{i}_k)$ do          • access high-level model
 5     $j \leftarrow \psi_k(\mathbf{j}_k)$;
 6     if $j$ = null then                  • $\mathbf{j}_k \notin \widehat{\mathcal{S}}_k$, new local state
 7       $\psi_k(\mathbf{j}_k) \leftarrow |\widehat{\mathcal{S}}_k|$;          • assign next local index to $\mathbf{j}_k$
 8       $j \leftarrow |\widehat{\mathcal{S}}_k|$;
 9       $\widehat{\mathcal{S}}_k \leftarrow \widehat{\mathcal{S}}_k \cup \{\mathbf{j}_k\}$;
10     $\mathbf{N}_{k,\alpha}[i,j] \leftarrow 1$;
11 $\mathcal{S}_k \leftarrow \mathcal{S}_k \cup \{\mathbf{i}_k\}$;
```

Fig. 14. The *Confirm* procedure for saturation on-the-fly

4.2 Example

Fig. 15 illustrates the application of the on-the-fly algorithm on our running example. The net is partitioned into three subnets, defined by sets of places $\{p\}$, $\{q,r\}$, and $\{s,t\}$, respectively. Accordingly, the events are partitioned into $\mathcal{E}_3 = \{a,b\}$, $\mathcal{E}_2 = \{c\}$, and $\mathcal{E}_1 = \{d\}$. In each snapshot we show the current status of the MDD on the left, and of the Kronecker matrices and local spaces on the right. Local states are described by the number of tokens, listed as superscripts, for each place in the subnet. Their equivalent indices, that is, the mapping ψ, is also given, and confirmed local states are underlined. Saturated MDD nodes are shown in reversed: dark background, light foreground. For the matrices we adopt a concise description listing only the nonzero entries, in the format *row*:*col*. This is very close to the actual sparse storage representation we use for each Kronecker matrix.

(a) The algorithm starts with the insertion of the initial state $(0,0,0)$, encoded with one MDD node per level.
(b) The initial local states, each indexed 0, are confirmed. Their corresponding rows in the Kronecker matrices are built by consulting the model. This leads to the discovery of new local states. In subspace $\widehat{\mathcal{S}}_3$, local state p^0, indexed 1 is obtained by firing event a, and local state p^2, indexed 2, by firing event b. Similarly, in subspaces $\widehat{\mathcal{S}}^2$ and $\widehat{\mathcal{S}}_1$, local states q^1r^1 and s^0t^1 are discovered from the initial local state, and are each indexed 1.
(c) The saturation process starts with the bottom node $\langle 1|2\rangle$. Since it enables no event, it is marked saturated. The same holds for node $\langle 2|2\rangle$. Moving up, we begin saturating the root node, $\langle 3|2\rangle$. Event a is enabled at level 3 by local state 0. *SatFire*$(a,3,2)$ calls *SatRecFire*$(a,2,\langle 3|2\rangle[0])$, which creates a new node, $\langle 2|3\rangle$. Since local state 0 at level 2 enables a, the recursive call continues downstream with *SatRecFire* $(a,1,\langle 2|2\rangle[0])$. This returns index $2 = \langle 2|2\rangle[0]$, since $1 < Bot(a)$.
(d) Upon return, an arc is set from $\langle 2|3\rangle[1]$ to $\langle 1|2\rangle$, meaning the unconfirmed local state $q^1r^1 \equiv 1$ has been globally reached and the *Confirm* procedure is called on substate q^1r^1. As a result, two new local states, q^0r^1 and q^1r^0, are discovered by locally firing events b and c, respectively, and indexed 2 and 3 in $\widehat{\mathcal{S}}_2$. The pairs $1:2$ and $1:3$ are added to the corresponding matrices, $\mathbf{N}_{2,b}$ and $\mathbf{N}_{2,c}$.
(e) At this point, *SatRecFire*$(a,2,2)$ has the result of firing a below level 2, encoded by $\langle 2|3\rangle$, which we need to saturate. The only event in $\mathcal{E}_2$, c, is enabled by local state 1, and moves subsystem 2 to local state 3, according to $\mathbf{N}_{2,c}$. *SatFire*$(c,2,3)$ makes the recursive call *SatRecFire*$(c,1,\langle 2|3\rangle[1])$. This creates the node $\langle 1|3\rangle$, and sets its arc $\langle 1|3\rangle[1]$ to $\langle 0|1\rangle$, the base case result of *SatRecFire*$(c,0,1)$.
(f) Local state $s^0t^1 \equiv 1$ has been globally reached and has to be confirmed in $\widehat{\mathcal{S}}_1$. The only possible move discovered from this state is via event d, and leads submodel 1 to the already known local state $s^1t^0 \equiv 0$.
(g) Since $\langle 1|3\rangle$ is a new node, it is immediately saturated, by firing event d, the only element of $\mathcal{E}_1$. This adds the arc $\langle 1|3\rangle[0]$ to point to the terminal node $\langle 0|1\rangle$. Local state 0 at level 1 is an old state, hence it does not need to be confirmed again. Node $\langle 1|3\rangle$ is declared saturated and returned as the result of *SatRecFire*$(c,1,2)$. An arc is set from $\langle 2|3\rangle[3]$ to $\langle 1|3\rangle$ to represent the local move from local state 1 to local state 3 in $\widehat{\mathcal{S}}_2$, as demanded by $\mathbf{N}_{2,c}$.
(h) Local state $q^1r^0 \equiv 3$ now has to be confirmed. It leads to the already known state $q^0r^0 \equiv 0$ via b, and a new local state q^2r^1, which is indexed 4, via a. Node $\langle 2|3\rangle$ is now saturated and it is returned as the result of *SatRecFire*$(a,2,2)$, which started in snapshot (c).
(i) Arc $\langle 3|2\rangle[1]$ is set to point to $\langle 2|3\rangle$, signifying that local state $p^0 \equiv 1$ has been reached. The confirmation of it in $\widehat{\mathcal{S}}_3$ adds one entry in the Kronecker matrices: to $p^1 \equiv 0$ in $\mathbf{N}_{3,b}$.
(j) The saturation of node $\langle 3|2\rangle$ continues with the firing of the other event in $\mathcal{E}_3$, b. *SatFire*$(b,3,2)$ is enabled by both local states at level 3 represented in the node, 0 and 1. The first subsequent call, *SatRecFire*$(b,2,\langle 3|2\rangle[0])$ finds no local states at level 2 enabled in node $\langle 3|2\rangle[0] = \langle 2|2\rangle$. The recursion stops and returns index 0, signaling the failure to fire event b from this particular combination of local states. The second call, *SatRecFire*$(b,2,\langle 3|2\rangle[1])$ is successful in firing b, based on the enabling pattern $(1,1,*)$, and creates node $\langle 2|4\rangle$.
(k) Since local state $q^0r^1 \equiv 2$ is reached in this firing, q^0r^1 needs to be confirmed in $\widehat{\mathcal{S}}_2$. The only move discovered is to $q^0r^0 \equiv 0$, via event c. The corresponding element, $1\!:\!0$, is inserted in $\mathbf{N}_{2,c}$.
(l) After setting the arc $\langle 2|4\rangle[2]$ to $\langle 1|3\rangle$, node $\langle 2|4\rangle$ has to be saturated. Event c is enabled, and its firing causes node $\langle 2|4\rangle$ to be updated by setting its arc $\langle 2|4\rangle[0]$ to $\langle 1|2\rangle$. No other firings produce changes to

the node, which is returned as the result of the call *SatRecFire* $(b, 2, \langle 3|2\rangle[1])$.

(m) Upon return, node $\langle 3|2\rangle$ is updated in-place to incorporate the effect of the recursive calls. This requires a union operation between its existing child in position 0, $\langle 2|2\rangle$, and $\langle 2|4\rangle$. The result of the union is $\langle 2|4\rangle$. The arc $\langle 3|2\rangle[0]$ is updated accordingly, causing node $\langle 2|2\rangle$ to become disconnected and, therefore, removed from the diagram.

(n) Even though both events a and b have already been fired in $\langle 3|2\rangle$, the saturation of the root node is not complete, since both firings produced new states. The loop continues and event a is fired again, which induces the creation of node $\langle 2|5\rangle$ at level 2.

(o) Node $\langle 2|5\rangle$ is saturated by firing event c, and adding in-place the arc $\langle 2|5\rangle[3]$ to $\langle 1|3\rangle$.

(p) The union of $\langle 2|3\rangle$ and $\langle 2|5\rangle$, required by the in-place update of the root, is node $\langle 2|5\rangle$, which is known to be saturated, as it represents the union of two saturated nodes. Node $\langle 2|5\rangle$ is the new successor of the root along arc 1. The old successor, node $\langle 2|3\rangle$, becomes disconnected and is deleted.

(q) Finally, we fire event b one more time and set $\langle 3|2\rangle[0]$ to the result of the union between $\langle 2|4\rangle$ and newly created $\langle 2|6\rangle$.

(r) The union of the two nodes is encoded by $\langle 2|6\rangle$. Further attempts to fire a or b in the root do not discover any other new states, hence the root is finally saturated.

(s) The algorithm concludes by discarding all local states still unconfirmed. These are $p^2 \equiv 2$, at level 3, and $q^2r^1 \equiv 4$, at level 2.

5 Implementation in SMART

We implemented the data structures and algorithms described in this paper are implemented in our tool SMART [12], as part of its symbolic model checking functionalities targeted to concurrent systems.

5.1 Data structures for the next-state function

We store the next-state function $\mathcal{N}$ of the model using a sparse matrix $\mathbf{N}$, whose entries are themselves sparse matrices. The rows of $\mathbf{N}$ correspond to the levels of the model. The columns of $\mathbf{N}$ correspond to the events of the model. Each entry of $\mathbf{N}$ contains a sparse matrix $\mathbf{N}_{k,\alpha}$ describing the effect on level k of firing event α. Storing $\mathbf{N}$ as a full two-dimensional array would waste a significant amount of memory, since most $\mathbf{N}_{k,\alpha}$ are identities, so we use a sparse encoding where a missing entry means "identity", not "zero".

We provide access using lists linked by columns (so that we can efficiently access all levels affected by an event α), and by rows (so that we can efficiently access all

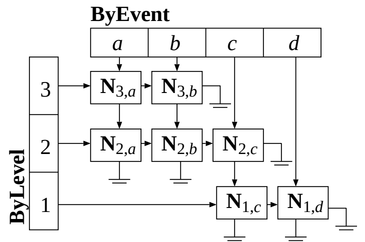


Fig. 16. Storage for the matrix nodes

events affecting a level k). These two types of access are required by the most common and expensive operations of state space generation, *firing* an event and *confirming* that a local state is globally reachable, respectively. The resulting data structure is shown in Fig. 16.

Each node in the figure consists of a pointer to the data structure storing $\mathbf{N}_{k,\alpha}$, plus pointers to the next elements along its row and along its column. Each $\mathbf{N}_{k,\alpha}$, in turn, is stored as a sparse row-wise matrix [39], since the only required access to $\mathbf{N}_{k,\alpha}$ is the retrieval of all local state indices j such that $\mathbf{N}_{k,\alpha}[i,j] = 1$, for a given i. The only implementation difficulty is that the matrix size must be dynamic, since as we discover local states on-the-fly, we need to add new rows to it (adding new columns is not an issue, since, in a sparse row-wise representation, column indices appear only as values in the entries for each row).

5.2 Unconfirmed vs. confirmed state indices

With the exception of the first local state for each level, which is assigned index 0 and is confirmed by definition, any other local state we find is always unconfirmed at first. Yet, this unconfirmed state $\mathbf{j}_k$ must be referenced as a column index $j = \psi_k(\mathbf{j}_k)$ on a confirmed state row i, that is, we need to be able to set $\mathbf{N}_{k,\alpha}[i,j] = 1$. It is natural and efficient to assign increasing sequential state indices to the states in $\widehat{\mathcal{S}}_k$, but, unfortunately, not all states in $\widehat{\mathcal{S}}_k$ will be necessarily confirmed. Worse yet, even those that will be confirmed are not guaranteed to be confirmed in the order they are discovered.

We could ignore this problem and simply index local states using $\psi_k : \widehat{\mathcal{S}}_k \to \{0, \ldots, \widehat{n}_k - 1\}$ everywhere, where $\widehat{n}_k = |\widehat{\mathcal{S}}_k|$, obviously for the currently-known $\widehat{\mathcal{S}}_k$. This would be indeed correct, but inefficient in two ways. First, the sparse row-wise storage of each $\mathbf{N}_{k,\alpha}$ would require $\widehat{n}_k$ pointers to the rows, while we know that only $n_k = |\mathcal{S}_k|$ rows corresponding to the confirmed local states need to be built and accessed, thus the remaining $\widehat{n}_k - n_k$ row pointers would simply be null. Second, each MDD node at level k would require $\widehat{n}_k$ arcs, while, again, we know that only the subset corresponding to states in $\mathcal{S}_k$ may be pointing to nodes other than $\langle k-1|0\rangle$. This second problem can potentially have a large impact, since it may substantially increase the memory requirements, depending on how the MDD nodes are stored.

To avoid these inefficiencies, we then use a second mapping $\Phi_k : \{0, \ldots, \widehat{n}_k - 1\} \to \{0, \ldots, n_k - 1\} \cup \{\mathsf{null}\}$ from "unconfirmed indices" to "confirmed indices". A local state $\mathbf{j}_k$, then, has an unconfirmed index $j = \psi_k(\mathbf{j})_k$, which is used exclusively for column indices in $\mathbf{N}_{k,\alpha}$. Once it is confirmed, though, it also has a confirmed index $j' = \Phi_k(j)$, which is used as a row index in $\mathbf{N}_{k,\alpha}$ and as an arc label in the MDD nodes at level k. Thus, to test whether a local state $\mathbf{j}_k \in \widehat{\mathcal{S}}$ is confirmed given its unconfirmed index j, we only need to test whether $\Phi_k(j) \neq \mathsf{null}$.

When state-space generation is complete, we know that any unconfirmed state can never be reached, so we:

- Discard the states in $\widehat{\mathcal{S}}_k \setminus \mathcal{S}_k$, keep only those in $\mathcal{S}_k$.
- Redefine ψ_k, so that $\psi_k : \mathcal{S}_k \to \{0, \ldots, n_k - 1\}$.
- Examine the matrices $\mathbf{N}_{k,\alpha}$ and, for each of them, delete any entry $\mathbf{N}_{k,\alpha}[i, j]$ such that $\Phi_k(j) = \mathsf{null}$.
- Switch from unconfirmed to confirmed indices in the column indices, i.e.,, change the entries of each $\mathbf{N}_{k,\alpha}$ so that $\mathbf{N}_{k,\alpha}[i, \Phi_k(j)] = 1$ if $\mathbf{N}_{k,\alpha}[i, j]$ was 1.
- Discard the array used to store the Φ_k mapping.

5.3 Storing MDD nodes

To store the state space, we use a quasi-reduced MDD, which naturally lends itself to a level-oriented node storage. In our implementation, the data for an MDD node $\langle k|p\rangle$ is divided into two extensible arrays *nodes*$_k$ and *arcs*$_k$ associated with level k (Fig. 17). Entry *nodes*$_k[p]$ occupies the same number of bytes for each node (at any level) and stores the following data:

- an integer *offset* into extensible array *arcs*$_k$,
- an integer *size* describing the length of the portion array used by the node,
- a *count* of incoming arcs,
- and the boolean flags *saturated*, *shared*, *sparse*, and *deleted* (the first two are required only when SMART manages multiple MDDs on the same potential state space $\mathcal{S}$, e.g., for model checking).

The data for the arcs of node $\langle k|p\rangle$ occupies instead a variable-size portion of *arcs*$_k$. If *nodes*$_k[p]$.*sparse* $=$ *true*, the arcs of node $\langle k|p\rangle$ occupy $2 \cdot$ *nodes*$_k[p]$.*size* entries of *arcs*$_k$ starting in position *nodes*$_k[p]$.*offset*, as pairs of *local* and *index* values, meaning that $\langle k|p\rangle[\mathit{local}] = \mathit{index}$. If instead *nodes*$_k[p]$.*sparse* $=$ *false*, "truncated full" storage is used, that is, $\mathit{arcs}_k[x] = q$ if $\langle k|p\rangle[x - \mathit{nodes}_k[p].\mathit{offset}] = q$. However, not all n_k arcs need to be stored: if the last m arcs are 0, i.e., $\langle k|p\rangle[n_k - m] = \cdots = \langle k|p\rangle[n_k - 1] = 0$, *nodes*$_k[p]$.*size* is set to $n_k - m$.

5.4 MDD nodes of variable size

We keep track of the index of the last node in *nodes*$_k$ and of the last used position in *arcs*$_k$, and request for a new node al level k are satisfied by using the next available position in these arrays. Thus, the node being saturated is always at the end of the array, and so are its arcs. This is particularly important with saturation on-the-fly, where we do not know beforehand how many entries in *arcs*$_k$ are needed to store the arcs of a node being saturated.

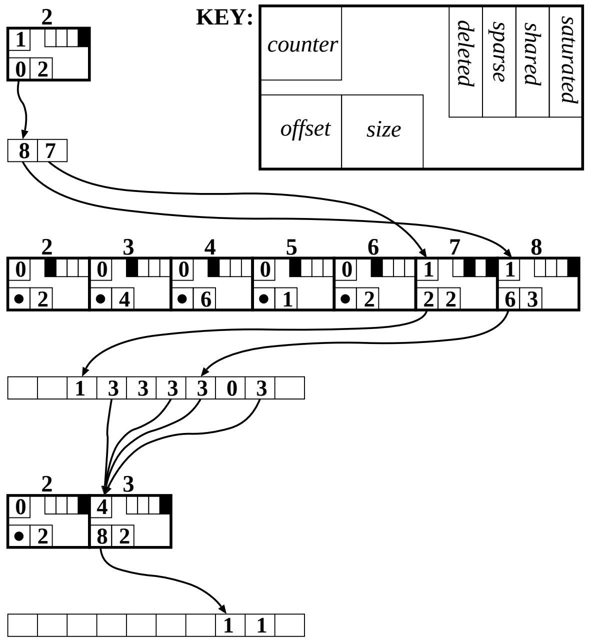


Fig. 17. Node and arc storage for MDDs

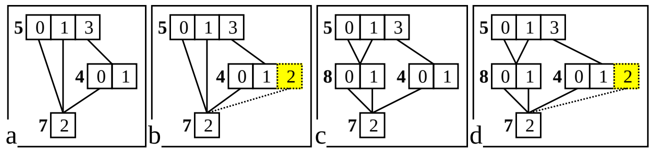


Fig. 18. Adding local state 2 to $\mathcal{S}_2$: reduced (a-b) vs. quasi-reduced (c-d) MDDs

A fundamental property of our encoding is that a saturated node $\langle k|p\rangle$ remains saturated and encodes the same set even after a new state i is added to $\mathcal{S}_k$. This is because, regardless of whether the node is using sparse or truncated full storage, it will be automatically and correctly interpreted as having $\langle k|p\rangle[i] = 0$. However, the possible growth of $\mathcal{S}_k$ implies that it becomes hard to exploit the special meaning for node $\langle k|1\rangle$, which, we recall is $\mathcal{B}(\langle k|1\rangle) = \mathcal{S}_k \times \cdots \times \mathcal{S}_1$. With pregeneration, this optimization speeds up computation whenever node $\langle k|1\rangle$ is involved in a union, since we immediately conclude that $\mathcal{B}(\langle k|1\rangle) \cup \mathcal{B}(\langle k|p\rangle) = \mathcal{B}(\langle k|1\rangle)$. Further, such nodes need not be explicitly stored.

To reserve index 1 for the same purpose with the on-the-fly approach is possible, but problematic, since, whenever a new state is added to $\mathcal{S}_l$, for $l \leq k$, the meaning of $\mathcal{B}(\langle k|1\rangle)$ implicitly changes. This is one of the reasons that led us to use *quasi-reduced* instead of *reduced*

MDDs. The latter eliminates redundant nodes and is potentially more efficient, but its arcs can span multiple levels. As discussed in [13], such arcs are more difficult to manage and can yield a slower state-space generation when exploiting locality. With the on-the-fly algorithm, they create an even worse problem: they become "incorrect" when a local state space grows. For example, both the reduced and the quasi-reduced 3-level MDDs in Fig. 18(a) and (c) encode the state space

$$\mathcal{S} = \{(0,0,2),(0,1,2),(1,0,2),(1,1,2),(3,0,2)\},$$

when $\mathcal{S}_3 = \{0,1,2,3\}$, $\mathcal{S}_2 = \{0,1\}$, and $\mathcal{S}_1 = \{0,1,2\}$. If we want to add global state $(3,2,2)$ to $\mathcal{S}$, we need to add local state 2 to $\mathcal{S}_2$ and set arc $\langle 2|4\rangle[2]$ to $\langle 1|7\rangle$. However, while the resulting quasi-reduced MDD in (d) is correct, the reduced one in (b) is not, since now it also encodes global states $(0,2,2)$ and $(1,2,2)$. To fix the problem, we could reintroduce the formerly-redundant node $\langle 2|8\rangle$ so that the new reduced and quasi-reduced MDDs coincide. While it would be possible to modify the MDD to obtain a correct reduced ordered MDD in this manner whenever a local state space grows, the cost of doing so is unjustifiably high. Of course, we can still reserve index 0 for the empty set, and exploit the relation $\mathcal{B}(\langle k|0\rangle) \cup \mathcal{B}(\langle k|p\rangle) = \mathcal{B}(\langle k|p\rangle)$, since the representation of the empty set does not change as the local state spaces expand.

5.5 Garbage collection

If a node $\langle k|p\rangle$ becomes disconnected, we "mark it for deletion" by setting its flag $nodes_k[p].deleted$ to $true$. However, the node itself remains in memory until the garbage collection manager performs a cleanup. Such a node can then be "resuscitated" by simply resetting this flag. This occurs if a reference to it is retrieved as the result of an operation cache lookup (of course if a cleanup is issued after marking $\langle k|p\rangle$ for deletion and prior to the cache lookup, there will be cache lookup miss, since all entries referring to nodes marked for deletion are eliminated when these nodes are actually deleted).

There is a tradeoff between reducing memory consumption through frequent cleanups versus optimistically hoping to resuscitate nodes and generally avoiding the inherent cost of cleanups. Thus, SMART allows the user to parameterize the garbage collection method by specifying a *lazy* policy, where nodes are physically removed only after having reached a fixed point (the state space, in our particular case), or a *strict* policy, where garbage collection is triggered when a given *threshold* for the number of nodes marked for deletion is exceeded.

One problem with MDD nodes of variable sizes is that the "hole" left in $arcs_k$ when a node is deleted can be smaller than what is needed for a new node, thus it cannot be easily "recycled". However, the use of separate $nodes_k$ and $arcs_k$ arrays affords us great flexibility

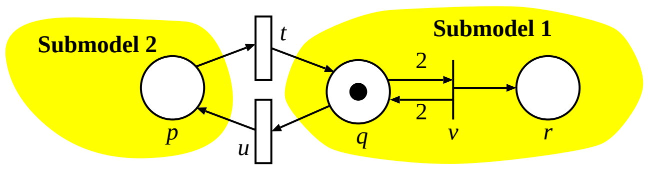


Fig. 19. Potential, not actual, overflow of a local state space

in our garbage collection strategy. In particular, we can "compact to the left" the entire $arcs_k$ array any time the memory used by its holes exceed some threshold (we use 10%) without having to access nodes at other levels. The space freed at the right end of $arcs_k$ can then be reused for new nodes. This compaction only requires to re-adjust the values of each $nodes_k[p].offset$, for each level-k node. There are several ways to do this.

One approach is to store, together with the chunk for $\langle k|p\rangle$ in $arcs_k$, a back pointer to the node itself, i.e., the value p. This allows compaction of $arcs_k$ via a linear scan: valid values may be shifted to the left over invalid values until all holes have been removed. Simultaneously, using the back pointer p, we access and update the value $nodes_k[p].offset$ after shifting the corresponding arcs. With respect to our previous pregeneration implementation, where array $arcs_k$ is not used because the arcs are stored directly in $nodes_k$ as arrays of fixed-size n_k, this requires 12 additional bytes per node, for *offset* and *size* in $nodes_k$ and for the back pointer in $arcs_k$. However, it can also save memory, since we can now employ sparse or truncated full storage.

A better alternative is to build a temporary array *tmp* when compacting the arcs at level k by scanning the nodes sequentially and copying the arcs of each non-deleted node from $arcs_k$ to *tmp*. When done copying, *tmp* becomes our new compacted $arcs_k$ array, and the old $arcs_k$ array is deleted. This additional array *tmp* temporarily increases the memory requirements, but avoids the need for back pointers, saving four bytes per node.

In addition to compacting the $arcs_k$ array, we also need to regularly and independently compact the $nodes_k$ array. This can also be done by "compacting to the left" over nodes marked for deletion but, this now means that what was node $\langle k|p\rangle$ is now node $\langle k|q\rangle$, with $q < p$. When performing this compaction, we build a temporary indirection array *old2new* of size equal to the old number of nodes, so that $old2new[p] = q$. Then, once compaction is completed, we scan the arcs in the level above, $arcs_{k+1}$, and change each occurrence of p into q using a lookup into *old2new*. Finally, *old2new* is destroyed when this scan has completed.

5.6 Overflow of potential local state spaces

Our new algorithm eliminates the need to specify additional constraints for any formalism where each state can reach a finite number of states in a single step. A subtle problem remains, however, if an infinite number of states can be reached in one step. For example, in Generalized Stochastic Petri Nets [1], *immediate* transitions, such as v in Fig. 19, are processed not by themselves, but as events that can take place instantaneously after the firing of *timed* transitions, such as t and u (somewhat analogous to *internal* events in process algebra). In Fig. 19, we partition the net into submodel 2, containing place p, and submodel 1, containing places q and r.

The initial local states are (p^0) and (q^1r^0) respectively. When the latter state is confirmed into $\mathcal{S}_1$, an explicit local exploration begins. Transition t can fire in submodel 1 in isolation, leading to marking (q^2r^0). This enables immediate transition v which, processed right away as part of the firing of t, leads to markings (q^2r^1), (q^2r^2), (q^2r^3), ... and so on. Thus, the explicit local exploration fails with an overflow in place r, while a traditional explicit global exploration would not, since it would never reach a global marking with two tokens in q. This situation is quite artificial, however. It can occur only if the formalism allows a state to reach an infinite number of states in one "timed step".

6 Results

We conducted two series of experiments on a 3 Ghz Pentium IV workstation with 1GB of memory. The first experiment compares the various exploration algorithms presented in Section 3, which we implemented in our tool SMART using MDDs and Kronecker matrices. The second experiment compares the saturation algorithm of SMART with the traditional approach of NuMSV [20], a symbolic verifier built on top of the CUDD library [45].

Our examples include four models parametrized by an integer N: dining philosophers and slotted ring [38], round robin mutual exclusion [28], and flexible manufacturing system (FMS) [19]. Detailed descriptions can be found in [11]. The first three models are safe Petri nets: N affects the height of the MDD but not the size of the local state spaces (except for $\mathcal{S}_1$ in the round robin model, which grows linearly in N). The FMS has instead an MDD with fixed height but size of the nodes increasing linearly in N.

Table 1 contains three groups of columns. To the left are the model parameters (N and $|\mathcal{S}|$, the state space size), while the runtimes and memory requirements for seven different exploration strategies are listed in the middle and right blocks of columns, respectively. The strategies considered are the two variants of breadth-first search, *BfSsGen* and *AllBfSsGen*, the two variants of BFS with chaining, *ChSsGen* and *AllChSsGen*, the forwarding arcs approach of [13] to implement update-in-place, the saturation algorithm with pregeneration of the local state spaces [14], and the saturation on-the-fly algorithm [15]. The last column lists the memory needed by the final MDD, encoding the entire state space, this quantity is of course independent of the algorithm used.

The results show a steady performance gain that "chronologically" follows the series of improvements presented in this paper. In a nutshell, there are three huge "leaps" in technology: the idea of chaining, the exploitation of event locality in the forwarding arcs approach, and the radical change in iteration strategy introduced by saturation. The chaining technique improves on traditional BFS, the in-place-updates and locality improve on chaining, while the saturation strategy is vastly superior to all predecessors by several orders of magnitude, both in terms of time and memory consumption.

On the issue of using just the MDD encoding of frontier set (strictly new states) versus the encoding of the entire current set in the BFS iterations, we found a rather surprising fact. Not only the unanimous preference for the frontier set approach seems to be unjustified, but, at least for the class of asynchronous systems studied here, the second approach that uses all states performs consistently better (even for the dining philosopher model, where *ChSsGen* and *AllChSsGen* have similar timing performance, *AllChSsGen* uses substantially less peak memory). This is counterintuitive only in the context of explicit methods. Decision diagrams exploit structure in the encoded sets, which seems to be poor in the frontier set.

Regarding our two saturation algorithms, the results demonstrate that the overhead of the on-the-fly algorithm versus pregeneration is acceptable. Moreover, the additional per-node memory overhead required to manage dynamically-sized nodes at a given level k can be offset by the ability to store nodes with $m < n_k$ arcs (because they were created when $\mathcal{S}_k$ contained only m states, or because the last $n_k - m$ arcs point to $\langle k-1|0\rangle$ and are truncated). In fact, for the FMS model, this results in smaller memory requirements than with pregeneration, suggesting that the use of sparse nodes is advantageous in models with large local state spaces. Even if our on-the-fly implementation is not yet as optimized as that of pregeneration, the runtime of the on-the-fly algorithm is still excellent, at most double (for the FMS model) and consistently under 70% overhead compared to the pregeneration version for the other models. This is a good tradeoff, given the increase in modeling ease and safety afforded by not having to worry about local state space generation in isolation.

Table 2 lists the peak and final memory and the runtime for SMART using saturation on-the-fly with NuSMV. For comparison's sake, we assume that a BDD node in NuSMV uses 16 bytes. To be fair, we point out that our memory consumption refers to the MDD only, while we believe that the number of nodes reported by NuSMV includes also those for the next-state function; however, our Kronecker encoding for $\mathcal{N}$ is extremely efficient, requiring at most 300KB in any model. Memory for the operation caches is not included in either our or NuSMV

Table 1. Comparing state-space generation algorithms in SMART ("—" indicates excessive time or memory requirements)

N	Reachable	Time (sec)							Memory (MB)							
	states	*Bf*	*AllBf*	*Ch*	*AllCh*	*Fwd*	*Sat*	*Otf*	*Bf*	*AllBf*	*Ch*	*AllCh*	*Fwd*	*Sat*	*Otf*	final
Dining Philosophers: $K=N$, $\lvert\mathcal{S}_k\rvert=34$ for all k																
50	2.2×10^{31}	37.6	36.8	1.3	1.3	0.1	0.0	0.0	146.8	131.6	2.2	2.2	0.0	0.0	0.0	0.0
100	5.0×10^{62}	644.1	630.4	5.4	5.3	0.3	0.2	0.2	> 999.9	> 999.9	8.9	8.9	0.2	0.2	0.2	0.0
1000	9.2×10^{626}	—	—	895.4	915.5	4.6	1.1	1.9	—	—	895.2	895.0	0.5	0.4	0.4	0.3
10000	4.3×10^{6269}	—	—	—	—	704.0	84.4	129.7	—	—	—	—	5.5	3.9	3.7	3.1
Slotted Ring Network: $K=N$, $\lvert\mathcal{S}_k\rvert=15$ for all k																
5	5.3×10^{4}	0.2	0.3	0.1	0.1	0.0	0.0	0.0	0.8	1.1	0.3	0.2	0.0	0.0	0.0	0.0
10	8.3×10^{9}	21.5	24.1	2.1	1.2	0.1	0.0	0.0	39.0	45.0	5.7	3.3	0.0	0.0	0.0	0.0
15	1.5×10^{15}	745.4	771.5	18.5	8.9	0.5	0.1	0.2	344.3	375.4	35.1	20.2	0.1	0.1	0.1	0.0
50	1.7×10^{52}	—	—	—	—	120.3	2.9	4.4	—	—	—	—	4.2	2.0	2.2	0.1
100	2.6×10^{105}	—	—	—	—	4976.1	21.6	33.9	—	—	—	—	44.5	14.8	16.1	0.4
Round Robin Mutual Exclusion: $K=N+1$, $\lvert\mathcal{S}_k\rvert=10$ for all k except $\lvert\mathcal{S}_1\rvert=N+1$																
10	2.3×10^{4}	0.2	0.3	0.1	0.1	0.0	0.0	0.0	0.6	1.2	0.1	0.1	0.0	0.0	0.0	0.0
20	4.7×10^{7}	2.7	4.4	0.3	0.3	0.1	0.0	0.0	5.9	12.8	0.5	0.5	0.0	0.0	0.0	0.0
30	7.2×10^{10}	16.4	26.7	0.7	0.7	0.3	0.1	0.1	22.7	48.2	1.3	1.1	0.1	0.0	0.0	0.0
50	1.3×10^{17}	263.2	427.6	2.9	2.8	1.0	0.2	0.2	126.7	257.7	4.3	3.8	0.2	0.1	0.1	0.1
100	2.9×10^{32}	—	—	22.1	19.3	7.7	1.2	1.7	—	—	22.1	20.0	0.7	0.4	0.4	0.4
200	7.2×10^{62}	—	—	—	—	58.1	10.9	13.6	—	—	—	—	2.7	1.4	1.4	1.4
FMS: $K=19$, $\lvert\mathcal{S}_k\rvert=N+1$ for all k except $\lvert\mathcal{S}_{17}\rvert=4$, $\lvert\mathcal{S}_{12}\rvert=3$, $\lvert\mathcal{S}_7\rvert=2$																
5	2.9×10^{6}	0.7	0.7	0.1	0.1	0.0	0.0	0.0	2.6	2.2	0.4	0.2	0.0	0.0	0.0	0.0
10	2.5×10^{9}	7.0	5.8	0.5	0.3	0.1	0.0	0.0	18.2	14.7	2.3	1.3	0.0	0.0	0.0	0.0
15	2.2×10^{11}	41.0	30.5	1.9	1.0	0.2	0.0	0.0	61.9	48.8	8.0	4.3	0.1	0.1	0.1	0.0
20	6.0×10^{12}	183.6	126.0	5.3	2.3	0.5	0.1	0.1	154.0	119.4	20.2	10.3	0.1	0.1	0.1	0.1
25	8.5×10^{13}	677.2	437.9	12.9	5.1	1.0	0.1	0.1	319.7	245.3	42.7	21.2	0.3	0.2	0.2	0.1
150	4.8×10^{23}	—	—	—	—	—	8.4	16.4	—	—	—	—	—	30.7	16.5	15.8

results (for our algorithms, caches never exceeded 2MB on these examples). Overall, SMART can handle significantly larger models than NuSMV. For each model, the last row corresponds to the largest value of N for which generation was possible with NuSMV. For these values, saturation shows speed-ups over 100,000 and memory reductions over 1,000.

7 Conclusions

We presented a novel approach for building the state space of asynchronous systems using MDDs to store sets of states and Kronecker operators on boolean matrices to store the next-state function. This avoids encoding the global next-state function as a single MDD, and disjunctively splits it instead according to the asynchronous events in the system, and then conjunctively according to the state variables. The resulting encoding gives us the opportunity to recognize *locality* in the effect of each event on each state variable, and the freedom to choose the order in which to fire events, that is, the fixed-point iteration strategy.

The resulting algorithm, *saturation*, was shown to be orders of magnitude faster and more memory efficient than traditional symbolic methods that discover the state space in breadth-first order. Furthermore, we presented an extended version of this algorithm that can be used when the state space is known to be finite but the possible range of the state variables is unknown. The resulting data structure, MDDs whose nodes can be expanded at runtime, may have useful applications beyond state-space generation.

We stress that saturation and its related data structures have also been employed to improve *symbolic CTL model checking* [18] and the computation of the *distance function* of every state from a set of states [17], which is useful for the efficient generation of shortest witnesses to an existential state query (i.e., EF in CTL).

One limitation of the approach we presented is its reliance on a *Kronecker consistent* partition of the model into submodels. While some formalisms always satisfy this condition for any partition of the model, others do not. For a model expressed in such a formalism, then, Kronecker consistency may be satisfied only by merging submodels or refining events. Given the effectiveness of saturation, we believe that an important research endeavor is to investigate ways to lift this limitation.

Table 2. Results for SMART vs. NuSMV

N	Reachable states	Final memory (KB)		Peak memory (KB)		Time (sec)	
		Otf	NuSMV	*Otf*	NuSMV	*Otf*	NuSMV
Dining Philosophers: $K=N$, $\lvert\mathcal{S}_k\rvert=34$ for all k							
20	3.46×10^{12}	4	4,178	5	4,192	0.01	0.4
50	2.23×10^{31}	11	8,847	14	8,863	0.03	13.1
100	4.97×10^{62}	24	8,891	28	15,256	0.06	990.8
200	2.47×10^{125}	48	21,618	57	59,423	0.15	18,129.3
Slotted Ring Network: $K=N$, $\lvert\mathcal{S}_k\rvert=15$ for all k							
5	5.39×10^{4}	1	502	5	507	0.01	0.1
10	8.29×10^{9}	5	4,332	28	8,863	0.06	6.1
15	1.46×10^{15}	10	771	80	11,054	0.18	2,853.1
Round Robin Mutual Exclusion: $K=N+1$, $\lvert\mathcal{S}_k\rvert=10$ for all k except $\lvert\mathcal{S}_1\rvert=N+1$							
10	2.30×10^{4}	5	917	6	932	0.01	0.2
20	4.72×10^{7}	18	5,980	20	5,985	0.04	1.4
30	7.25×10^{10}	37	2,222	41	8,716	0.09	5.6
100	2.85×10^{32}	357	13,789	372	21,814	2.11	2,836.5
FMS: $K=19$, $\lvert\mathcal{S}_k\rvert=N+1$ for all k except $\lvert\mathcal{S}_{17}\rvert=4$, $\lvert\mathcal{S}_{12}\rvert=3$, $\lvert\mathcal{S}_7\rvert=2$							
5	1.92×10^{4}	5	2,113	6	2,126	0.01	1.0
10	2.50×10^{9}	16	1,152	26	8,928	0.02	41.6
25	8.54×10^{13}	86	17,045	163	152,253	0.16	17,321.9

References

1. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, New York, 1995.
2. V. Amoia, G. De Micheli, and M. Santomauro. Computer-oriented formulation of transition-rate matrices via Kronecker algebra. *IEEE Trans. Rel.*, 30:123–132, June 1981.
3. J. A. Bergstra, A. Ponse, and S. A. Smolka. *Handbook of Process Algebra*. Elsevier Science, 2001.
4. R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *Proc. DAC*, pages 29–34, Los Angeles, CA, USA, 2000. ACM Press.
5. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *Computer Aided Verification*, pages 403–418, 2000.
6. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, Aug. 1986.
7. P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. Comp.*, 12(3):203–222, 2000.
8. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th Annual IEEE Symp. on Logic in Computer Science*, pages 428–439, Philadelphia, PA, 4–7 June 1990. IEEE Comp. Soc. Press.
9. J. R. Burch, E. M. Clarke, and D. 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, Aug. 1991. IFIP Transactions, North-Holland.
10. G. Cabodi, P. Camurati, and S. Quer. Improving symbolic traversals by means of activity profiles. In *Design Automation Conference*, pages 306–311, 1999.
11. G. Ciardo et al. SMART: Stochastic Model checking Analyzer for Reliability and Timing, User Manual. Available at http://www.cs.ucr.edu/∼ciardo/SMART/.
12. G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. In P. Kemper and W. H. Sanders, editors, *Proc. Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS 2794, pages 78–97, Urbana, IL, USA, Sept. 2003. Springer-Verlag.
13. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In M. Nielsen and D. Simpson, editors, *Proc. 21th Int. Conf. on Applications and Theory of Petri Nets*, LNCS 1825, pages 103–122, Aarhus, Denmark, June 2000. Springer-Verlag.
14. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In T. Margaria and W. Yi, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2031, pages 328–342, Genova, Italy, Apr. 2001. Springer-Verlag.
15. G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In H. Garavel and J. Hatcliff, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2619, pages 379–393, Warsaw, Poland, Apr. 2003. Springer-Verlag.
16. G. Ciardo and A. S. Miner. Storage alternatives for large structured state spaces. In R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, editors, *Proc. 9th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS 1245, pages 44–57, St. Malo, France, June 1997. Springer-Verlag.
17. G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest

paths. In M. D. Aagaard and J. W. O'Leary, editors, *Proc. Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, LNCS 2517, pages 256–273, Portland, OR, USA, Nov. 2002. Springer-Verlag.

18. G. Ciardo and R. Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. In W. Hunt, Jr. and F. Somenzi, editors, *Computer Aided Verification (CAV'03)*, LNCS 2725, pages 40–53, Boulder, CO, USA, July 2003. Springer-Verlag.
19. G. Ciardo and K. S. Trivedi. A decomposition approach for stochastic reward net models. *Perf. Eval.*, 18(1):37–59, 1993.
20. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *CAV '99*, LNCS 1633, pages 495–499. Springer-Verlag, 1999.
21. E. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in model checking. In *CAV '93*, LNCS 697, pages 450–462. Springer-Verlag, 1993.
22. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
23. O. Coudert and J. C. Madre. Symbolic computation of the valid states of a sequential machine: algorithms and discussion. In *1991 Int. Workshop on Formal Methods in VLSI Design*, pages 1–19, Miami, FL, USA, 1991.
24. A. Geser, J. Knoop, G. Lüttgen, B. Steffen, and O. Rüthing. Chaotic fixed point iterations. Technical Report MIP-9403, Univ. of Passau, 1994.
25. P. Godefroid. *Partial-order Methods for the Verification of Concurrent Systems – An Approach to the State-explosion Problem*, volume 1032 of *LNCS 1032*. Springer-Verlag, 1996.
26. P. Godefroid and D. E. Long. Symbolic protocol verification with queue BDDs. *Formal Methods in System Design*, 14(3):257–271, May 1999.
27. S. Graf and B. Steffen. Compositional minimization of finite state systems. In E. M. Clarke and R. P. Kurshan, editors, *Proc. CAV*, LNCS 531, pages 186–196, 1990.
28. S. Graf, B. Steffen, and G. Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Formal Asp. of Comp.*, 8(5):607–616, 1996.
29. O. Grumberg, T. Heyman, and A. Schuster. A work-efficient distributed algorithm for reachability analysis. In *Computer Aided Verification 2003*, July 2003.
30. J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In E. Brinksma, R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019, pages 89–110. Springer, 1995.
31. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
32. G. Holzmann and D. Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.
33. 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.
34. S. Kimura and E. M. Clarke. A parallel algorithm for constructing binary decision diagrams. In *Proc. Int. Conf. on Computer Design (ICCD)*, pages 220–223, Cambridge, MA, Sept. 1990. IEEE Comp. Soc. Press.
35. J.-P. Krimm and L. Mounier. Compositional state space generation from Lotos programs. In E. Brinksma, editor, *Proc. TACAS*, LNCS 1217, pages 239–258, Enschede, The Netherlands, Apr. 1997. Springer-Verlag.
36. A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In H. Kleijn and S. 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.
37. T. Murata. Petri Nets: properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–579, Apr. 1989.
38. E. Pastor, O. Roig, J. Cortadella, and R. Badia. Petri net analysis using boolean manipulation. In R. Valette, editor, *Proc. 15th Int. Conf. on Applications and Theory of Petri Nets*, LNCS 815, pages 416–435, Zaragoza, Spain, June 1994. Springer-Verlag.
39. S. Pissanetzky. *Sparse Matrix Technology.* Academic Press, 1984.
40. B. Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. ACM SIGMETRICS*, pages 147–153, Austin, TX, USA, May 1985.
41. H. Preuss and A. Srivastav. Blockwise variable orderings for shared BDDs. In *MFCS: Symp. on Mathematical Foundations of Computer Science*, 1998.
42. K. Ravi and F. Somenzi. High-density reachability analysis. In *ICCAD '95*, pages 154–158. IEEE Computer Society Press, 1995.
43. O. Roig, J. Cortadella, and E. Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri nets. In G. De Michelis and M. Diaz, editors, *Proc. 16th Int. Conf. on Applications and Theory of Petri Nets, Turin, Italy*, LNCS 935, pages 374–391. Springer-Verlag, June 1995.
44. M. Solé and E. Pastor. Traversal techniques for concurrent systems. *Lecture Notes in Computer Science*, 2517:220–237, 2002.
45. F. Somenzi. CUDD: CU Decision Diagram Package, Release 2.3.1. http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html.
46. A. Valmari. A stubborn attack on the state explosion problem. In *CAV '90*, pages 25–42. AMS, 1990.
47. B. Yang and D. R. O'Hallaron. Parallel breadth-first BDD construction. In *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'97)*, pages 145–156, Las Vegas, NV, June 1997.