Ten Years of Saturation: a Petri Net Perspective

Gianfranco Ciardo, Yang Zhao, and Xiaoqing Jin

Department of Computer Science and Engineering, University of California, Riverside
ciardo@cs.ucr.edu zhaoy@cs.ucr.edu jinx@cs.ucr.edu

Abstract. Due to their appealing conceptual simplicity and availability of computer tools for their analysis, Petri nets are widely used to model discrete-event systems in many areas of engineering. However, the computational resources required to carry out the analysis of a Petri net model are often enormous, hindering their practical impact. In this survey, we consider how *symbolic* methods based on the use of *decision diagrams* can greatly increase the size of Petri nets that an ordinary computer can reasonably tackle. In particular, we present this survey from the perspective of the efficient *saturation* method we proposed a decade ago, and introduce along the way the most appropriate classes of decision diagrams to answer important Petri net questions, from reachability to CTL model checking and counterexample generation, from p-semiflow computation to the solution of timed or Markovian nets.

1 Introduction

Since their definition 50 years ago [45], Petri nets have become a well known formalism to model discrete-state systems and many algorithms have been developed for their analysis [46]. However, even when the net is bounded, algorithms that manipulate the set of reachable markings usually require enormous computational resources when run on practical models.

Binary decision diagrams, introduced 25 year ago [8], have had a tremendous impact on industrial hardware verification, due to their ability to *symbolically* encode complex boolean function on enormous domains [9, 39]. More precisely, while their worst-case complexity remains exponential in the number of boolean variables, many cases of practical interest have much better behavior, as attested by numerous successful verification and bug finding studies [24].

In this survey, we explore how decision diagram technology can greatly enhance our capability to analyze various classes of Petri nets. We do so in light of an algorithm we introduced 10 years ago, *saturation* [13], which tends to perform extremely well when applied to discrete-event systems having multiple asynchronous events that depend and affect only relatively small subsystems, a prime example of which is, of course, Petri nets. To provide a solid introductory treatment to decision diagrams, their manipulation algorithms, and their application to Petri net problems, this survey contains many illustrations, detailed pseudocode for representative algorithms, an extensive set of models, and memory and time results obtained when running the algorithms on these models.

The rest of our survey is organized as follows. Section 2 discusses *reachability-set generation* for Petri nets, from safe nets up to extended Petri nets, focusing on efficient canonical forms of boolean-valued decision diagrams and the use of acceleration techniques to improve efficiency, including *saturation*. Section 3 considers *CTL model checking* for Petri nets, which benefits from the techniques introduced for reachability; to generate shortest CTL witnesses/counterexamples or bound exploration, we employ instead a form of *edge-valued decision diagrams* able to encode partial integer-valued functions, which can nevertheless be used in conjunction with saturation. Section 4 moves to non-boolean Petri net questions: *p-semiflows computation*, using zero-suppressed decision diagrams; *timed and earliest reachability* for a class of integer-timed Petri nets, using a combination of boolean and integer-valued decision diagrams; *stationary analysis of generalized stochastic Petri nets*, for which the generation of the transition rate matrix using real-valued decision diagrams is usually quite efficient, but for which an efficient fully-symbolic exact solution still eludes us; and *heuristic derivation of good variable orders* for the decision diagram variables based on the structural characteristics of the Petri net. Section 5 presents a brief list of available decision diagram libraries and tools using decision diagram technology.

2 Reachability-set generation for Petri nets

This section recalls first the standard definition of Petri nets and later an extended definition for self-modifying nets, introduces the most basic classes of decision diagrams, BDDs and MDDs, and uses them to generate a symbolic encoding of the reachability set. In addition to the simpler breadth-first approach, it introduces more sophisticated fixpoint iteration strategies, chaining and saturation, which can provide much greater memory and runtime efficiency.

2.1 Petri nets

We use the standard definition of a Petri net as a tuple $(\mathcal{P},\mathcal{T},\mathbf{D}^-,\mathbf{D}^+,\mathbf{i}_{init})$ where:

- $\mathcal{P}$ is a set of *places*, drawn as circles, and $\mathcal{T}$ is a set of *transitions*, drawn as rectangles, satisfying $\mathcal{P} \cap \mathcal{T} = \emptyset$.
- $\mathbf{D}^- : \mathcal{P} \times \mathcal{T} \to \mathbb{N}$ and $\mathbf{D}^+ : \mathcal{P} \times \mathcal{T} \to \mathbb{N}$ are the *input arc* and the *output arc* cardinalities, respectively.
- $\mathbf{i}_{init} \in \mathbb{N}^{|\mathcal{P}|}$ is the *initial marking*, specifying a number of *tokens* initially present in each place.

If the current marking is $\mathbf{i} \in \mathbb{N}^{|\mathcal{P}|}$, we say that $\alpha \in \mathcal{T}$ is *enabled* in $\mathbf{i}$, written $\alpha \in \mathcal{T}(\mathbf{i})$, iff $\forall p \in \mathcal{P}, \mathbf{D}^-_{p,\alpha} \leq i_p$. Then, $\alpha \in \mathcal{T}(\mathbf{i})$ can *fire*, changing the marking to $\mathbf{j}$, written $\mathbf{i} \xrightarrow{\alpha} \mathbf{j}$, satisfying $\forall p \in \mathcal{P}, j_p = i_p - \mathbf{D}^-_{p,\alpha} + \mathbf{D}^+_{p,\alpha}$. A Petri net is a special case of a discrete-state system with a *potential state space* $\mathcal{X}_{pot} = \mathbb{N}^{|\mathcal{P}|}$, a *next-state* or *forward function* $\mathcal{N} : \mathcal{X}_{pot} \to 2^{\mathcal{X}_{pot}}$ given by the union $\mathcal{N} = \bigcup_{\alpha \in \mathcal{T}} \mathcal{N}_\alpha$ of the forward functions for each Petri net transition, where $\mathcal{N}_\alpha(\mathbf{i}) = \emptyset$ if $\alpha \notin \mathcal{T}(\mathbf{i})$,

while $\mathcal{N}_\alpha(\mathbf{i}) = \{\mathbf{j}\}$ if instead $\mathbf{i} \xrightarrow{\alpha} \mathbf{j}$, and an *initial state set* $\mathcal{X}_{init} = \{\mathbf{i}_{init}\}$. Indeed, we could have defined the Petri net with an arbitrary initial set of markings $\mathcal{X}_{init} \subseteq \mathbb{N}^{|\mathcal{P}|}$, and let the net nondeterministically start in one of these markings. The *reachable state space* $\mathcal{X}_{rch}$, or *reachability set*, of such a model is then defined as the smallest set $\mathcal{X} \subseteq \mathcal{X}_{pot}$ containing $\mathcal{X}_{init}$ and satisfying either the *recursive definition* $\mathbf{i} \in \mathcal{X} \wedge \mathbf{j} \in \mathcal{N}(\mathbf{i}) \;\Rightarrow\; \mathbf{j} \in \mathcal{X}$, which is the base for *explicit* state-space generation methods, or, equivalently, the *fixpoint equation* $\mathcal{X} = \mathcal{X} \cup \mathcal{N}(\mathcal{X})$, which is the base for *symbolic* state-space generation methods. Either way, we can write

$$\mathcal{X}_{rch} = \mathcal{X}_{init} \cup \mathcal{N}(\mathcal{X}_{init}) \cup \mathcal{N}^2(\mathcal{X}_{init}) \cup \mathcal{N}^3(\mathcal{X}_{init}) \cup \cdots = \mathcal{N}^*(\mathcal{X}_{init}).$$

2.2 Binary decision diagrams and their operations

An *(ordered) binary decision diagram* (BDD) over the sequence of *domain* variables $(v_L, \ldots, v_1)$, with an order $v_l \succ v_k$ iff $l > k$ defined on them, is an acyclic directed edge-labeled graph where:

- The only *terminal* nodes can be **0** and **1**, and are associated with the *range* variable $\mathbf{0}.var = \mathbf{1}.var = v_0$, satisfying $v_k \succ v_0$ for any domain variable v_k.
- A *nonterminal* node p is associated with a domain variable $p.var$.
- A nonterminal node p has two outgoing edges labeled 0 and 1, pointing to *children* denoted respectively $p[0]$ and $p[1]$.
- The variable of the children is lower than that of p, that is, $p.var \succ p[0].var$ and $p.var \succ p[1].var$.

Node p with $p.var = v_k$ encodes function $f_p : \mathbb{B}^L \to \mathbb{B}$, recursively defined by

$$f_p(i_L, ..., i_1) = \begin{cases} p & \text{if } k = 0 \\ f_{p[i_k]}(i_L, ..., i_1) & \text{if } k > 0 \end{cases}$$

(we write f_p as a function of L variables even when $k < L$, to stress that *any* variable x_h not explicitly appearing on a path from p to a terminal node is a "don't care" for f_p, regardless of whether $h < k$ or $h > k$).

We restrict ourselves to *canonical* forms of decision diagrams, where each function that can be encoded by a given class of decision diagrams has a unique representation in that class. For BDDs, canonicity is achieved by requiring that

1. There is no *duplicate*: if $p.var = q.var$, $p[0] = q[0]$, and $p[1] = q[1]$, then $p = q$.
2. One of the following forms holds.
   Quasi-reduced form: there is no variable skipping, i.e., if $p.var = v_k$, then $p[0].var = p[1].var = v_{k-1}$ and $k = L$ if p is a root (has no incoming arcs).
   Fully-reduced form: there is maximum variable skipping, i.e., no *redundant* node p exists, with $p[0] = p[1]$.

Fig. 1 shows a quasi-reduced and a fully-reduced BDD encoding the same function over (v_4, v_3, v_2, v_1), redundant nodes are indicated in grey.

Both canonical versions enjoy desirable properties: if functions f and g are encoded using BDDs, then *satisfiability*, "$f \neq 0$?", and *equivalence*, "$f = g$?",

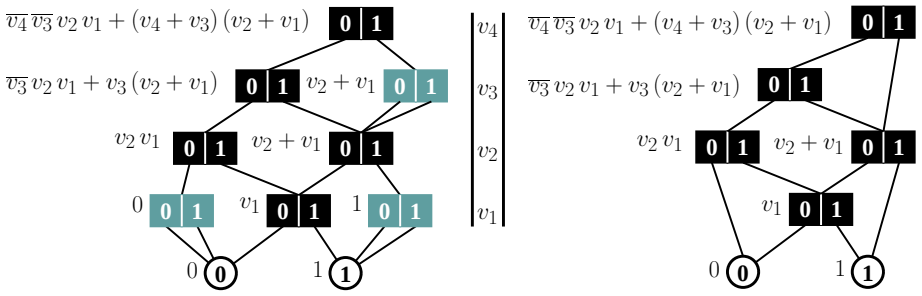

Fig. 1. Quasi-reduced (left) and fully-reduced (right) BDDs for the same function.

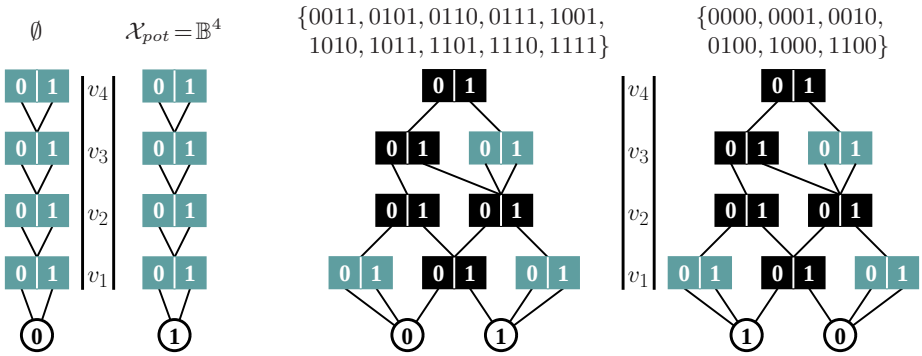

Fig. 2. Two sets and their complement, encoded as quasi-reduced BDDs.

can be answered in $O(1)$ time, while the BDD encoding the *conjunction* $f \wedge g$ or the *disjunction* $f \vee g$ can be built in $O(||f|| \times ||g||)$ time and space, if using the fully-reduced form, or $\sum_{L \geq k \geq 1} O(||f||_k \times ||g||_k)$ time and space, if using the quasi-reduced form, where $||f||$ is the number of nodes in the BDD encoding f and $||f||_k$ is the number of nodes associated with v_k in the BDD encoding f.

We can encode a set $\mathcal{Y} \subseteq \mathbb{B}^L$ as a BDD p through its *characteristic function*, so that $\mathbf{i} = (i_L, ..., i_1) \in \mathcal{Y} \;\Leftrightarrow\; f_p(i_L, ..., i_1) = 1$. The *size of the set* encoded by the BDD rooted at p is not directly related to the *size of the BDD* itself. For example, any set requires as many nodes as its complement, as shown in Fig. 2.

Algorithms for the manipulation of decision diagrams follow a typical recursive style. For example, Fig. 3 shows the pseudocode for the union operation on two sets encoded by fully-reduced BDDs p and q (i.e., the logical "or" of their characteristic functions). The pseudocode for *Intersection* or *Diff* (set difference) differs from that for *Union* only in the terminal cases. For *Intersection*(p, q), we return q if $q = \mathbf{0}$ or $p = \mathbf{1}$ and p if $q = \mathbf{1}$ or $p = \mathbf{0}$, while the case $p = q$, which might arise before reaching the terminal nodes, remains the same; for *Diff*(p, q), we return $\mathbf{0}$ if $p = \mathbf{0}$ or $q = \mathbf{1}$ and $\mathbf{1}$ if $q = \mathbf{0}$ and $p = \mathbf{1}$, while we return $\mathbf{0}$ if $p = q$. Another fundamental operation is the *relational product* which, given a BDD on $(v_L, ..., v_1)$ rooted at p encoding a set $\mathcal{Y} \subseteq \mathcal{X}_{pot}$ and a BDD on $(v_L, v'_L, ..., v_1, v'_1)$ rooted at r encoding a function $\mathcal{N} : \mathcal{X}_{pot} \rightarrow 2^{\mathcal{X}_{pot}}$, as the set of pairs $(\mathbf{i}, \mathbf{j})$

```
bdd Union(bdd p, bdd q) is                                    • fully-reduced version
 1 if p = 0 or q = 1 then return q;
 2 if q = 0 or p = 1 then return p;
 3 if p = q then return p;
 4 if Cache contains entry ⟨Union, {p, q} : r⟩ then return r;
 5 if p.var = q.var then
 6    r ← UniqueTableInsert(p.var, Union(p[0], q[0]), Union(p[1], q[1]));
 7 else if p.var ≻ q.var then
 8    r ← UniqueTableInsert(p.var, Union(p[0], q), Union(p[1], q));
 9 else since q.var ≻ p.var then
10    r ← UniqueTableInsert(q.var, Union(p, q[0]), Union(p, q[1]));
11 enter ⟨Union, {p, q} : r⟩ in Cache;
12 return r;
```

Fig. 3. The union operation on fully-reduced BDDs.

such that $\mathbf{j} \in \mathcal{N}(\mathbf{i})$, returns the root of the BDD on $(v_L, ..., v_1)$ encoding the set $\{\mathbf{j} : \exists \mathbf{i} \in \mathcal{Y} \ \wedge \ \mathbf{j} \in \mathcal{N}(\mathbf{i})\}$. An *interleaved order* is usually adopted for $\mathcal{N}$, that is, if $\mathbf{j} \in \mathcal{N}(\mathbf{i})$, then the BDD encoding $\mathcal{N}$ contains a path corresponding to $(i_L, j_L, ..., i_1, j_1)$ from r to $\mathbf{1}$, except that, of course, some of the values may be skipped if the fully-reduced form is used. Fig. 4 shows this function assuming quasi-reduced BDDs. As it can be seen, this results in simpler code, as the nodes in a recursive call, p and r in our case, are associated with the same variable.

We conclude this brief introduction to BDDs by observing that, to efficiently ensure canonicity, all decision diagram algorithms use a *Unique Table* (a hash table) which, given a search key consisting of a node's variable v_k and the *node_id* (e.g., the pointer to a unique memory location) of each child returns the *node_id* of either an existing node (if it exists, to avoid duplicates) or of a newly created node associated with v_k with those children. Since the manipulation is performed recursively bottom-up, children are already in the Unique Table when looking up their parent. Furthermore, to achieve polynomial complexity, the algorithms also use an *Operation Cache* (also a hash table) which, given a search key consisting of an *op_code* and the sequence of operands' *node_id*s, returns the *node_id* of the result, if it has been previously computed. These two hash tables have different requirements with respect to managing collisions on the hash value. The Unique Table must be *lossless*, while a lossy implementation for the Operation Cache is acceptable, as this at worst reduces efficiency.

2.3 Symbolic reachability-set generation for safe Petri nets

If a Petri net is *safe* (each place contains at most one token), $\mathcal{X}_{pot} = \mathbb{B}^L$, where $L = |\mathcal{P}|$, and we can store any set of markings $\mathcal{Y} \subseteq \mathcal{X}_{pot} = \mathbb{B}^L$ for such a Petri net with an L-variable BDD, and any relation over $\mathcal{X}_{pot}$, or function $\mathcal{X}_{pot} \to 2^{\mathcal{X}_{pot}}$, such as $\mathcal{N}$, with a $2L$-variable BDD. However, for safe nets, Pastor et al. showed how to generate the reachability set [43] by encoding $\mathcal{N}$ using $4|\mathcal{T}|$ boolean functions, each corresponding to a simple L-variable BDD describing:

- $APM_\alpha = \bigwedge_{p:\mathbf{D}^-_{p,\alpha}=1}(v_p = 1)$, i.e., all predecessor places of α are marked.

```
bdd RelProd(bdd p, bdd2 r) is                                  • quasi-reduced version
 1 if p = 0 or r = 0 then return 0;
 2 if p = 1 and r = 1 then return 1;
 3 if Cache contains entry ⟨RelProd, p, r : q⟩ then return q;
 4 q0 ← Union(RelProd(p[0], r[0][0]), RelProd(p[1], r[1][0]));
 5 q1 ← Union(RelProd(p[0], r[0][1]), RelProd(p[1], r[1][1]));
 6 q ← UniqueTableInsert(p.var, q0, q1);
 7 enter ⟨RelProd, p, r : q⟩ in Cache;
 8 return q;
```

Fig. 4. The relational product operation on quasi-reduced BDDs.

- $NPM_\alpha = \bigwedge_{p:\mathbf{D}^-p,\alpha=1}(v_p = 0)$, i.e., no predecessor place of α is marked.
- $ASM_\alpha = \bigwedge_{p:\mathbf{D}^+p,\alpha=1}(v_p = 1)$, i.e., all successor places of α are marked.
- $NSM_\alpha = \bigwedge_{p:\mathbf{D}^+p,\alpha=1}(v_p = 0)$, i.e., no successor place of α is marked.

The effect of transition α on a set of markings $\mathcal{U}$ can then be expressed as $\mathcal{N}_\alpha(\mathcal{U}) = (((\mathcal{U} \div APM_\alpha) \cdot NPM_\alpha) \div NSM_\alpha) \cdot ASM_\alpha$, where "$\cdot$" indicates boolean conjunction and "$\div$" indicates the *cofactor* operator, defined as follows: given a boolean function $f(v_L, \ldots, v_1)$ and a literal $v_k = i_k$, with $L \geq k \geq 1$ and $i_k \in \mathbb{B}$, the cofactor $f \div (v_k = i_k)$ is $f(v_L, \ldots, v_{k+1}, i_k, v_{k-1}, \ldots, v_1)$, and the extension to multiple literals, $f \div (v_{k_c} = i_{k_c}, \ldots, v_{k_1} = i_{k_1})$, is recursively defined as $f(v_L, \ldots, v_{k_c+1}, i_{k_c}, v_{k_c-1}, \ldots, v_1) \div (v_{k_{c-1}} = i_{k_{c-1}}, \ldots, v_{k_1} = i_{k_1})$.

If we instead follow a more traditional and general approach, $\mathcal{N}$ is stored either in *monolithic form* as a single $2L$-variable BDD, or in *disjunctively partition form* as $\bigcup_{\alpha\in\mathcal{T}} \mathcal{N}_\alpha$, where each $\mathcal{N}_\alpha$ is encoded as a $2L$-variable BDD. As the reachability set $\mathcal{X}_{rch}$ is the fixpoint of the iteration $\mathcal{X}_{init} \cup \mathcal{N}(\mathcal{X}_{init}) \cup \mathcal{N}(\mathcal{N}(\mathcal{X}_{init})) \cup \cdots$, we can compute it using a *breadth-first search* that only requires *Union*, *Diff*, and *RelProd* operations. Fig. 5 on the left shows the pseudocode for a traditional implementation of this algorithm, where sets and relations are encoded using BDDs, so that its runtime is proportional to the BDD sizes, not the size of the encoded sets and relations. Fig. 5 on the right shows an alternative approach that operates on a different sequence of sets: instead of applying the forward function $\mathcal{N}$ to the unexplored markings $\mathcal{U}$ (at the d^{th} iteration, the markings at distance exactly d from $\mathcal{X}_{init}$), the "*all*" version applies $\mathcal{N}$ to all the markings $\mathcal{O}$ known so far (at the d^{th} iteration, the markings at distance up to d from $\mathcal{X}_{init}$).

The decision diagrams manipulated by these algorithms, like many of those we will see in the following, encode an increasing amount of data as the fixpoint iterations progress. However, a fundamental, and sometimes counterintuitive, property of decision diagrams is that their size can swell and shrink during the iterations, so that the *final* size of, for example, the BDD encoding $\mathcal{X}_{rch}$ is often many orders of magnitude smaller than the *peak* BDD size encountered at some point along these iterations. This means that attempts to limit exploration (e.g., using partial order reduction techniques [52]) might be beneficial, or might instead hinder, decision-diagram-based approaches. One must report both final memory (for the decision diagram encoding the desired result) and peak memory requirements (including decision diagrams used in the computation and the hash

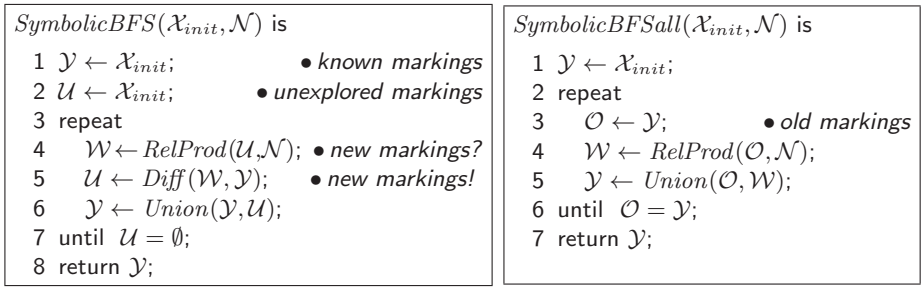

Fig. 5. Two versions of a symbolic BFS algorithm to generate the reachability set [15].

table for the operation cache and the unique table), as the latter determine when it is feasible to run the analysis on a given machine and strongly affect runtime.

2.4 Multiway decision diagrams

When v_k has finite but not necessarily boolean domain $\mathcal{X}_k = \{0,1,...,n_k - 1\}$ for some $n_k > 0$, we can use multiple boolean variables to encode it. Two common approaches are a standard *binary encoding* with $\lceil \log_2 n_k \rceil$ boolean variables, or a *one-hot encoding* with n_k boolean variables, exactly one of which is set to 1 in each reachable marking. Another alternative, which we adopt, is to directly use a natural extension of BDDs, (ordered) *multiway decision diagrams* (MDDs) [34]. The definition of an MDD is exactly as that of a BDD, except that the number of a node's children depends on its associated variable:

– For each $i_k \in \mathcal{X}_k$, a nonterminal node p associated with v_k has an outgoing edge labeled with i_k and pointing to a child $p[i_k]$.

The quasi-reduced and fully-reduced canonical forms are also exactly analogous to those for BDDs, except that, of course, p duplicates q if $p.var = q.var$ and $p[i_k] = q[i_k]$ for all $i_k \in \mathcal{X}_k$, and p is redundant if $p[0] = p[1] = \cdots = p[n_k - 1]$.

MDDs open an interesting possibility: while we might know (or even just simply hope) that $\mathcal{X}_{rch}$ is finite, we might not know a priori the size of each $\mathcal{X}_k$, i.e., the bound on the number of tokens in each place. In this more general setting, we only know that the reachability set satisfies $\mathcal{X}_{init} \subseteq \mathcal{X}_{rch} \subseteq \mathbb{N}^{|\mathcal{P}|}$. Thus, we need an MDD-based reachability-set generation approach where the size of each node p associated with domain variable v_k is *conceptually* infinite, as we do not know a priori which children $p[i_k]$ might be eventually needed to store the reachable marking (i.e., whether the place corresponding to v_k might ever contain i_k tokens). However, if the net is indeed bounded, only a finite number of these children will be on paths leading to the terminal **1** in the MDD encoding $\mathcal{X}_{rch}$. We then define a *sparse* canonical form, where a node has as many children as the number of edges pointing to nodes encoding a non-empty set and only nodes encoding the empty set can be classified as redundant. Fig. 6 shows the same function encoded as a quasi-reduced, fully-reduced, and sparse canonical MDD. In the first two forms, we must know that $\mathcal{X}_4 = \{0,1,2,3\}$, $\mathcal{X}_3 = \{0,1,2\}$,

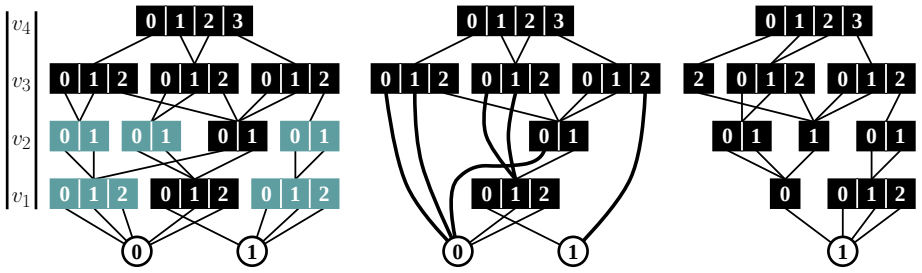

Fig. 6. Quasi-reduced, fully-reduced, and sparse canonical forms for MDDs.

$\mathcal{X}_2 = \{0, 1\}$, and $\mathcal{X}_1 = \{0, 1, 2\}$ while, in the third form, no such assumption is required and the resulting MDD does not contain the terminal **0** nor any node leading only to it (the two leftmost redundant nodes associated with variables v_2 and v_1 in the quasi-reduced form), but it contains all other redundant nodes. As the size of the nodes associated with a given v_k is variable, this requires a slightly more complex implementation, but this is usually a small price to pay compared to the greater flexibility and generality they provide.

2.5 Extended Petri nets

Now that we know how to encode arbitrary finite sets $\mathcal{Y} \subset \mathbb{N}^{|\mathcal{P}|}$ of markings using MDDs in sparse form, we can assume a more general extended Petri net model, for example one that allows *inhibitor arcs*:

– $\mathbf{D}^\circ : \mathcal{P} \times \mathcal{T} \to \mathbb{N} \cup \{\infty\}$ are the inhibitor arc cardinalities,

so that transition α is disabled in marking $\mathbf{i}$ if there is a place p such that $\mathbf{D}^\circ_{p,\alpha} \leq i_p$, or *self-modifying* behavior where arcs cardinalities depend on the current marking:

– $\mathbf{D}^-, \mathbf{D}^+ : \mathcal{P} \times \mathcal{T} \times \ \mathbb{N}^{|\mathcal{P}|} \to \mathbb{N}$ are the *marking-dependent* input and output arc cardinalities,

so that α is enabled in $\mathbf{i}$ iff, $\forall p \in \mathcal{P}$, $\mathbf{D}^-_{p,\alpha}(\mathbf{i}) \leq i_p$ and, if α fires, it leads to marking $\mathbf{j}$ satisfying $j_p = i_p - \mathbf{D}^-_{p,\alpha}(\mathbf{i}) + \mathbf{D}^+_{p,\alpha}(\mathbf{i})$. Both $\mathbf{D}^-_{p,\alpha}$ and $\mathbf{D}^+_{p,\alpha}$ are evaluated on the current, not the new, marking; also, these two extensions can be combined, allowing marking-dependent inhibitor arc cardinalities. It is well-known that inhibitor arcs alone suffice to achieve Turing-equivalence, while self-modifying behavior may or may not, depending on the type of functions allowed to specify arc cardinalities. However, since our approach is to simply run the symbolic reachability-set generation and "hope for the best" (i.e., for a finite $\mathcal{X}_{reach}$ that can be encoded in an MDD), these extensions are welcome from the point of view of modeling flexibility and ease.

We conclude this discussion of MDD encoding for general Petri nets by stressing that, while we have so far assumed that each Petri net place is mapped to an MDD variable, this is neither required in theory nor necessarily the most

efficient approach in practice. First of all, even when MDD variable v_k corresponds to a single Petri net place a with bound B_a, not all numbers of tokens $n_a \in \{0, ..., B_a\}$ might be possible in a (in particular, even B_a itself might be just an overestimate on the real bound). It is then best to map the different numbers of tokens *observed* in a during reachability-set generation to the indices 0, 1, and so on; then, $|\mathcal{X}_k| \leq B_a + 1$ will at the end count the distinct numbers of tokens achievable in a. More importantly, if we know that the range of values for the number of tokens in a is large, we might want to *split* this number over multiple MDD variables; this is of course required if we are using BDDs, as we already observed. On the other hand, if multiple places are highly correlated, we might want to *merge* them into the same variable; for example, if two places are *complementary*, i.e., the sum of the tokens in them is a constant, we clearly want to represent them with a single MDD variable as this does not increase the complexity of the encoding but reduces instead the height of the MDD.

Unfortunately, finding a good mapping of places to variables is a difficult problem, just as the related one about finding a good order for the variables, known to be NP-hard [7]. This is especially true for the more advanced algorithms we discuss later, which strongly rely on the structural dependencies between the Petri net transitions and the MDD variables.

2.6 Running examples for experiments

Fig. 7 presents the Petri net models used to obtain memory and runtime results for the various algorithms presented in this paper. All experiments are run on a server of Intel Xeon CPU 2.53GHz with 38G RAM under Linux 2.6.18

To study a Petri net with MDDs, we decompose the potential state space into the product of local state spaces, by partitioning the places into L *submodels*. Each (sub)marking of the k^{th} submodel is mapped to a local state in $\mathcal{X}_k$, so that, if the Petri net is bounded, the local state spaces $\mathcal{X}_k$ are finite. Hence, the L MDD variables correspond to the L submodels. The choice of partition affects the performance of our algorithms, and finding an optimal partition is non-trivial. In the experiments, we use a partition that works well in practice.

phils [44] models the dining-philosophers problem, where N philosophers sit around a table with a fork between each of them. To eat, a philosopher must acquire both the fork to his left and to his right. If all philosophers choose to take the fork to their left first, the model deadlocks (the same happens if all take the fork to their right). The entire model contains N subnets, the figure shows the i^{th} one, for some $i \in \{1, ..., N\}$; place $Fork_{\bmod (i,N)+1}$ is drawn with dashed lines because it is not part of this subnet. We group the places for two adjacent philosophers into a submodel, thus $L = N/2$ (with N even).

robin [28] models a round-robin resource sharing protocol, where N processes cyclically can access a resource. Again, the model has N subnets and the i^{th} subnet (shown in the figure) can access the resource only when its place $Pask_i$ contains the token initially in $Pask_1$. Place *Res*, containing the resource when idle, is drawn with bold lines because it is not part of any subnet. We group the

places for each process ($Pask_i, Pok_i, Psnd_i, Pwt_i$) into a submodel and assign place *Res* to a separate submodel. The number of domain variables is $L = N+1$.

fms [20] models a flexible manufacturing system where N parts of each of three types circulate around and are machined at stations. Parts of type 1 are machined by machine 1, of which there are three instances. Parts of type 2 are machined by machine 2, of which there is only one instance; however, when not working on parts of type 2, machine 2 can work on parts of type 3. Finally, parts of type 1 and 2, when completed, can be shipped (and replaced by new raw parts of the same type), just like parts of type 3, or might be joined by machine 3, of which there are two instances. Transitions shown in black are immediate [1] (from a logical standpoint, this simply means that they have priority over the other transitions, more details on the analysis of this type of models is given in Section 4.3). As N grows, only the possible number of tokens in (most of) the places grows, not the net itself. We group $\{M1, P1wM1, P1M1\}$, $\{M2, P2wM2, P2M2\}$ and $\{M3, P12wM3, P12M3\}$ into three submodels, and treat each remaining place as an individual submodel. Thus, $L = 16$.

slot [43] models a slotted ring transmission protocol where, as in **robin**, a token cycling around N subnets grants access to the medium. Unlike **robin**, though, this resource is not modeled, thus each subnet is just connected to its two neighbors through shared transitions, not to a globally shared place. We group the places for each subnet ($A_i, B_i, C_i, ..., G_i$) into a submodel and $L = N$.

kanban [51] models an assembly line using tokens (kanbans) to control the flow of parts to four assembly stations. Station 1, when completing work on a part, feeds that part to both stations 2 and 3 for further processing, thus transition s_{1_23} correspond to a "fork". When both stations 2 and 3 have completed parts waiting in places out_2 and out_3, the "join" transition s_{23_4} can pass them as a single token to station 4. As in **fms**, N controls the token population but not the size of the net. We group places for each station into a submodel and $L = 4$.

leader [32] models the distributed asynchronous protocol proposed by Itai and Rodeh to elect a leader among N processes in a unidirectional ring, by sending messages. The algorithm employs a randomization strategy to choose whether each process will continue running for election (my_pref=1) or not (my_pref=0). A process is eliminated from the race only if it chooses not to run and its closest active predecessor chooses to run. Once this happens, the process becomes inactive and only relays messages from active nodes. Termination is realized by having the active processes send a token around the ring to count the inactive nodes; if a process receives its own token with count $N - 1$, it knows to be the elected leader. The model contains N subnets (thus the number of places grows linearly in N, while the number of transitions grows quadratically in N) but, in addition, the token population in some places also grows linearly in N. We group places for each subnet into a submodel, thus $L = N$. We only show the automaton model due to the size and complexity of the net.

counter models a simple N-bit counter, which is incremented by one at each step, from 0 up to $2^N - 1$, then reset back to 0. We treat each place (bit) as a submodel and $L = N$.

queen models the placement of N queens on an $N \times N$ chessboard in such a way that they are not attacking each other. Places q_i, for $i \in \{1, ..., N\}$, represent the queens still to be placed on the chessboard, while places $p_{i,j}$, for $i, j \in \{1, ..., N\}$ represent the N^2 chessboard positions, thus are initially empty. The non-attacking restriction is achieved through inhibitor arcs (on the same column or either of the two diagonals, k ranges from 1 to $i-1$) plus the limitation on the row placed by having the single token in q_i be contended among transitions $t_{i,j}$, for $j \in \{1, ..., N\}$. In addition, we force the placement of the queens to happen in row order, from 1 to N, thus $t_{i,j}$ has inhibitor arcs from q_k, again for $k \in \{1, ..., i-1\}$. The number of places and transitions grows quadratically in N. We group the places for each queen (q_i and p_{ij}) into a submodel and $L = N$.

2.7 Accelerating the fixpoint computation: chaining and saturation

So far, we have considered only breadth-first methods, characterized by the fact that reachable markings are found strictly in order of their distance from $\mathcal{X}_{init}$. If we are willing to forgo this property, enormous runtime and memory improvements can often be achieved in practice.

The first such approach, *chaining* [47], observes that, when $\mathcal{N}$ is stored in disjunctively partitioned form using one BDD or MDD $\mathcal{N}_\alpha$ for each $\alpha \in \mathcal{T}$, the effect of statements 4 and 5 in Algorithm *SymbolicBFS* of Fig. 5 is exactly achieved with the statements:

$\mathcal{W} \leftarrow \emptyset$;
for each $\alpha \in \mathcal{T}$ do
  $\mathcal{W} \leftarrow \mathit{Union}(\mathcal{W}, \mathit{RelProd}(\mathcal{U}, \mathcal{N}_\alpha))$;
$\mathcal{U} \leftarrow \mathit{Diff}(\mathcal{W}, \mathcal{Y})$;

However, if we do not require strict breadth-first order, we can "chain" the Petri net transitions and use the following statements instead:

for each $\alpha \in \mathcal{T}$ do
  $\mathcal{U} \leftarrow \mathit{Union}(\mathcal{U}, \mathit{RelProd}(\mathcal{U}, \mathcal{N}_\alpha))$;
$\mathcal{U} \leftarrow \mathit{Diff}(\mathcal{U}, \mathcal{Y})$;

The effect of this change is that, if $\mathbf{i}$ is in $\mathcal{U}$, thus will be explored in the current iteration, and we use a particular transition order, say α, then β, then γ, the current iteration will discover all markings reachable from $\mathbf{i}$ by firing sequences (α), (α, β), (α, β, γ), (α, γ), (β), (β, γ), and (γ). This may find more markings in the current iteration than if we used a strictly breadth-first approach, which instead just considers the individual effect of transitions α, β, and γ.

It is easy to prove that the use of chaining can only reduce, not increase, the *number* of iterations. However, the *cost* of each iteration depends on the size of the involved decision diagrams encoding the various sets being manipulated (the encodings of the forward functions are fixed, thus independent of the iteration strategy). Fewer iterations on different decision diagrams do not necessarily imply higher efficiency in a symbolic setting. The same can be said

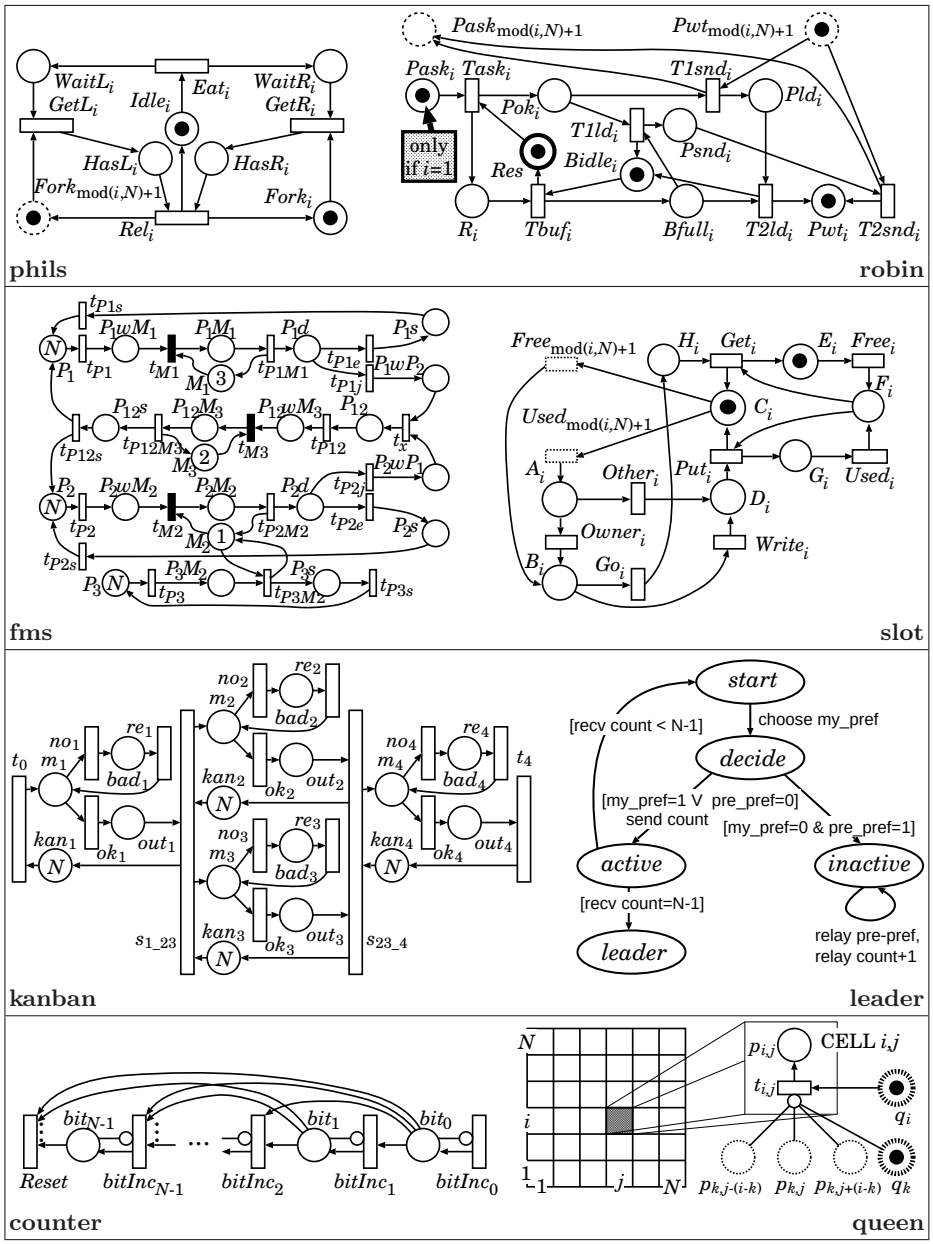

Fig. 7. The Petri net models used for experiments in our study.

for the difference between the two algorithms of Fig. 5, which end up encoding either the *frontier* markings at *exactly* a given distance d from $\mathcal{X}_{init}$, or *all* the markings *up to* distance d, respectively. Combining these two choices (BFS vs. chaining, frontier vs. all) gives us the four algorithms shown in Fig. 8, whose per-

```
BfSsGen(X_init, {N_α : α ∈ T})
 1 Y ← X_init;                    • known markings
 2 U ← X_init;                    • frontier markings
 3 repeat
 4    W ← ∅;
 5    for each α ∈ T do
 6       W ← Union(W, RelProd(U, N_α));
 7    U ← Diff(W, Y);
 8    Y ← Union(Y, U);
 9 until U = ∅;
10 return Y;
```

```
ChSsGen(X_init, {N_α : α ∈ T})
 1 Y ← X_init;                    • known markings
 2 U ← X_init;                    • frontier markings
 3 repeat
 4    for each α ∈ T do
 5       W ← Diff(RelProd(U, N_α), Y);
 6       U ← Union(U, W);
 7    U ← Diff(U, Y);
 8    Y ← Union(Y, U);
 9 until U = ∅;
10 return Y;
```

```
AllBfSsGen(X_init, {N_α : α ∈ T})
 1 Y ← X_init;                    • known markings
 2 repeat
 3    O ← Y;                      • save old markings
 4    W ← ∅;
 5    for each α ∈ T do
 6       W ← Union(W, RelProd(O, N_α));
 7    Y ← Union(O, W);
 8 until O = Y;
 9 return Y;
```

```
AllChSsGen(X_init, {N_α : α ∈ T})
 1 Y ← X_init;                    • known markings
 2 repeat
 3    O ← Y;                      • save old markings
 4    for each α ∈ T do
 5       Y ← Union(Y, RelProd(Y, N_α));
 6 until O = Y;
 7 return Y;
```

Fig. 8. Four simple variants of symbolic reachability-set generation [15].

formance spans several orders of magnitude, as shown by the results of Fig. 9, where *AllChSsGen* is a consistently good choice, both (peak) memory and time-wise. Still, the memory required to store $\mathcal{X}_{rch}$ alone is a negligible fraction of the peak requirement (except for the **queen** model, which is pathologically difficult for symbolic methods), leaving hope for improvement, as we see next.

Much lower peak memory requirements can usually be achieved by adopting a *saturation* strategy [13]. Before presenting this approach, we need to introduce the concept of *locality* and a new reduction rule to exploit locality when encoding forward functions of asynchronous systems.

Given $\alpha \in \mathcal{T}$, define the subsets of the state variables $\{v_L, ..., v_1\}$ that can be modified by α as $\mathcal{V}_M(\alpha) = \{v_k : \exists \mathbf{i}, \mathbf{i}' \in \mathcal{X}_{pot}, \mathbf{i}' \in \mathcal{N}_\alpha(\mathbf{i}) \wedge i_k \neq i'_k\}$, and that can disable α as $\mathcal{V}_D(\alpha) = \{v_k : \exists \mathbf{i}, \mathbf{j} \in \mathcal{X}_{pot}, \forall h \neq k, i_h = j_h \wedge \mathcal{N}_\alpha(\mathbf{i}) \neq \emptyset \wedge \mathcal{N}_\alpha(\mathbf{j}) = \emptyset\}$.

If $v_k \notin \mathcal{V}_M(\alpha) \cup \mathcal{V}_D(\alpha)$, we say that α and v_k are *independent*. Most events in *globally-asynchronous locally-synchronous* models are highly *local*; for example, in Petri nets, a transition is independent of any place not connected to it. We let $Top(\alpha) = \max(\mathcal{V}_M(\alpha) \cup \mathcal{V}_D(\alpha))$ and $Bot(\alpha) = \min(\mathcal{V}_M(\alpha) \cup \mathcal{V}_D(\alpha))$ be the highest and the lowest variables dependent on α, respectively, and observe that the *span* of variables $(Top(\alpha), ..., Bot(\alpha))$ is often much smaller than $(v_L, ..., v_1)$.

However, the canonical forms considered so far do not exploit this locality in a symbolic encoding of $\mathcal{N}_\alpha$. We need a more general and flexible canonical form that associates an individual *reduction rule* $R(v_k)$ with each variable v_k:

N	$\lvert\mathcal{X}_{rch}\rvert$	Time				Memory				$\lvert\mathcal{X}_{rch}\rvert$
		Bf	*AllBf*	*Ch*	*AllCh*	*Bf*	*AllBf*	*Ch*	*AllCh*	**mem.**
phils										
50	$2.22{\cdot}10^{31}$	4.26	3.52	0.18	0.08	87.81	73.88	9.65	4.98	0.02
100	$4.96{\cdot}10^{62}$	61.01	50.49	1.04	0.34	438.51	371.44	35.18	16.01	0.04
200	$2.46{\cdot}10^{125}$	1550.86	1177.16	5.62	2.31	2795.03	2274.97	138.85	61.30	0.09
robin										
50	$1.26{\cdot}10^{17}$	24.41	34.55	0.43	0.34	172.46	110.83	11.53	9.74	0.10
60	$1.55{\cdot}10^{20}$	51.17	72.41	0.74	0.69	294.42	178.61	17.24	15.70	0.14
70	$1.85{\cdot}10^{23}$	101.45	138.93	1.20	1.14	476.12	278.69	25.01	22.90	0.19
fms										
15	$7.24{\cdot}10^{8}$	9.49	4.12	1.35	0.80	91.11	46.63	23.56	17.68	0.05
20	$8.83{\cdot}10^{9}$	37.13	14.00	4.35	2.26	166.10	79.42	64.00	29.05	0.08
30	$3.43{\cdot}10^{11}$	501.26	94.14	16.41	10.02	497.63	175.90	142.99	77.78	0.19
slot										
20	$2.73{\cdot}10^{20}$	12.81	12.82	12.78	12.81	239.91	239.91	239.91	239.91	0.03
30	$1.03{\cdot}10^{31}$	93.23	92.33	93.18	93.07	1140.41	1140.41	1140.41	1140.41	0.07
40	$4.15{\cdot}10^{41}$	504.23	504.62	490.25	473.10	6272.78	6272.78	6272.78	6272.78	0.13
kanban										
20	$8.05{\cdot}10^{11}$	4.63	4.52	0.92	0.85	89.10	81.12	60.37	53.38	0.07
30	$4.98{\cdot}10^{13}$	28.78	27.45	5.96	5.72	539.73	399.76	160.52	160.45	0.21
40	$9.94{\cdot}10^{14}$	124.92	130.47	28.48	43.45	1530.40	1098.08	527.80	482.77	0.47
leader										
6	$1.89{\cdot}10^{6}$	7.90	15.29	7.86	6.79	73.25	75.66	81.93	57.64	0.32
7	$2.39{\cdot}10^{7}$	37.59	75.16	40.23	35.62	201.83	130.37	254.89	156.12	0.81
8	$3.04{\cdot}10^{8}$	175.88	558.51	200.84	171.32	516.79	201.75	692.89	259.06	1.82
counter										
10	$1.02{\cdot}10^{3}$	0.01	0.04	0.01	0.01	0.71	1.72	0.79	0.67	0.00
15	$3.27{\cdot}10^{4}$	2.29	5.84	1.50	2.38	22.73	5.17	12.14	6.80	0.00
20	$1.04{\cdot}10^{6}$	652.15	447.09	192.11	204.36	795.92	9.33	212.35	15.28	0.00
queen										
10	$3.55{\cdot}10^{4}$	0.68	0.68	0.68	0.68	10.39	10.48	10.40	10.39	1.13
12	$8.56{\cdot}10^{5}$	23.63	23.64	23.62	23.61	220.33	220.54	220.35	220.35	20.77
14	$2.73{\cdot}10^{7}$	1096.51	1097.97	1094.58	1093.75	7305.90	7306.30	7305.93	7305.94	505.73

Fig. 9. Results for four reachability-set generation variants (memory: MB, time: sec).

– If $R(v_k) = F$, variable v_k is *fully-reduced*. No node p associated with v_k can have all children $p[i_k]$, for $i_k \in \mathcal{X}_k$, coincide.

– If $R(v_k) = I$, variable k is *identity-reduced*. Let q associated with v_k be *singular* if it has exactly one edge $q[i_k] \neq \mathbf{0}$; then, no edge $p[i_l]$ can point to q if $i_l = i_k$ or if the edge skips a fully-reduced variable v_h such that $i_k \in \mathcal{X}_h$.

– If $R(v_k) = Q$, variable v_k is *quasi-reduced*. There must be at least one node associated with v_k and no edge can skip v_k.

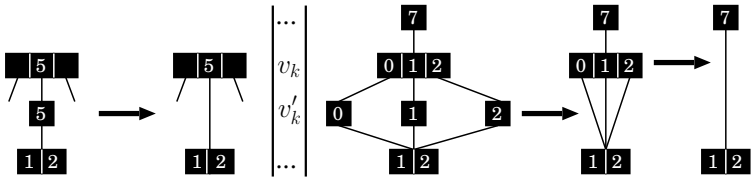

Fig. 10. Advantages of using the identity-reduced rule in the presence of locality.

The function $f_p : \mathcal{X}_{pot} \to \mathbb{B}$ encoded by a node p associated with v_k is then

$$f_p(i_L,...,i_1) = \begin{cases} p & \text{if } k=0 \\ f_{p[i_k]}(i_L,...,i_1) & \text{if } k>0 \text{ and } \forall v_k \succ v_h \succ p[i_k].var,\ R(v_h)=I \Rightarrow i_h=i_k \\ \mathbf{0} & \text{otherwise,} \end{cases}$$

thus, if an edge $p[i_k]$ skips over an identity-reduced variable v_h, the value of i_h must be the same as that of i_k.

This more general reduction is particularly useful when encoding the forward functions of a model with widespread locality. Specifically, when encoding $\mathcal{N}_\alpha$, we still use an interleaved order for the variables, but set $R(v'_k) = I$ for all *"to"*, or *"primed"*, variables v'_k, while *"from"*, or *"unprimed"* variables are either all fully-reduced or, alternatively, quasi-reduced if $v_k \in \mathcal{V}_M(\alpha) \cup \mathcal{V}_D(\alpha)$ and fully-reduced otherwise. Fig. 10 illustrates the effect of this choice. On the left, the node associated with v'_k is singular, thus it is eliminated (this corresponds to a Petri net where the firing of α leaves the corresponding place p_k unchanged if p_k contains five tokens). More importantly, the situation on the right considers the common case where α is independent of place p_k. The identity-reduced rule for v'_k means that *all* nodes associated with v'_k will be singular, thus are eliminated; but then, *all* nodes associated with v_k, which is fully-reduced, will become redundant, so are eliminated as well. The result is that all edges will simply skip *both* variables v_k and v'_k in the MDD encoding of $\mathcal{N}_\alpha$.

To fully exploit locality and efficiently build an encoding for the forward function, not only we partition $\mathcal{N}$ as the disjunction $\bigcup_{\alpha\in\mathcal{T}} \mathcal{N}_\alpha$, but we further encode each $\mathcal{N}_\alpha$ as a conjunction. In the best case, $\mathcal{N}_\alpha$ has a conjunctive representation by variable, $\mathcal{N}_\alpha = (\bigcap_{k\in\mathcal{V}_M(\alpha)\cup\mathcal{V}_D(\alpha)} \mathcal{N}_{k,\alpha}) \cap (\bigcap_{k\notin\mathcal{V}_M(\alpha)\cup\mathcal{V}_D(\alpha)} \mathcal{I}_k)$, where the *local* function $\mathcal{N}_{k,\alpha} : \mathcal{X}_k \to 2^{\mathcal{X}_k}$ describes how α depends on v_k and affects v'_k, independently of the value of all the other state variables, while $\mathcal{I}_k$ is the identity for v_k, that is, $\mathcal{I}_k(i_k) = \{i_k\}$ for any $i_k \in \mathcal{X}_k$. Along the same lines, a *disjunctive-then-conjunctive* decomposition of $\mathcal{N}$ can be defined for arbitrary models, $\mathcal{N}_\alpha = (\bigcap_{c=1}^{m_\alpha} \mathcal{N}_{c,\alpha}) \cap (\bigcap_{k\notin\mathcal{V}_M(\alpha)\cup\mathcal{V}_D(\alpha)} \mathcal{I}_k)$, where we now have some m_α conjuncts, each one depending on some set of variables $\mathcal{D}_{c,\alpha}$, so that $\bigcup_{c=1}^{m_\alpha} \mathcal{D}_{c,\alpha} = \mathcal{V}_M(\alpha) \cup \mathcal{V}_D(\alpha)$ and $\mathcal{N}_{c,\alpha} : \times_{k\in\mathcal{D}_{c,\alpha}} \mathcal{X}_k \to 2^{\times_{k\in\mathcal{D}_{c,\alpha}} \mathcal{X}_k}$ describes how α is affected and affects the variables in $\mathcal{D}_{c,\alpha}$ as a whole.

As an example, consider the portion of self-modifying Petri net in Fig. 11 (the pseudocode on the right, to be interpreted as an atomic check and concurrent

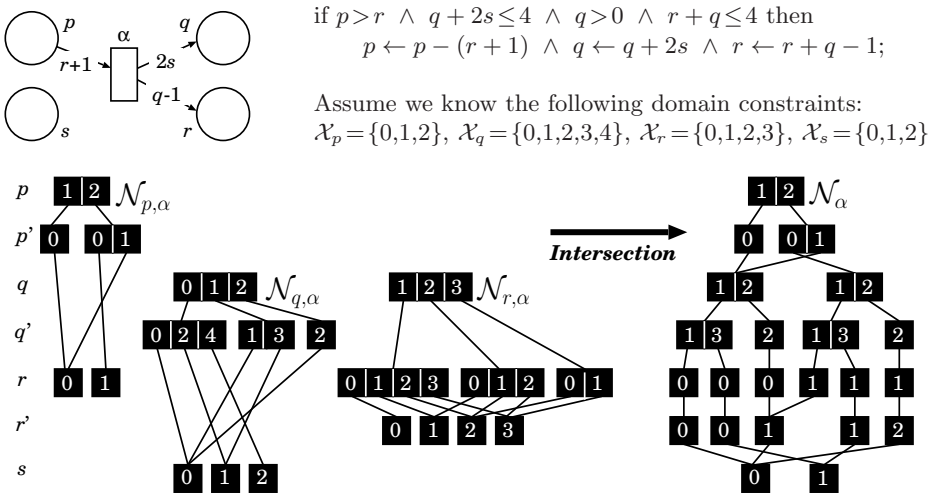

Fig. 11. A transition α in a self-modifying Petri net, its equivalent pseudocode, and the conjunctive decomposition of its forward function $\mathcal{N}_\alpha$.

assignment, illustrates the semantic of transition α). The bottom of the figure shows three conjuncts $\mathcal{N}_{p,\alpha}$, $\mathcal{N}_{q,\alpha}$, and $\mathcal{N}_{r,\alpha}$, whose intersection describes the overall enabling conditions and effect of transition α. For example, $\mathcal{N}_{p,\alpha}$ describes how and when p might be updated: p must be greater than r and its value is decreased by $r+1$ if α fires (for clarity, we use the quasi-reduced rule for variables on which a conjunct depends). By decomposing each $\mathcal{N}_\alpha$ in this fashion, we can then build the MDD for each conjunct $\mathcal{N}_{c,\alpha}$ explicitly (as shown in Fig. 11, these conjuncts tend to be small because of locality), and then use symbolic intersection operations to build the MDD encoding the overall $\mathcal{N}_\alpha$, which might instead encode a very large relation. Note that, for this to work efficiently, we need to "play a trick" with the reduction rules: for example, q, q', s, s' and any other variable not explicitly mentioned must be interpreted as fully-reduced in $\mathcal{N}_{p,\alpha}$ prior to performing the intersection with $\mathcal{N}_{q,\alpha}$; however, after performing the intersection, s' must be interpreted as identity-reduced (since s is not affected by α), and any pair of unprimed and primed variables not explicitly mentioned must be interpreted as fully-reduced and identity-reduced, respectively.

We can now describe the *saturation* strategy. We say that a node associated with v_k is *saturated* if it is a fixpoint with respect to all events α such that $Top(\alpha) \leq k$. This implies that all MDD nodes reachable from p are also saturated. The idea is then to saturate the nodes bottom-up, and to saturate any new node prior to inserting it in the unique table:

- Build the L-variable MDD encoding of $\mathcal{X}_{init}$.
- Saturate nodes associated with v_1: fire in them all events α s.t. $Top(\alpha) = 1$.
- Saturate nodes associated with v_2: fire in them all events α s.t. $Top(\alpha) = 2$. If this creates nodes associated with v_1, saturate them immediately.
- ...

```
mdd Saturate(variable v_k, mdd p) is                    • quasi-reduced version
 1 if v_k = v_0 then return p;
 2 if Cache contains entry ⟨Saturate, p : r⟩ then return r;
 3 foreach i_k ∈ X_k do
 4    r_{i_k} ← Saturate(v_{k−1}, p[i_k]);      • first, be sure that the children are saturated
 5 repeat
 6    choose α ∈ T, i_k, j_k ∈ X_k s.t. Top(α) = k and r_{i_k} ≠ 0 and N_α[i_k][j_k] ≠ 0;
 7    r_{j_k} ← Union(r_{j_k}, RelProdSat(v_{k−1}, r_{i_k}, N_α[i_k][j_k]));
 8 until r_0, ..., r_{n_k−1} do not change;
 9 r ← UniqueTableInsert(v_k, r_0, ..., r_{n_k−1});
10 enter ⟨Saturate, p : r⟩ in Cache;
11 return r;
```

```
mdd RelProdSat(variable v_k, mdd q, mdd2 f) is
 1 if v_k = v_0 then return q ∧ f;
 2 if Cache contains entry ⟨RelProdSat, q, f : r⟩ then return r;
 3 foreach i_k, j_k ∈ X_k s.t. q[i_k] ≠ 0 and f[i_k][j_k] ≠ 0 do
 4    r_{j_k} ← Union(r_{j_k}, RelProdSat(v_{k−1}, q[i_k], f[i_k][j_k]));
 5 r ← Saturate(v_k, UniqueTableInsert(v_k, r_0, ..., r_{n_k−1}));
 6 enter ⟨RelProdSat, q, f : r⟩ in Cache;
 7 return r.
```

Fig. 12. The saturation algorithm to build the reachability set [11].

– Saturate the root associated with v_L: fire in it all events α s.t. $Top(\alpha) = L$. If this creates nodes associated with v_k, $L > k \geq 1$, saturate them immediately.

Fig. 12 shows the pseudocode to implement these steps. With saturation, the traditional idea of a global fixpoint iteration for the overall MDD disappears, and, while markings are not discovered in breadth-first order, the tradeoff is sometimes enormous memory and runtime savings when applied to asynchronous systems. Saturation is not guaranteed to be optimal, but has many advantages: (1) firing α in p benefits from having saturated the nodes below p, thus finds the maximum number of markings under p, (2) once node p associated with v_k is saturated, we never fire an event α with $Top(\alpha) \leq k$ on p or any of the nodes reachable from p, (3) except for the nodes of the MDD describing $\mathcal{X}_{init}$, only saturated nodes are placed in the unique table and the operation cache, and (4) many of these nodes will still be present in the final MDD which, by definition, can only contain saturated nodes.

We observe that, while the notion of a transition is clearly defined for a Petri net level, it is somewhat arbitrary for the corresponding MDDs. Given two transitions α and β with $Top(\alpha) = Top(\beta)$, we could choose to encode $\mathcal{N}_{\alpha,\beta} = \mathcal{N}_\alpha \cup \mathcal{N}_\beta$ with a single MDD instead of two individual MDDs, without affecting (much) the order of node operations performed by saturation. We then consider two extreme cases, *saturation by event*, where each Petri net transition is encoded with a separate MDD, and *saturation by variable*, where we use L MDDs, the k^{th} one encoding $\mathcal{N}_k = \bigcup_{\alpha: Top(\alpha)=v_k} \mathcal{N}_\alpha$ (some $\mathcal{N}_k$ could be empty). The latter requires fewer unions during saturation, but its efficiency depends on

N	$\lvert\mathcal{X}_{rch}\rvert$	$\mathcal{X}_{rch}$ **mem.**	$\mathcal{N}$ **mem.**		**Peak mem.**		**Time**	
			Event	Variable	Event	Variable	Event	Variable
phils								
200	$2.46 \cdot 10^{125}$	0.09	0.64	0.43	2.05	2.59	0.19	0.20
500	$3.03 \cdot 10^{313}$	0.22	1.60	1.07	4.06	4.43	0.52	0.53
1000	$9.18 \cdot 10^{626}$	0.43	3.20	2.14	6.80	8.34	1.21	1.26
robin								
100	$2.85 \cdot 10^{32}$	0.38	0.13	0.09	7.01	7.00	0.52	0.51
200	$7.23 \cdot 10^{62}$	1.44	0.26	0.18	47.64	47.47	5.19	5.11
500	$3.68 \cdot 10^{153}$	8.75	0.65	0.45	691.14	689.13	102.66	97.37
fms								
30	$3.43 \cdot 10^{11}$	0.19	0.21	0.20	5.49	7.35	11.31	11.56
40	$4.96 \cdot 10^{12}$	0.37	0.35	0.32	7.81	15.61	34.47	35.71
50	$4.08 \cdot 10^{13}$	0.63	0.51	0.48	12.04	26.18	85.30	86.86
slot								
30	$1.03 \cdot 10^{31}$	0.07	0.06	0.04	2.53	1.85	0.17	0.09
50	$1.72 \cdot 10^{52}$	0.20	0.10	0.07	4.22	4.17	0.78	0.35
100	$2.60 \cdot 10^{105}$	0.77	0.19	0.14	6.42	5.75	9.58	2.32
kanban								
10	$1.00 \cdot 10^{9}$	0.01	0.18	0.16	5.19	5.26	0.14	0.14
20	$8.05 \cdot 10^{11}$	0.07	1.22	1.08	44.65	46.13	4.36	4.39
30	$4.98 \cdot 10^{13}$	0.21	3.88	3.42	164.77	160.17	44.81	46.69
leader								
8	$3.04 \cdot 10^{8}$	1.82	0.18	0.09	9.59	9.28	20.26	19.33
9	$3.90 \cdot 10^{9}$	3.75	0.23	0.11	16.36	16.27	66.84	68.43
10	$5.02 \cdot 10^{10}$	7.37	0.29	0.13	29.89	30.48	241.70	231.12
counter								
100	$1.26 \cdot 10^{30}$	0.01	0.51	0.50	0.12	0.12	0.02	0.02
200	$1.60 \cdot 10^{60}$	0.01	2.01	1.99	0.21	0.21	0.11	0.11
300	$2.03 \cdot 10^{90}$	0.02	4.51	4.48	0.29	0.29	0.28	0.31
queen								
12	$8.56 \cdot 10^{5}$	20.77	0.12	0.11	46.69	46.71	2.96	2.69
13	$4.67 \cdot 10^{6}$	99.46	0.16	0.14	218.54	218.57	16.29	14.91
14	$2.73 \cdot 10^{7}$	505.73	0.21	0.19	1112.93	1112.96	95.69	88.12

Fig. 13. Results for saturation by event vs. by variable (memory: MB, time: sec).

the size of the MDD encoding $\mathcal{N}_k$, as compared to that of the various $\mathcal{N}_\alpha$ used to build it. Experimentally, we have found that either approach can work best, depending on the particular model, as shown in Fig. 13. Either one, however, when compared with the results of Fig. 9, demonstrates the effectiveness of the saturation algorithm for reachability-set generation: saturation can scale to much larger values of N because of greatly reduced peak memory and runtime.

3 CTL model checking of Petri nets

The generation of the reachability set for a Petri net is usually just a first step. Often, we want to analyze the behavior of the net, for example verify that certain

$\sigma \models p \quad \Leftrightarrow \sigma = (\mathbf{s}, \cdots)$ and $\mathbf{s} \models p$
$\sigma \models \neg q \quad \Leftrightarrow \sigma \not\models q$
$\sigma \models q \vee q' \Leftrightarrow \sigma \models q$ or $\sigma \models q'$
$\sigma \models q \wedge q' \Leftrightarrow \sigma \models q$ and $\sigma \models q'$
$\sigma \models \mathsf{X}q \quad \Leftrightarrow \sigma_{[1]} \models q$
$\sigma \models \mathsf{F}q \quad \Leftrightarrow \exists n \in \mathbb{N}, \sigma_{[n]} \models q$
$\sigma \models \mathsf{G}q \quad \Leftrightarrow \forall n \in \mathbb{N}, \sigma_{[n]} \models q$
$\sigma \models q\mathsf{U}q' \Leftrightarrow \exists n \in \mathbb{N}, \sigma_{[n]} \models q'$ and $\forall m < n, \sigma_{[m]} \models q$
$\sigma \models q\mathsf{R}q' \Leftrightarrow \forall m \geq 0,$ if $\forall n < m, \sigma_{[n]} \not\models q$ then $\sigma_{[m]} \models q'$

$\mathbf{s} \models a \quad \Leftrightarrow a \in \mathcal{L}(\mathbf{s})$
$\mathbf{s} \models \neg p \quad \Leftrightarrow \mathbf{s} \not\models p$
$\mathbf{s} \models p \vee p' \Leftrightarrow \mathbf{s} \models p$ or $\mathbf{s} \models p'$
$\mathbf{s} \models p \wedge p' \Leftrightarrow \mathbf{s} \models p$ and $\mathbf{s} \models p'$
$\mathbf{s} \models \mathsf{E}q \quad \Leftrightarrow \exists \sigma = (\mathbf{s}, \cdots), \sigma \models q$
$\mathbf{s} \models \mathsf{A}q \quad \Leftrightarrow \forall \sigma = (\mathbf{s}, \cdots), \sigma \models q$

Fig. 14. CTL semantics [23].

properties are satisfied. In this section, we consider the problem of CTL model checking for Petri nets, including the generation of witnesses or counterexamples and the use of bounded approaches, both of which use more general decision diagrams that can encode integer, not just boolean, functions.

3.1 CTL model checking

Temporal logics usually refer to a Kripke structure $(\mathcal{X}_{pot}, \mathcal{X}_{init}, \mathcal{N}, \mathcal{A}, \mathcal{L})$. For Petri nets, we can think of the "$\mathcal{X}_{pot}, \mathcal{X}_{init}, \mathcal{N}$" portion as the potential state space $\mathbb{N}^{|\mathcal{P}|}$, the initial marking(s), and the forward function, while $\mathcal{A}$ is a finite set of *atomic propositions* of interest, such as "place a contains three tokens" or "transition α is enabled", and $\mathcal{L} : \mathcal{X}_{pot} \to 2^{\mathcal{A}}$ is a *labeling function* that specifies which atomic propositions hold in each marking.

We consider the temporal logic CTL (*computation tree logic*) [23], which talks about *state formulas* and *path formulas* using the following syntax:

- If $p \in \mathcal{A}$, p is a state formula.
- If p and p' are state formulas, $\neg p$, $p \vee p'$, and $p \wedge p'$ are state formulas.
- If q is a path formula, $\mathsf{E}q$ and $\mathsf{A}q$ are state formulas.
- If p and p' are state formulas, $\mathsf{X}p$, $\mathsf{F}p$, $\mathsf{G}p$, $p\mathsf{U}p'$, and $p\mathsf{R}p'$ are path formulas.

Fig. 14 illustrates the semantics of a CTL formula, where $\mathbf{s} \models p$ means that state formula p holds in marking $\mathbf{s}$, $\sigma \models q$ means that path formula q holds on path σ, and, given an infinite execution path $\sigma = (\mathbf{s}_0, \mathbf{s}_1, \mathbf{s}_2, \cdots)$ and an integer $n \in \mathbb{N}$, we let $\sigma_{[n]}$ denote the infinite path $(\mathbf{s}_n, \mathbf{s}_{n+1}, \cdots)$.

As we are ultimately interested in whether a certain marking (usually the initial marking) satisfies a given state formula, CTL model-checking can be performed by manipulating just sets of states, not paths, by considering all CTL operators as being formed by the combination of a *path quantifier*, E or A, followed by a *temporal operator*, X, F, G, U, or R. Then, we only need to be able to build the set of markings corresponding to atomic propositions or to the formulas $\mathsf{EX}p$, $\mathsf{E}p\mathsf{U}q$, and $\mathsf{EG}p$, where p and q are themselves atomic propositions or EX, EU, or EG formulas, because of the following equivalences:

$\mathsf{AX}p = \neg\mathsf{EX}\neg p$, $\mathsf{EF}p = \mathsf{E}true\mathsf{U}p$, $\mathsf{AG}p = \neg\mathsf{EF}\neg p$, $\mathsf{E}p\mathsf{R}q = \neg\mathsf{A}\neg p\mathsf{U}\neg q$,
$\mathsf{AF}p = \neg\mathsf{EG}\neg p$, $\mathsf{A}p\mathsf{U}q = \neg(\mathsf{E}\neg q\mathsf{U}(\neg p \wedge \neg q)) \wedge \neg\mathsf{EG}\neg q$, $\mathsf{A}p\mathsf{R}q = \neg\mathsf{E}\neg p\mathsf{U}\neg q$.

3.2 Symbolic CTL model checking algorithms

To compute the set of markings satisfying a CTL formula such as $\mathsf{E}p\mathsf{U}q$, we first compute the sets of markings $\mathcal{X}_p$ and $\mathcal{X}_q$ satisfying the inner formulas p and q, respectively. Fig. 15 shows the explicit algorithms for the EX, EU, and EG operators, on the left, while the their respective symbolic versions are on the right. The symbolic algorithms use $\mathcal{N}^{-1}$, the *backward function*, satisfying $\mathbf{i} \in \mathcal{N}^{-1}(\mathbf{j}) \Leftrightarrow \mathbf{j} \in \mathcal{N}(\mathbf{i})$, to "go backwards". By doing so, they might go through unreachable markings, but this does not lead to incorrect results because, while $\mathcal{N}$ might contain transitions from unreachable to reachable markings, it cannot by definition contain transitions from reachable to unreachable markings. It is remarkable that, while the explicit EG algorithm first discovers all strongly-connected components of $\mathcal{N} \cap \mathcal{X}_p \times \mathcal{X}_p$ (or of its reachable portion), the symbolic algorithm starts with a larger set of markings, (all those in $\mathcal{X}_p$, and *reduces* it, without having to compute the strongly connected components of $\mathcal{N}$.

The symbolic EU and EG algorithms of Fig. 15 proceed in strict breadth-first order. Thus, just as for reachability-set generation, it is natural to attempt to improve them with a saturation-based approach. For EU, we start from $\mathcal{X}_q$ and walk backwards using $\mathcal{N}^{-1}$, while remaining in $\mathcal{X}_p$ at all times. A breadth-first exploration can simply perform an intersection with $\mathcal{X}_p$ at each step, but it is too costly to do this at each lightweight saturation step. We then have two options.

The simplest one is to use *constrained functions* $\mathcal{N}^{-1}_{\alpha,\mathcal{X}_p} = \mathcal{N}^{-1}_\alpha \cap (\mathcal{X}_{pot} \times \mathcal{X}_p)$, i.e., symbolically remove from $\mathcal{N}^{-1}_\alpha$ any backward steps to markings not in $\mathcal{X}_p$. After computing these functions, we can run ordinary saturation, analogous to that for reachability-set generation except that we start from $\mathcal{X}_q$ instead of $\mathcal{X}_{init}$ and we apply $\{\mathcal{N}^{-1}_{\alpha,\mathcal{X}_p} : \alpha \in \mathcal{T}\}$ instead of $\{\mathcal{N}_\alpha : \alpha \in \mathcal{T}\}$. The downside of this approach is a possible decrease in locality, since, if $\mathcal{X}_p$ depends on v_k, i.e., if its fully-reduced encoding has nodes associated with v_k, then $Top(\mathcal{N}^{-1}_{\alpha,\mathcal{X}_p}) \trianglerighteq v_k$. In particular, if $\mathcal{X}_p$ depends on v_L, this approach degrades to chaining.

A second approach is instead *constrained saturation*, which uses the original (unconstrained) backward functions $\mathcal{N}^{-1}_\alpha$, but has three, not just two, parameters in its recursive calls: s (a node in the MDD encoding the set of markings being saturated), r (a node in the MDD encoding a backward function), and c (a node in the MDD encoding the constraint $\mathcal{X}_p$), as shown by the pseudocode in Fig. 16. In other words, instead of modifying the backward functions, we constrain the marking exploration *on-the-fly* during the "check-and-fire" steps in saturation, thus we do not reduce locality.

Fig. 17 reports peak memory and runtime experimental results for EU computation. Columns *BFS*, $ConNSF_{rch}$, and $ConSat_{rch}$ correspond to the traditional symbolic breadth-first algorithm of Fig. 15, the ordinary saturation algorithm employing constrained functions, and the constrained saturation approach of Fig. 16, respectively. In general, saturation does better than BFS, sometimes by a huge factor. Then, among the two saturation approaches, constrained saturation tends to do much better than explicitly constraining the functions and running ordinary saturation. In particular, an advantage of constrained satura-

ExplicitEX$(\mathcal{X}_p, \mathcal{N})$ is

```
 1 Y ← ∅;
 2 while X_p ≠ ∅ do
 3     remove a marking i from X_p;
 4     for each i ∈ N^{-1}(i) do
 5         Y ← Y ∪ {i};
 6 return Y;
```

SymbolicEX$(\mathcal{X}_p, \mathcal{N})$ is

```
 1 Y ← RelProd(X_p, N^{-1});
 2 return Y;
```

ExplicitEU$(\mathcal{X}_p, \mathcal{X}_q, \mathcal{N})$ is

```
 1 Y ← ∅;
 2 for each i ∈ X_q do
 3     Y ← Y ∪ {i};
 4 while X_q ≠ ∅ do
 5     remove a marking i from X_q;
 6     for each i ∈ N^{-1}(i) do
 7         if i ∉ Y and i ∈ X_p then
 8             Y ← Y ∪ {i};
 9             X_q ← X_q ∪ {i};
10 return Y;
```

SymbolicEU$(\mathcal{X}_p, \mathcal{X}_q, \mathcal{N})$ is

```
 1 Y ← X_q;
 2 repeat
 3     O ← Y;
 4     W ← RelProd(Y, N^{-1});
 5     Z ← Intersection(W, X_p);
 6     Y ← Union(Z, Y);
 7 until O = Y;
 8 return Y;
```

ExplicitEG$(\mathcal{X}_p, \mathcal{N})$ is

```
 1 build the set of markings W in the SCCs
   in the subgraph of N induced by X_p;
 2 Y ← ∅;
 3 for each i ∈ W do
 4     Y ← Y ∪ {i};
 5 while W ≠ ∅ do
 6     remove a marking i from W;
 7     for each i ∈ N^{-1}(i) do
 8         if i ∉ Y and i ∈ X_p then
 9             Y ← Y ∪ {i};
10             W ← W ∪ {i};
11 return Y;
```

SymbolicEG$(\mathcal{X}_p, \mathcal{N})$ is

```
 1 Y ← X_p;
 2 repeat
 3     O ← Y;
 4     W ← RelProd(Y, N^{-1});
 5     Y ← Intersection(Y, W);
 6 until O = Y;
 7 return Y;
```

Fig. 15. Explicit vs. symbolic CTL model-checking algorithms [11].

tion is that its lightweight "check-and-fire" approach is not as sensitive to the complexity of the constraint.

Improving the computation of EG using saturation is instead more difficult, since EGp is a *greatest fixpoint*, which means that we initialize a larger set of markings $\mathcal{Y}$ to $\mathcal{X}_p$, then remove from it any marking $\mathbf{i}$ that cannot reach the current set $\mathcal{Y}$ through *any* of the transitions in $\mathcal{T}$. In other words, we cannot eliminate $\mathbf{i}$ from $\mathcal{Y}$ just because $\mathcal{N}_\alpha(\mathbf{i}) \cap \mathcal{Y} = \emptyset$ for a particular α. We then take a completely different approach and build a $2L$-variable MDD encoding the *backward reachability relation*, using constrained saturation, that is, given a set $\mathcal{Y}$ and a marking $\mathbf{i} \in \mathcal{Y}$, we define $\mathbf{j} \in (\mathcal{N}^{-1}_{\mathcal{Y},\mathcal{X}_p})^+(\mathbf{i})$ iff there exists a *nontrivial* forward path of markings in $\mathcal{X}_p$ from $\mathbf{j}$ to $\mathbf{i}$, where $\mathcal{X}_p$ is the constraint.

To compute the set of markings satisfying EGp, we set $\mathcal{Y}$ to $\mathcal{X}_p$ and observe that, if $\mathbf{j} \in (\mathcal{N}^{-1}_{\mathcal{X}_p,\mathcal{X}_p})^+(\mathbf{i})$, there must exist a marking $\mathbf{i}' \in \mathcal{N}^{-1}(\mathbf{i}) \cap \mathcal{X}_p$ such that

```
mdd ConsSaturate(mdd c, mdd s)
 1 if InCache(ConsSaturate, c, s, t) then return t;
 2 v_k ← s.var;
 3 t ← NewNode(v_k);
 4 r ← N_k^{-1};
 5 foreach i ∈ X_k s.t. s[i] ≠ 0 do
 6    if c[i] ≠ 0 then t[i] ← ConsSaturate(c[i], s[i]); else t[i] ← s[i];
 7 repeat
 8    foreach i, i' ∈ X_k s.t. r[i][i'] ≠ 0 do
 9       if c[i'] ≠ 0 then
10          u ← ConsRelProd(c[i'], t[i], r[i][i']);
11          t[i'] ← Or(t[i'], u);
12 until t does not change;
13 t ← UniqueTablePut(t);
14 CacheAdd(ConsSaturate, c, s, t);
15 return t;
```

```
mdd ConsRelProd(mdd c, mdd s, mdd r)
 1 if s = 1 and r = 1 then return 1;
 2 if InCache(ConsRelProd, c, s, r, t) then return t;
 3 v_l ← s.var;
 4 t ← 0;
 5    foreach i, i' ∈ X_l s.t. r[i][i'] ≠ 0 do
 6       if c[i'] ≠ 0 then
 7          u ← ConsRelProd(c[i'], s[i], r[i][i']);
 8          if u ≠ 0 then
 9             if t = 0 then t ← NewNode(v_l);
10             t[i'] ← Or(t[i'], u);
11 t ← ConsSaturate(c, UniqueTablePut(t));
12 CacheAdd(ConsRelProd, c, s, r, t);
13 return t;
```

Fig. 16. The pseudocode for constrained saturation [57].

$\mathbf{j} \in \mathit{ConsSaturate}(\mathcal{X}_p, \{\mathbf{i}'\})$. To build $(\mathcal{N}^{-1}_{\mathcal{X}_p,\mathcal{X}_p})^+$, we apply constrained saturation to the primed variables of the MDD encoding $\mathcal{N}^{-1} \cap (\mathcal{X}_p \times \mathcal{X}_p)$.

Then, EGp holds in marking $\mathbf{j}$ iff there exists a marking $\mathbf{i} \in \mathcal{X}_p$ that can reach itself through a nontrivial path in $\mathcal{X}_p$, $\mathbf{i} \in (\mathcal{N}^{-1}_{\mathcal{X}_p,\mathcal{X}_p})^+(\mathbf{i})$, and $\mathbf{j}$ can reach $\mathbf{i}$ through a path in $\mathcal{X}_p$, $\mathbf{j} \in (\mathcal{N}^{-1}_{\mathcal{X}_p,\mathcal{X}_p})^+(\mathbf{i})$. To obtain the MDD encoding EGp, we build the $2L$-variable MDD for $(\mathcal{N}^{-1}_{\mathcal{X}_p,\mathcal{X}_p})^+$, then the L-variable MDD for all markings in nontrivial strongly connected components of the restriction of the transition relation to $\mathcal{X}_p$, $\mathcal{X}_{scc} = \{\mathbf{i} : \mathbf{i} \in (\mathcal{N}^{-1}_{\mathcal{X}_p,\mathcal{X}_p})^+(\mathbf{i})\}$, and finally the L-variable MDD for EGp, by computing $\mathit{RelProd}(\mathcal{X}_{scc}, (\mathcal{N}^{-1}_{\mathcal{X}_p,\mathcal{X}_p})^+)$.

Building the reachability relation $(\mathcal{N}^{-1}_{\mathcal{X}_p,\mathcal{X}_p})^+$ is the most expensive step in this approach, but the use of constrained saturation instead of breadth-first iterations greatly improves efficiency. Furthermore, $(\mathcal{N}^{-1}_{\mathcal{X}_p,\mathcal{X}_p})^+$ contains information useful beyond the computation of EG. For example, in EG computation under a fairness

N	BFS			$ConsNSF_{rch}$		$ConsSat_{rch}$	
	iter.	**time**	**mem.**	**time**	**mem.**	**time**	**mem.**
phils : $\mathsf{E}(\neg HasL_1 \vee \neg HasR_1)\mathsf{U}(HasL_0 \wedge HasR_0)$							
100	200	7.63	62.06	0.27	10.87	0.01	2.04
500	1000	1843.32	550.01	10.58	186.07	0.09	7.35
1000	–	–	–	73.41	720.98	0.20	13.83
robin : $\mathsf{E}(Pld_1 = 0)\mathsf{U}(Psnd_0 = 1)$							
20	82	0.37	10.52	0.02	1.56	0.00	0.34
100	402	161.46	442.90	1.52	62.29	0.10	8.70
200	–	–	–	796.55	425.72	0.11	51.04
fms : $\mathsf{E}(P_1M_1 > 0)\mathsf{U}(P_1s = P_2s = P_3s = N)$							
10	103	7.11	16.40	0.44	5.35	0.03	2.32
25	–	–	–	410.34	20.73	0.93	8.24
50	–	–	–	–	–	53.29	40.17
slot : $\mathsf{E}(C_0 = 0)\mathsf{U}(C_0 = 1)$							
10	16	0.06	1.94	0.01	1.54	0.00	0.34
20	31	1.46	27.05	0.17	9.39	0.01	1.04
30	46	9.67	129.56	1.50	29.09	0.03	2.20
kanban : $\mathsf{E}(true)\mathsf{U}(out_4 = 0)$							
20	21	0.51	46.13	1.49	46.14	0.04	46.13
30	31	3.75	160.46	16.00	160.47	0.23	160.17
40	41	19.29	378.91	62.67	378.98	0.93	378.91
leader : $\mathsf{E}(pref_1 = 0)\mathsf{U}(status_0 = leader)$							
6	101	12.51	129.39	2.21	16.10	0.78	5.74
7	123	55.63	461.53	23.17	45.78	5.90	8.03
8	145	214.07	1191.79	192.06	119.81	28.23	12.51
counter : $\mathsf{E}(bit_{N-1} = 0)\mathsf{U}(bit_{N-1} = 1)$							
10	513	0.02	1.31	0.00	0.29	0.00	0.04
15	16385	3.01	6.42	0.36	3.42	0.00	0.05
20	524289	394.35	61.62	28.33	7.14	0.00	0.05
queen : $\mathsf{E}(q_0 = 1)\mathsf{U}(q_0 = 0)$							
12	13	11.92	164.95	2.51	185.77	4.40	76.87
13	14	80.83	889.39	18.58	1070.50	44.16	334.36
14	15	531.42	4566.38	–	–	865.23	1677.45

Fig. 17. Results for CTL EU queries (memory: MB, time: sec).

constraint $\mathcal{F}$, we seek an infinite execution where some markings in $\mathcal{F}$ appear infinitely often. Marking $\mathbf{j}$ satisfies $\mathsf{EG}p$ under fairness constraint $\mathcal{F}$ iff there is a marking $\mathbf{i} \in (\mathcal{N}^{-1}_{\mathcal{F}\cap\mathcal{X}_p,\mathcal{X}_p})^+(\mathbf{i})$ such that $\mathbf{j} \in (\mathcal{N}^{-1}_{\mathcal{F}\cap\mathcal{X}_p,\mathcal{X}_p})^+(\mathbf{i})$. In other words, we simply build the backward reachability relation starting from $\mathcal{F} \cap \mathcal{X}_p$, instead of all of $\mathcal{X}_p$. Experimentally, we found that, while traditional approaches suffer when adding a fairness constraint, our approach may instead become faster, as it performs the same computation, but starting from a reduced set of markings.

A question arising in symbolic CTL model checking is whether to constrain the paths to the reachable markings $\mathcal{X}_{rch}$ during the iterations, since we are usually only interested in reachable markings in the end. This is an issue because

backward search can reach unreachable markings from reachable ones but not vice versa, by definition (forward search from a set of reachable markings only finds reachable markings, instead). In other words, there are two reasonable ways to define $\mathcal{X}_p$: as $\mathcal{X}_{p,pot}$ (include in $\mathcal{X}_p$ all potential markings satisfying property p, even if they are not reachable) and as $\mathcal{X}_{p,rch}$ (include in $\mathcal{X}_p$ only the reachable markings satisfying property p), so that $\mathcal{X}_{p,rch} = \mathcal{X}_{p,pot} \cap \mathcal{X}_{rch}$. The latter of course tends to result in (much) smaller sets of markings being manipulated, but it does not follow that the MDD encoding these sets are correspondingly smaller. Indeed, exploration restricted to $\mathcal{X}_{p,pot}$ usually depends on fewer variables, thus may well lead to better saturation behavior.

Fig. 18 reports experimental results for EG computation. Columns *BFS* and *ReachRel* correspond to the traditional symbolic breadth-first algorithm of Fig. 15 and to the approach that builds the reachability relation using constrained saturation, respectively. Obviously, the high cost building *ReachRel* is often overwhelming. However, it should be observed that most of the models we consider are best-case-scenarios for *BFS*, since convergence is achieved immediately (**robin**, **fms**, and **kanban**) or in relatively few iterations; for **counter**, instead, BFS convergence requires $2^N - 1$ iterations and the *ReachRel* saturation approach is obviously enormously superior in runtime.

3.3 Integer-valued decision diagrams

In the following section, we will need to manipulate integer-valued, instead of boolean-valued, functions. Specifically, we will need to encode partial functions from $\mathcal{X}_{pot}$ to $\mathbb{N}$, or, which is the same since it is natural for our purposes to think of "undefined" as being "infinite", total functions from $\mathcal{X}_{pot}$ to $\mathbb{N} \cup \{\infty\}$.

We now introduce two variants of decision diagrams that can do just that. The first one, *(ordered) multi-terminal MDDs* is a fairly obvious extension of ordinary MDDs, where we allow each terminal to correspond to a distinct element of the range of the function to be encoded:

- The *terminal* nodes are distinct elements of a *range* set $\mathcal{X}_0$ and are associated with the range variable v_0, satisfying $v_k \succ v_0$ for any domain variable v_k.

The semantic of this multiway generalization of the MTBDDs introduced by Clarke et al. [22] is still very similar to that of BDDs and MDDs: to evaluate the encoded function on $(i_L, ..., i_1)$, we follow the corresponding path from the root and the reached terminal node, now an element of $\mathcal{X}_0$ instead of just **0** or **1**, is the desired value. Quasi-reduced and fully-reduced (and even identity-reduced) canonical forms can be defined exactly as for BDDs and MDDs.

The second variant *additive edge-valued MDDs*, or EV$^+$MDDs [19], is instead a more substantial departure from the decision diagrams seen so far, as there is a single terminal node carrying no information:

- The only *terminal* node is Ω, associated with variable $\Omega.var = v_0$, satisfying $v_k \succ v_0$ for any domain variable v_k.

N	*BFS*			*ReachRel*	
	iter.	**time**	**mem.**	**time**	**mem.**
phils : EG$\neg(HasL_0 \wedge HasR_0)$					
300	4	0.02	2.78	0.81	8.16
500	4	0.03	4.06	1.60	11.44
1000	4	0.08	7.63	16.91	20.89
robin : EG$(true)$					
20	1	0.00	0.48	0.15	4.16
50	1	0.03	2.58	2.48	36.10
100	1	0.63	14.60	18.96	179.76
fms : EG$(P_1 wP_2 > 0 \wedge P_2 = P_3 = N)$					
8	1	0.00	1.53	27.29	17.92
10	1	0.01	2.18	215.12	31.07
12	1	0.02	2.32	2824.78	59.46
slot : EG$(C_0 = 0)$					
20	781	2.17	6.22	12.23	14.24
25	1227	7.30	11.31	252.18	20.29
30	1771	20.30	20.36	–	–
kanban : EG$(back_2 = N-1 \vee back_3 = N-1)$					
10	1	0.00	5.26	3.08	5.29
15	1	0.00	17.43	31.59	17.49
20	1	0.01	46.13	189.99	46.40
leader : EG$(status_0 \neq leader)$					
3	14	0.00	0.64	0.74	7.38
4	18	0.01	1.96	81.29	20.83
5	22	0.05	3.67	–	–
counter : EG$(bit_{N-1} = 0)$					
10	512	0.00	0.24	0.00	0.05
15	16384	0.23	2.46	0.00	0.07
20	524288	17.35	4.74	0.00	0.08
queen : EG$(q_0 = 1)$					
10	10	0.65	16.88	0.58	12.62
11	11	4.34	46.06	4.15	53.42
12	12	37.00	189.22	23.26	244.65

Fig. 18. Results for CTL EG queries (memory: MB, time: sec).

while each edge has not only a label but also a value:

– For each $i_k \in \mathcal{X}_k$, a nonterminal node p associated with v_k has an outgoing edge labeled with i_k, pointing to $p[i_k].child$ and having a value $p[i_k].val \in \mathbb{N} \cup \{\infty\}$. We write $p[i_k] = \langle p[i_k].val, p[i_k].child \rangle$ and call such pair an *edge*.

The function $f_p : \mathcal{X}_{pot} \to \mathbb{N} \cup \{\infty\}$ encoded by a node p associated with v_k is

$$f_p(x_L, ..., x_1) = \begin{cases} 0 & \text{if } k = 0, \text{ i.e., } p = \Omega \\ p[i_k].val + f_{p[i_k].child}(x_L, ..., x_1) & \text{if } k > 0, \text{ i.e., } p \neq \Omega. \end{cases}$$

As we mostly operate on edges, we also define the function $f_{\langle \sigma, p \rangle} : \mathcal{X}_{pot} \to \mathbb{N} \cup \{\infty\}$ encoded by edge $\langle \sigma, p \rangle$ as $f_{\langle \sigma, p \rangle} = \sigma + f_p$.

The canonical forms already seen extend to EV$^+$MDDs, once we give proper care to edge values:

1. All nodes are *normalized*: for any nonterminal node p associated with v_k, we must have $p[i_k].val = 0$ for at least one $i_k \in \mathcal{X}_k$. Thus, the minimum of the function encoded by any node is 0, and a function $f : \mathcal{X}_{pot} \to \mathbb{N} \cup \{\infty\}$ is encoded by an edge $\langle \rho,p \rangle$ where $\rho = \min_{\mathbf{i} \in \mathcal{X}_{pot}} \{f(\mathbf{i})\}$.
2. All edges with value ∞ point to Ω: if $p[i_k].val = \infty$ then $p[i_k].child = \Omega$.
3. There are *no duplicates*: if $p.var = q.var = x_k$ and $p[i_k] = q[i_k]$ for all $i_k \in \mathcal{X}_k$, then $p = q$ (now, $p[i_k] = q[i_k]$ means equality for both the identities of the children and the values of the edges).
4. One of the following forms holds
   quasi-reduced form: there is no variable skipping, i.e., if $p.var = v_k$, then $p[i_k].child.var = v_{k-1}$ for any $i_k \in \mathcal{X}_k$, and $k = L$ if p is a root node.
   fully-reduced form: there is maximum variable skipping, i.e., no *redundant* node p exists, with $p[i_k].val = 0$ and $p[i_l].child = q$, for all $i_k \in \mathcal{X}_k$. Thus, a constant function $f = c \in \mathbb{N} \cup \{\infty\}$ is encoded by $\langle c,\Omega \rangle$.

Indeed, while not needed in this survey, we can even associate reduction rules with individual variables, as already discussed for MDDs. If a node q associated with an *identity-reduced* variable v_k is *singular*, i.e., it has one edge $q[i_k]$ with value 0 and all other edges with value ∞, then no edge $p[i_l]$ can point to q if $i_l = i_k$ or if it skips over a fully-reduced variable v_h such that $i_k \in \mathcal{X}_h$. In other words, EV$^+$MDD edges $\langle \infty,\Omega \rangle$ play the same role as MDD edges to the terminal **0**. The next section gives an example of MTMDDs and EV$^+$MDDs.

EV$^+$MDD manipulation algorithms are somewhat more complex than their MDD or MTMDD analogues, as node normalization requires the propagation of numerical values along the edges of the diagram, but they retain the overall recursive flavor seen for other decision diagrams. For example, Fig. 19 shows the pseudocode for the *Minimum* operator, assuming quasi-reduced EV$^+$MDDs. Observe that, when looking up the operation cache in the computation of the minimum of $\langle \alpha,p \rangle$ and $\langle \beta,q \rangle$, we can swap p and q to force a particular order on them (e.g., increasing *node_id*), since *Minimum* is commutative. Furthermore, we can just focus on the difference $\alpha - \beta$, which can be positive, negative, or zero, instead on the actual values of α and β, since $Minimum(\langle 3,p \rangle, \langle 5,q \rangle) = 3 + Minimum(\langle 0,p \rangle, \langle 2,q \rangle)$. As the result is always going to be of the form $\langle \min\{\alpha,\beta\},r \rangle$, thus $\langle 3,r \rangle$ in our example, the cache only needs to remember the node portion of the result, r.

We conclude this section by stressing a key difference between the original definition of EVBDDs [37] and our EV$^+$MDDs (beyond the facts that EV$^+$MDDs allow a non-binary choice at each nonterminal node and the use of ∞ as an edge value, both of which could be considered as "obvious" extensions). EVBDDs extended BDDs by associating an integer (thus possibly negative) value to each edge, and achieved canonicity by requiring that $p[0].val = 0$, so that a function $f : \mathcal{X}_{pot} \to \mathbb{N}$ is encoded by an edge $\langle \rho,p \rangle$ where $\rho = f(0,...,0)$. While this canonical form has smaller memory requirements, as it eliminates the need to explicitly store $p[0].val$, it cannot encode all partial functions, thus is not as

```
edge Minimum(edge ⟨α,p⟩, edge ⟨β,q⟩)                    • edge is a pair ⟨int,node⟩
 1 if α = ∞ then return ⟨β,q⟩;
 2 if β = ∞ then return ⟨α,p⟩;
 3 v_k ← p.var;                                          • same as q.var
 4 μ ← min{α, β};
 5 if p = q then return ⟨μ,p⟩;          • includes the case k = 0, i.e., p = q = Ω
 6 if Cache contains entry ⟨Minimum, p, q, α − β : r⟩ then return ⟨μ,r⟩;
 7 r ← NewNode(k); • create new node associated with v_k with edges set to ⟨∞,Ω⟩
 8 foreach i_k ∈ X_k do
 9    r[i_k] ← Minimum(⟨α−μ+p[i_k].val,p[i_k].child⟩, ⟨β−μ+q[i_k].val,q[i_k].child⟩);
10 UniqueTableInsert(r);
11 enter ⟨Minimum, p, q, α − β : r⟩ in Cache;
12 return ⟨μ,r⟩;
```

Fig. 19. The *Minimum* operator for quasi-reduced EV$^+$MDDs [19].

general as EV$^+$MDDs. For example, EVBDDs cannot encode the function δ considered in Fig. 21, because $\delta(1,0,0) = \infty$ but $\delta(1,0,1) = 4$, thus there is no way to normalize the node associated with x_1 on the path where $x_3 = 1$ and $x_2 = 0$ in such a way that its 0-edge has value 0.

3.4 Symbolic computation of CTL witnesses

A *witness* for an *existential* CTL formula is a finite path $\mathbf{i}^{(0)}$, $\mathbf{i}^{(1)}$, ..., $\mathbf{i}^{(d)}$ of markings in the reachability graph, with $\mathbf{i}^{(0)} \in \mathcal{X}_{init}$, which proves the validity of the formula. For $\mathsf{EX}a$ and $\mathsf{EF}a$, assuming a is an atomic proposition, we simply require that a holds in $\mathbf{i}^{(1)}$, or $\mathbf{i}^{(d)}$ for some $d \geq 0$, respectively. For $\mathsf{E}a\mathsf{U}b$, a must hold in $\mathbf{i}^{(0)}, \mathbf{i}^{(1)}, ..., \mathbf{i}^{(d-1)}$, and b, also assuming it is an atomic proposition, must hold in $\mathbf{i}^{(d)}$. For $\mathsf{EG}a$, all the markings on the path must satisfy a and, in addition, $\mathbf{i}^{(d)}$ must equal $\mathbf{i}^{(m)}$, for some $m \in \{0, ..., d-1\}$, so that the path contains a cycle. Of course, we cannot provide witnesses to a *universal* CTL formula, but we can disprove it with a *counterexample*, i.e., a witness for its negation.

Witnesses and counterexamples can help us understand and debug a complex model and they are most useful when they are short, thus, ideally, we seek minimal length ones. We then define the *distance function* $\delta : \mathcal{X}_{pot} \to \mathbb{N} \cup \{\infty\}$ as $\delta(\mathbf{i}) = \min\{d : \mathbf{i} \in \mathcal{N}^d(\mathcal{X}_{init})\}$, so that $\delta(\mathbf{i}) = \infty$ iff $\mathbf{i} \notin \mathcal{X}_{rch}$.

We can build δ as a sequence of $d_{max} + 1$ sets encoded as MDDs, where d_{max} is the maximum distance of any marking from $\mathcal{X}_{init}$, using one of the two algorithms shown in Fig. 20. The difference between the two algorithms is that $\mathcal{X}^{[d]}$ contains the markings at distance *exactly* d while $\mathcal{Y}^{[d]}$ contains the markings at distance *up to* d (this is analogous to the "frontier vs. all" approaches to reachability-set generation in Fig. 8). In fact, these algorithms essentially also build the reachability set as a byproduct, since $\mathcal{X}_{rch} = \bigcup_{d=0}^{d_{max}} \mathcal{X}^{[d]} = \mathcal{Y}^{[d_{max}]}$.

Instead of encoding the distance function with a sequence of MDDs, we can encode it more naturally with a single decision diagram having range $\mathbb{N} \cup \{\infty\}$. Fig. 21(a,b) illustrates how the same distance function δ is encoded by five

Build $\mathcal{X}^{[d]} = \{\mathbf{i} : \delta(\mathbf{i}) = d\}$, $d = 0, 1, ..., d_{max}$

DistanceMddEQ$(\mathcal{X}_{init}, \mathcal{N})$ is

```
1 d ← 0;
2 X_rch ← X_init;
3 X^[0] ← X_init;
4 repeat
5     X^[d+1] ← N(X^[d]) \ X_rch;
6     d ← d + 1;
7     X_rch ← X_rch ∪ X^[d];
8 until X^[d] = ∅;
9 return X^[0], ..., X^[d−1];
```

Build $\mathcal{Y}^{[d]} = \{\mathbf{i} : \delta(\mathbf{i}) \leq d\}$, $d = 0, 1, ..., d_{max}$

DistanceMddLE$(\mathcal{X}_{init}, \mathcal{N})$ is

```
1 d ← 0;
2 Y^[0] ← X_init;
3 repeat
4     Y^[d+1] ← N(Y^[d]) ∪ Y^[d];
5     d ← d + 1;
6 until Y^[d] = Y^[d−1];
7 return Y^[0], ..., Y^[d−1];
```

Fig. 20. Two ways to build a set of MDDs encoding the distance function.

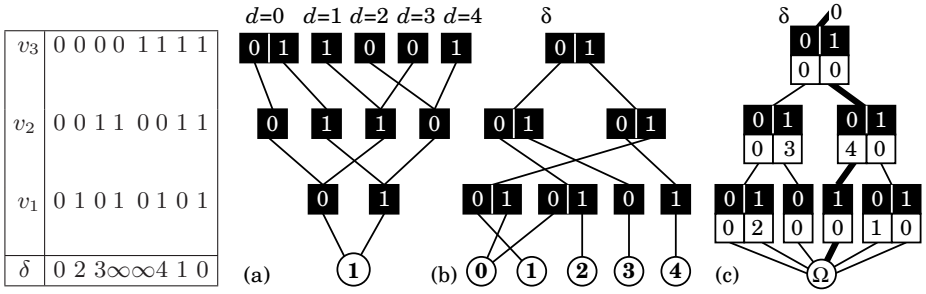

Fig. 21. Encoding the distance function: sequence of MDDs, a MTMDD, an EV$^+$MDD.

MDDs or a single MTMDD with five terminal nodes (plus terminal ∞, which is omitted). However, while MTMDDs are a natural and often useful extension, it is obvious that they can have poor merging toward the bottom of the diagram, just as a sequence of MDDs might have an overall poor merging toward the top. The real problem is that both approaches are explicit in the number of distinct distance values. If the maximum distance d_{max} is very large, these symbolic encodings do not help much: we need edge values, that is, EV$^+$MDDs.

The canonical EV$^+$MDD encoding the distance function δ previously considered is shown in Fig. 21(c), where edges with value ∞ are omitted and the highlighted path describes $\delta(1, 0, 1) = 0 + 0 + 4 + 0 = 4$. While this example does not demonstrate the benefits of EV$^+$MDDs over MTMDDs, one can simply consider the function $f(i_L, ..., i_1) = \sum_{k=1}^{L} i_k \cdot \prod_{l=1}^{k-1} |\mathcal{X}_l|$ to realize that an EV$^+$MDD for this function requires L nonterminal nodes and $\sum_{k=1}^{L} |\mathcal{X}_l|$ edges, an exponential improvement over a canonical MTMDD, which requires $\sum_{k=1}^{L} \prod_{l=2}^{k} |\mathcal{X}_l|$ nonterminal nodes and $\sum_{k=1}^{L} \prod_{l=1}^{k} |\mathcal{X}_l|$ edges.

While storing the distance function using EV$^+$MDDs instead of multiple MDDs or a MTMDD has the potential of improving memory and runtime, much larger improvements are possible if we, again, apply the idea of saturation. First of all, we need to revisit the idea of how the distance function is computed. If we

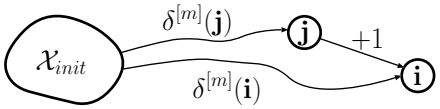

Fig. 22. Computing the distance function as a fixpoint.

follow a traditional breadth-first thinking, we simply initialize δ to the default constant function ∞, then lower its value to 0 for any marking in $\mathcal{X}^{[0]} = \mathcal{X}_{init}$, then lower its value to 1 for any marking reachable from $\mathcal{X}^{[0]}$ in one application of $\mathcal{N}$ and not already in $\mathcal{X}^{[0]}$, then lower its value to 2 for any marking reachable from $\mathcal{X}^{[1]}$ in one application of $\mathcal{N}$ and not already in $\mathcal{X}^{[0]}$ or $\mathcal{X}^{[1]}$, and so on. However, we can also think of the distance function δ as the *fixpoint* of the sequence $(\delta^{[m]} : m \in \mathbb{N})$, initialized with $\delta^{[0]}$ satisfying $\delta^{[0]}(\mathbf{i}) = 0$ if $\mathbf{i} \in \mathcal{X}_{init}$ and ∞ otherwise, and recursively updated by choosing an $\alpha \in \mathcal{T}$ and letting $\delta^{[m+1]}$ satisfy $\delta^{[m+1]}(\mathbf{i}) = \min\left\{\min_{\mathbf{j}\in\mathcal{N}_\alpha^{-1}(\mathbf{i})}\{1 + \delta^{[m]}(\mathbf{j})\}, \delta^{[m]}(\mathbf{i})\right\}$, as illustrated in Fig. 22. With this approach, each iteration reduces the value of $\delta^{[m]}(\mathbf{i})$ for at least one marking $\mathbf{i}$, until no more reduction is possible for any $\alpha \in \mathcal{T}$. The value of the distance function for a particular marking $\mathbf{i}$ may be reduced multiple times, as we keep finding new ways to improve our estimate, unlike the traditional *breadth-first* iteration which reduces this value exactly once if $\mathbf{i}$ is reachable: from $\delta^{[m]}(\mathbf{i}) = \infty$, for values of m less than the actual distance d of $\mathbf{i}$, to $\delta^{[m]}(\mathbf{i}) = d$, for values of m greater or equal d.

Fig. 23 shows the memory and time requirements to generate the distance function δ using three approaches: an EV$^+$MDD and saturation (E_s), MDDs and the breadth-first iterations of Fig. 20 on the left, and then either accumulating the distances of the markings $\mathcal{X}^{[d]}$ into a single EV$^+$MDD (E_b), or keeping them as separate MDDs (M_b). Of course, E_s and E_b result in the same final EV$^+$MDD encoding of δ, but they greatly differ in peak memory requirements, thus time. Except for the **queen** model, where all techniques perform approximately the same (and not particularly well compared to an explicit approach, since the decision diagram is quite close to a tree in all cases), the results clearly show the superiority of EV$^+$MDDs over the simpler MDDs for the final encoding of δ. More importantly, the results show the enormous memory and time superiority of saturation over breadth-first search, except for the **queen** model (where E_s is better than E_b but both are somewhat worse than M_b), and the **leader** model (where E_s uses half the memory and twice the time of E_b, but both are worse than M_b, sometimes by up to a factor of two or three, even if their final encoding of δ is over four times better).

Once the distance function has been generated and encoded as an EV$^+$MDD $\langle\rho_*,r_*\rangle$, we can generate a minimal witness for $\mathsf{EF}q$ as shown in Fig. 24, assuming that q_* is the MDD encoding the nonempty set of reachable markings satisfying q. First, we build the EV$^+$MDD $\langle 0,x\rangle$, which is essentially the same as the MDD q^*, except that (1) all its edges have an associated value of 0 unless the MDD edge points to $\mathbf{0}$, in which case the associated value is ∞, and (2) the two MDD

N	$\lvert\mathcal{X}_{rch}\rvert$	d_{max}	**Time**			δ **mem.**		**Peak mem.**		
			E_s	E_b	M_b	E_s, E_b	M_b	E_s	E_b	M_b
phils										
100	$4.96{\cdot}10^{62}$	200	0.01	1.33	1.13	0.06	4.20	0.25	36.60	24.31
200	$2.46{\cdot}10^{125}$	400	0.02	21.60	20.07	0.11	17.10	0.44	117.14	70.86
500	$3.03{\cdot}10^{313}$	1000	0.05	–	–	0.28	–	0.92	–	–
robin										
50	$1.26{\cdot}10^{17}$	394	0.01	2.04	1.88	0.12	2.78	1.33	34.06	25.06
100	$2.85{\cdot}10^{32}$	794	0.05	33.53	31.76	0.43	16.91	7.36	307.14	257.29
150	$4.81{\cdot}10^{47}$	1194	0.10	174.57	168.34	0.93	50.99	23.20	1206.50	1056.45
fms										
10	$2.53{\cdot}10^{7}$	80	0.02	0.82	0.77	0.03	0.73	0.71	19.89	17.56
15	$7.24{\cdot}10^{8}$	120	0.06	5.49	5.00	0.06	2.24	1.55	57.05	49.23
20	$8.83{\cdot}10^{9}$	160	0.17	24.56	24.08	0.10	5.22	3.24	118.53	102.65
slot										
10	$8.29{\cdot}10^{9}$	114	0.04	0.44	0.39	0.03	0.87	0.53	11.75	8.74
20	$2.73{\cdot}10^{20}$	379	0.34	30.12	28.49	0.17	15.95	3.73	120.93	66.42
30	$1.03{\cdot}10^{31}$	794	1.41	2576.32	2336.05	0.45	92.56	12.50	703.54	388.75
kanban										
20	$8.05{\cdot}10^{11}$	280	0.85	3.26	2.92	0.10	9.32	4.26	75.94	49.57
30	$4.98{\cdot}10^{13}$	420	6.71	28.63	26.99	0.29	43.27	18.65	291.22	176.74
40	$9.94{\cdot}10^{14}$	560	38.84	180.23	162.43	0.66	130.92	43.78	836.24	452.74
leader										
6	$1.89{\cdot}10^{6}$	93	2.64	1.59	1.16	0.85	3.31	13.36	32.12	15.65
7	$2.39{\cdot}10^{7}$	115	16.88	8.63	6.29	2.56	10.79	41.36	96.01	39.08
8	$3.04{\cdot}10^{8}$	139	122.38	69.03	52.72	6.99	31.29	126.79	270.43	91.23
counter										
10	$1.02{\cdot}10^{3}$	$10^{10}-1$	0.00	0.01	0.01	0.00	0.11	0.03	0.70	0.39
20	$1.04{\cdot}10^{6}$	$10^{20}-1$	0.00	400.42	95.13	0.00	112.00	0.04	510.09	183.41
30	$1.07{\cdot}10^{9}$	$10^{30}-1$	0.00	–	–	0.00	–	0.04	–	–
queen										
12	$8.56{\cdot}10^{5}$	12	3.75	3.77	2.77	23.54	20.77	71.84	95.07	63.69
13	$4.67{\cdot}10^{6}$	13	21.57	21.53	16.59	112.61	99.46	340.26	452.85	309.37
14	$2.73{\cdot}10^{7}$	14	127.69	130.78	102.06	572.20	505.74	1737.16	2308.17	1577.99

Fig. 23. Results for distance function computation (memory: MB, time: sec).

terminals **0** and **1** are merged into the unique EV$^+$MDD terminal Ω. Then, we build the EV$^+$MDD $\langle\mu,m\rangle$ encoding the elementwise maximum of the functions encoded by $\langle\rho_*,r_*\rangle$ and $\langle 0,x\rangle$). Note that μ is then the length of one of the minimal witnesses we are seeking. Then, we simply extract from $\langle\mu,m\rangle$ a marking $\mathbf{j}^{(\mu)} = (j_L^{(\mu)}, \ldots, j_1^{(\mu)})$ on a 0-valued path from m to Ω, which is then a reachable marking satisfying q and at the minimum distance μ from $\mathcal{X}_{init}$. Finally, we "walk back" from $\mathbf{j}^{(\mu)}$, finding at each iteration a marking one step closer to $\mathcal{X}_{init}$. We have described this last sequence of steps in an explicit, not symbolic, fashion since its complexity is minimal as long as each $\mathcal{N}^{-1}(\mathbf{j}^{(\nu+1)})$ is small; were this not the case, a symbolic implementation is certainly possible.

```
marking sequence  Witness(edge ⟨ρ*,r*⟩, mdd q*)
 1 ⟨0,x⟩ ← MddToEvmdd(q*);   • f_⟨0,x⟩(i)=0 if f_q*(i)=1, f_⟨0,x⟩(i)=∞ otherwise
 2 ⟨μ,m⟩ ← Maximum(⟨ρ*,r*⟩, ⟨0,x⟩);   • analogous to the Minimum algorithm
 3 choose a marking j^(μ) on a 0-valued path in ⟨μ,m⟩;
 4 for ν = μ − 1 downto 0 do   • walk backward a witness of length μ
 5   foreach i ∈ N^-1(j^(ν+1)) do
 6     if f_⟨ρ*,r*⟩(i) = ν then   • there must be at least one such marking i
 7       j^(ν) ← i;
 8       break;   • move to the next value of ν
 9 return (j^(0), ..., j^(μ));
```

Fig. 24. Minimal EF witness computation using the distance function [19].

3.5 Bounded model checking

When the reachability set is infinite (because some place is unbounded), or even simply when its MDD encoding is unwieldy, even the powerful symbolic methods seen so far fail. *Bounded model checking* has then been proposed as a practical alternative in these cases. Usually implemented through SAT solvers [6], the idea of bounded model checking is in principle independent of their use, as it simply centers around the idea of exploring only a smaller (and certainly finite) portion of the reachability set. Answers to certain CTL queries may still be obtained, as long as no marking can reach an infinite number of states in a single step, which is certainly true for Petri nets, even in the self-modifying case.

As initially proposed, bounded model checking explores only markings whose distance from the initial markings does not exceed a given a bound B. Then, for *safety* queries, we might find a counterexample, or prove that no counterexample exists, or, in the worst case, determine that no incorrect behavior arises within the first B steps (in this last inconclusive case, we might choose to increase B and repeat the analysis). Obviously, B breadth-first iterations starting from $\mathcal{X}_{init}$ can be used to encode the bounded reachability set as a BDD or MDD, but performance tends to be poor compared to SAT-based methods [30, 31, 42].

One reason for this is that, as we have already observed, the MDD encoding the set of markings reachable in exactly, or up to, d steps is often quite large. Saturation tends to be much better than breadth-first iteration when exploring the entire reachability set but, for bounded exploration, it does not offer an obvious way to limit its exploration to markings within a given distance from $\mathcal{X}_{init}$: ordinary saturation uses an MDD and computes a fixpoint at each node p associated with v_k, stopping only when p encodes all (sub)markings reachable from the (sub)markings it initially encoded, by firing any α with $Top(\alpha) \leq k$.

We then introduced a *bounded saturation* approach that uses an EV$^+$MDD, not simply an MDD, to build a *truncated distance function* δ_{trunc} and bound the amount of exploration on a node p associated with v_k, in one of two ways [56]:

- Ensure that, for any marking $\mathbf{i} \in \mathcal{X}_L \times \cdots \times \mathcal{X}_1$, $\delta_{trunc}(\mathbf{i}) = \delta(\mathbf{i})$ if $\delta(\mathbf{i}) \leq B$ and $\delta_{trunc}(\mathbf{i}) = \infty$ otherwise (*EVMDD-Exact* method).
- Ensure that, for any marking $\mathbf{i} \in \mathcal{X}_L \times \cdots \times \mathcal{X}_1$, $\delta_{trunc}(\mathbf{i}) = \delta(\mathbf{i})$ if $\delta(\mathbf{i}) \leq B$ and $\delta_{trunc}(\mathbf{i}) \geq \delta(\mathbf{i})$ otherwise, while also ensuring that, for any EV$^+$MDD

node associated with v_k and any $i_k \in \mathcal{X}_k$, $p[i_k].val \leq B$ or $p[i_k].val = \infty$ (*EVMDD-Approx* method).

While *EVMDD-Exact* finds only and all the markings with distance exactly up to B, *EVMDD-Approx* finds not only all those markings, but also many other markings with distance up to $L \cdot B$. Even with these additional markings being encoded in the EV$^+$MDD, the *EVMDD-Approx* method is sometimes by far the best approach, as shown in Fig. 25, which compares the traditional BFS bounded exploration using MDDs with the two approaches just described. In addition, since *EVMDD-Approx* explores a larger set of markings $\mathcal{X}_{expl}$, it is more likely to find the (un)desired behavior we are looking than its exact alternatives, for a given bound B. Indeed, for the two models where *EVMDD-Approx* performs much worse than breadth-first search, the former ends up exploring the entire state space, i.e., $\mathcal{X}_{expl} = \mathcal{X}_{rch}$ (in this case, of course, ordinary saturation using MDDs would be much more efficient than bounded saturation using EV$^+$MDDs).

4 Further synergy of decision diagrams and Petri nets

So far, we have considered logical applications of decision diagram technology for the analysis of Petri nets. Even the non-boolean forms of decision diagrams we introduced, MTMDDs and EV$^+$MDDs, have after all been used only to count distances between markings. However, analysis techniques involving numerical computations are often required, especially when working with extensions of the Petri net formalism that take into account timing and probabilistic behavior, and decision diagrams have been successfully employed here as well. We now briefly discuss three such applications where saturation might also be applied, at least on some of the required computation. The last topic of this section, instead of presenting decision diagram algorithms that improve Petri net analysis, illustrates how the help can also "flow the other way", by discussing a heuristic based on the invariants of a Petri net to derive a good order for the variables of the decision diagrams used to study the Petri net itself.

4.1 P-semiflow computation

Invariant or semiflow analysis can provide fast (partial) answers to Petri net reachability questions. A *p-semiflow* is a non-negative, non-zero integer solution $\mathbf{w} \in \mathbb{N}^{|\mathcal{P}|}$ to the set of linear flow equations $\mathbf{w}^T \cdot \mathbf{D} = \mathbf{0}$, where $\mathbf{D} = \mathbf{D}^+ - \mathbf{D}^-$ is the *incidence matrix*. If there exists a p-semiflow $\mathbf{w}$ with $w_p > 0$, place p is bounded regardless of the initial marking $\mathbf{i}_{init}$. Indeed, given $\mathbf{i}_{init}$, we can conclude that $i_p \leq (\mathbf{w} \cdot \mathbf{i}_{init})/w_p$ in any reachable marking $\mathbf{i}$. P-semiflows provide necessary, not sufficient, conditions on reachability: we can conclude that a marking $\mathbf{i}$ is not reachable if there is p-semiflow $\mathbf{w}$ such that $\mathbf{w} \cdot \mathbf{i} \neq \mathbf{w} \cdot \mathbf{i}_{init}$.

Explicit p-semiflow computation. As any nonnegative integer linear combination of p-semiflows is a p-semiflow, we usually seek the (unique) *generator*

N	*BFS*			*EVMDD-Exact*		*EVMDD-Approx*		
	$\mathcal{X}_{rch}$	**time**	**mem.**	**time**	**mem.**	$\mathcal{X}_{rch}$	**time**	**mem.**
phils : $B=100$								
100	2.13×10^{59}	0.85	18.52	6.38	14.30	4.96×10^{62}	0.10	1.53
200	1.16×10^{86}	2.63	38.56	22.32	41.10	2.46×10^{125}	0.20	2.60
300	1.18×10^{101}	5.60	89.27	42.54	70.20	1.22×10^{188}	0.31	2.93
robin : $B=N\times2$								
100	1.25×10^{18}	2.44	30.17	2.34	22.04	8.11×10^{20}	0.12	1.70
200	2.65×10^{35}	37.72	250.80	35.06	264.61	7.62×10^{40}	1.05	9.18
300	5.61×10^{52}	192.53	909.86	187.83	1091.43	9.64×10^{60}	3.98	28.09
fms : $B=50$								
10	2.20×10^{7}	0.57	5.63	1.55	9.45	2.53×10^{7}	0.22	1.98
15	1.32×10^{8}	1.93	11.24	28.59	28.52	7.24×10^{8}	0.90	4.27
20	2.06×10^{8}	4.27	19.40	414.29	44.47	8.82×10^{9}	2.55	4.28
slot : $B=30$								
20	1.57×10^{14}	0.10	3.65	7.30	7.49	1.58×10^{20}	0.19	1.14
25	1.93×10^{16}	0.13	4.98	28.17	10.31	2.41×10^{25}	0.26	1.50
30	1.14×10^{18}	0.16	6.12	79.26	12.63	3.69×10^{30}	0.34	2.12
kanban : $B=100$								
11	7.35×10^{8}	1.52	18.77	0.74	6.38	2.43×10^{9}	0.23	6.10
12	1.05×10^{9}	2.51	29.70	1.11	9.30	5.51×10^{9}	0.34	9.13
13	1.42×10^{9}	3.45	25.87	1.57	11.79	1.18×10^{10}	0.53	12.09
leader : $B=50$								
7	7.34×10^{6}	0.94	6.02	90.03	70.55	2.38×10^{7}	18.54	22.33
8	4.73×10^{7}	1.97	6.91	1342.46	204.01	3.04×10^{8}	143.67	72.37
9	3.11×10^{8}	4.15	8.28	–	–	3.87×10^{9}	1853.66	243.89
counter : $B=2^{N/10}$								
100	1.02×10^{3}	0.32	13.39	0.01	0.12	2.04×10^{3}	0.02	0.07
200	1.04×10^{6}	745.80	107.62	3.31	36.96	2.09×10^{6}	0.06	0.11
300	–	–	–	–	–	2.14×10^{9}	0.18	0.14
queen : $B=5$								
13	3.86×10^{4}	0.06	1.18	0.20	2.40	3.86×10^{4}	0.15	1.80
14	6.52×10^{4}	0.07	1.73	0.35	3.58	6.52×10^{4}	0.25	2.97
15	1.05×10^{5}	0.10	2.58	0.58	5.59	1.05×10^{5}	0.42	4.45

Fig. 25. Results for bounded saturation (memory: MB, time: sec).

$\mathcal{G}$, i.e., the set of *minimal* p-semiflows $\mathbf{w}$ satisfying: (1) $\mathbf{w}$ has a minimal support $Supp(\mathbf{w}) = \{p \in \mathcal{P} : w_p > 0\}$, i.e., no other p-semiflow $\mathbf{w}'$ exists with $Supp(\mathbf{w}') \subset Supp(\mathbf{w})$, and (2) $\mathbf{w}$ is scaled back, i.e., the greatest common divisor of its (non-zero) entries is one.

The standard explicit approach to compute $\mathcal{G}$ is Farka's algorithm [26], which manipulates a matrix $[\mathbf{T}|\mathbf{P}]$ stored as a set $\mathcal{A}$ of integer row vectors of length $m+n$, where m and n are the number of transitions and places, respectively. Initially, $[\mathbf{T}|\mathbf{P}] = [\mathbf{D}\,|\,\mathbf{I}] \in \mathbb{Z}^{n\times m} \times \mathbb{N}^{n\times n}$, where $\mathbf{D}$ is the flow matrix and $\mathbf{I}$ is the $n \times n$ identity matrix, thus $\mathcal{A}$ contains n rows. Then, we iteratively force zero entries in column j for $j = 1, ..., m$, using the following process:

1. Let $\mathcal{A}_N$ ($\mathcal{A}_P$) be the set of rows with negative (positive) j-entry, respectively.
2. Remove the rows in $\mathcal{A}_N$ or $\mathcal{A}_P$ from $\mathcal{A}$.
3. For each $\mathbf{a}_N \in \mathcal{A}_N$ and $\mathbf{a}_P \in \mathcal{A}_P$, add row $\{(-v/\mathbf{a}_N[j]) \cdot \mathbf{a}_N + (v/\mathbf{a}_P[j]) \cdot \mathbf{a}_P\}$ to $\mathcal{A}$, where v is the minimum common multiple of $-\mathbf{a}_N[j]$ and $\mathbf{a}_P[j]$.

The number of rows may grow quadratically at each step, as we add $|\mathcal{A}_N| \cdot |\mathcal{A}_P|$ rows to $\mathcal{A}$ but only remove $|\mathcal{A}_N| + |\mathcal{A}_P|$ rows from it (the number decreases iff $\mathcal{A}_N$ or $\mathcal{A}_P$ is a singleton). Thus $\mathcal{A}$ can grow exponentially in m in pathological cases. Also, if at the beginning of a step either $\mathcal{A}_N$ or $\mathcal{A}_P$ is empty but not both, no p-semiflows exist.

Once the first m columns of $[\mathbf{T}|\mathbf{P}]$ are all zeros, $\mathbf{P}$, i.e., the rows in $\mathcal{A}$ describe the required p-semiflows, except that some of them may be need to be scaled back and some might not have minimal support. Scaling back $\mathcal{A}$ can be accomplished by dividing each $\mathbf{a} \in \mathcal{A}$ by the GCD of its entries, with time complexity $O(|\mathcal{A}| \cdot n)$. To eliminate non-minimal support rows, instead, we compare the supports of each *pair* of distinct $\mathbf{a}$ and $\mathbf{b}$ in $\mathcal{A}$ and delete $\mathbf{b}$ if its support is a superset of that of $\mathbf{a}$, with time complexity $O(|\mathcal{A}|^2 \cdot n)$. Alternatively, we can minimize $\mathcal{A}$ during Farka's algorithm, so that, before iteration j, $\mathcal{A}$ contains the minimal support p-semiflows ignoring transitions $j, ..., m$, and iteration j adds a row to $\mathcal{A}$ only if it scaled back and its support is not a superset of that of a row already in $\mathcal{A}$; this tends to perform better in practice.

Symbolic p-semiflow computation using zero-suppressed MDDs [16]. As described, Farka's algorithm manages a set of rows with elements from $\mathbb{Z}$ (in the first m columns) or $\mathbb{N}$ (in the last n columns). This is similar to managing a set of markings, for which MDDs are ideally suited, with two differences:

- The first m row can have negative entries. We cope by letting MDD nodes have index set $\mathbb{Z}$ instead of $\mathbb{N}$. In either case, the key restriction is the same, only a finite number of indices can be on a path to the terminal node $\mathbf{1}$.
- Many of the entries are zero (and the first m entries of each row will be zero at the end). While this is not a problem for MDDs, it turns out that a new MDD reduction rule (initially proposed for BDDs [55]) improves efficiency. We use *zero-suppressed* MDDs (ZMDDs) [16], where no node p has $p[i_k] = \mathbf{0}$ for all $i_k \neq 0$, and the semantic of an edge $p[i_k] = q$ skipping over variable v_h, is the same as if v_h were quasi-reduced and we inserted a node p' associated with v_h along that edge, with $p'[0] = q$ and $p'[i_h] = \mathbf{0}$ for $i_h \neq 0$, see Fig. 26.

In [16] we described a fully symbolic algorithm for p-semiflow computation that implements Farka's algorithm using an $(m+n)$-variable ZMDD to encode the set of rows $\mathcal{A}$. After the j^{th} iteration, all the rows in $\mathcal{A}$ have zero entries in the first j columns, thus the ZMDD encoding of $\mathcal{A}$ skips those first j variables altogether. Symbolic algorithms were provided to perform the required linear combinations, to eliminate non-minimal support p-semiflows (either periodically or at the end), and to scale back a set of p-semiflows.

Experimental results show that our symbolic approach is more efficient than the explicit algorithm implemented in GreatSPN [3] for large enough instances

$\mathcal{A} = \{000, 001, 002, 110, 111, 112\}$

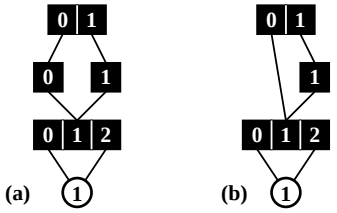

Fig. 26. An MDD (a) and a ZMDD (b) encoding the same set $\mathcal{A}$.

of Petri nets whose generator contains even just a few p-semiflows, while it is of course enormously more efficient when the generator is pathologically large.

While our symbolic algorithms are not expressed as fixpoints, thus do not use saturation, it is conceivable that saturation could be used, especially for the elimination of non-minimal support p-semiflows, to further improve efficiency.

4.2 Integer timed Petri nets

Two classic ways to incorporate timed, but not probabilistic, behavior in Petri nets are *timed Petri nets* [58] (the firing time of a transition α is a non-negative real constant) and *time Petri nets* [40] (the firing time lies in a non-negative real interval). In [53], we considered a variant, *integer-timed Petri nets* (ITPNs), where the firing time of a transition is nondeterministically chosen from a finite set of positive integers, and explored two fundamental reachability problems:

- *Timed reachability*: find the set of markings where the Petri net can be at a given finite point θ_f in time.
- *Earliest reachability*: find the first instant of time $\epsilon(\mathbf{i})$ when the Petri net might enter each reachable marking $\mathbf{i}$.

We tackle timed reachability by performing a symbolic simultaneous simulation that manipulates sets of *states* of the form $(\mathbf{i}, \boldsymbol{\tau})$, where $\mathbf{i} \in \mathbb{N}^{|\mathcal{P}|}$ is a marking and $\boldsymbol{\tau} \in (\mathbb{N} \cup \{\infty\})^{|\mathcal{T}|}$ is a vector of *remaining firing times* (RFTs), so that τ_t is the number of time units until t will attempt to fire (∞ if t is disabled in $\mathbf{i}$). We use an MDD on $|\mathcal{P}| + |\mathcal{T}|$ variables to encode the set of states in which the ITPN might be at global time θ, starting with $\theta = 0$. Then, we iteratively:

1. Compute the minimum RFT τ_b for any state encoded by the MDD.
2. Advance θ to the next *breakpoint* $\theta + \tau_b$, and update the MDD by subtracting τ_b from every RFT.
3. Now the MDD contains some *vanishing* states (having one or more RFT equal to zero). Fire all possible *maximally serializable* sets of transitions having zero RFT, until finding all the *tangible* states (having only positive remaining firing times) reachable without further advancing the global time.

When θ reaches the desired final time θ_f, we stop and call a function *Strip* that, given the MDD encoding the states $\mathcal{S}_\theta$ reachable at time θ, computes the MDD encoding just the corresponding markings $\mathcal{X}_\theta = \{\mathbf{i} : \exists \boldsymbol{\tau}, (\mathbf{i}, \boldsymbol{\tau}) \in \mathcal{S}_\theta\}$.

For earliest reachability, we carry out timed reachability, calling *Strip* at every breakpoint, and accumulating the results in an EV$^+$MDD over just the place variables. Initially, the EV$^+$MDD encodes the function $\epsilon(\mathbf{i}) = 0$ if $\mathbf{i}$ is an initial marking, and $\epsilon(\mathbf{i}) = \infty$ otherwise. Then, after stripping the set of markings $\mathcal{X}_\theta$ from the set of states $\mathcal{S}_\theta$ encoded by the MDD at time θ, we transform it into an EV$^+$MDD ϵ_θ evaluating to θ if $\mathbf{i} \in \mathcal{X}_\theta$ and to ∞ otherwise, and we update the EV$^+$MDD so that it encodes the function $\min(\epsilon, \epsilon_\theta)$. In other words, we set the earliest reachability time of any marking in $\mathcal{X}_\theta$ to θ, unless it has already been reached earlier, thus already has a smaller earliest reachability time.

The timed reachability algorithm is guaranteed to halt because θ_f is finite, all firing times are positive integers, and the number of possible firing times for a transitions is finite, However, the earliest reachability algorithm is guaranteed to halt only if the set of reachable markings is finite. Furthermore, the earliest reachability algorithm can be halted only if we accumulate the set of *states* encountered during the timed reachability iterations: failure to increase this set at a breakpoint indicates that no more states can be found; however, accumulating just the reachable markings, or even their earliest reachability function, is not enough, as it is possible to advance to a breakpoint and increase the set of states encountered so far, but not the set of markings.

The results in [53] indicate excellent scalability, showing that problems with very large reachable state spaces can be explored, thanks to the use of symbolic data structures and algorithms. In particular saturation is used in the computation of the tangible states reachable after advancing to a new breakpoint, as this process can be expressed as a fixpoint computation where we need to fire all transitions with a zero RFT in all possible (non-conflicting) sequences. Of course, much of the approach presented is feasible due to the assumption of a discrete set of possible positive integer firing times. Extension to more general firing time assumptions is an interesting challenge that remains to be explored.

4.3 Continuous-time markovian Petri nets

As a last application of decision diagram technology to Petri net analysis, we now consider the generation and numerical solution of Markov models.

Generalized stochastic Petri nets (GSPNs) and their explicit solution. GSPNs [1] extend the (untimed) Petri net model by associating a firing time distribution to each transition. Formally, the set of transitions is partitioned into $\mathcal{T}_T$ (*timed*, with an exponentially-distributed firing time) and $\mathcal{T}_V$ (*immediate*, with a constant zero firing time). Immediate transitions have *priority* over timed transitions, thus, a marking $\mathbf{i}$ is either *vanishing*, if $\mathcal{T}(\mathbf{i}) \cap \mathcal{T}_V \neq \emptyset$, which then implies $\mathcal{T}(\mathbf{i}) \subseteq \mathcal{T}_V$ as any timed transition is disabled by definition, or *tangible*, if $\mathcal{T}(\mathbf{i}) \subseteq \mathcal{T}_T$, which includes the case of an absorbing marking, $\mathcal{T}(\mathbf{i}) = \emptyset$.

If marking $\mathbf{i}$ is vanishing, its sojourn time is 0 and the probability that a particular $t \in \mathcal{T}(\mathbf{i})$ fires in $\mathbf{i}$ is $w_t(\mathbf{i})/(\sum_{y \in \mathcal{T}(\mathbf{i})} w_y(\mathbf{i}))$, where $w : \mathcal{T}_V \times \mathbb{N}^{|P|} \to [0, +\infty)$ specifies the marking-dependent firing *weights* (i.e., unnormalized probabilities)

of the immediate transitions. If $\mathbf{i}$ is instead tangible, its sojourn time is exponentially distributed with rate $\lambda(\mathbf{i}) = \sum_{y \in \mathcal{T}(\mathbf{i})} \lambda_y(\mathbf{i})$, where $\lambda : \mathcal{T}_T \times \mathbb{N}^{|P|} \to [0, +\infty)$ specifies the marking-dependent firing rates of the timed transitions (of course, the sojourn time is infinite if $\lambda(\mathbf{i}) = 0$, i.e., if $\mathbf{i}$ is absorbing). A *race policy* is employed in this case, thus the probability of $t \in \mathcal{T}(\mathbf{i})$ firing in $\mathbf{i}$ is $\lambda_t(\mathbf{i})/\lambda(\mathbf{i})$.

Except for pathological cases containing *absorbing vanishing loops* [2], a GSPN defines an underlying *continuous-time Markov chain* (CTMC) [10] whose state space is the set of reachable tangible markings of the GSPN, $\mathcal{X}_{tan}$, and whose transition rate matrix $\mathbf{R}$ satisfies $\mathbf{R}[\mathbf{i},\mathbf{j}] = \sum_{\mathbf{i} \xrightarrow{t} \mathbf{h}} \lambda_t(\mathbf{i}) \cdot \sum_{\sigma \in \mathcal{T}_V^* : \mathbf{h} \xrightarrow{\sigma} \mathbf{j}} Pr\{\mathbf{h} \xrightarrow{\sigma}\}$, where $Pr\{\mathbf{h} \xrightarrow{\sigma}\}$ is the probability of firing the sequence σ of immediate transitions when the marking is $\mathbf{h}$; of course, $\mathbf{i}$ and $\mathbf{j}$ are tangible markings and either $\mathbf{h}$ is vanishing or $\mathbf{h} = \mathbf{j}$, σ is the empty sequence, and we let $Pr\{\mathbf{h} \xrightarrow{\sigma}\} = 1$.

Explicit approaches to GSPN analysis mostly focus on the efficient *elimination* of the vanishing markings, that is, on ways to compute the probability of reaching the *tangible frontier* reachable through immediate firings following a timed firing (although a *preservation* approach that uses as an intermediate step an embedded discrete-time Markov chain with state space including both tangible and vanishing markings has been used as well [18]). Once $\mathbf{R}$ has been obtained, the CTMC can be solved numerically for its steady-state probability (using Gauss-Seidel or Jacobi, possibly with relaxation [50]) or transient probability (using uniformization [29]), after which further measures defined at the net level through *reward measures* can be computed.

We assume that the CTMC is finite and ergodic (all tangible markings are mutually reachable), and focus on steady-state analysis, i.e., computing the probability vector $\boldsymbol{\pi} \in [0,1]^{|\mathcal{X}_{tan}|}$ solution of $\boldsymbol{\pi} \cdot \mathbf{Q} = \mathbf{0}$, where the *infinitesimal generator* $\mathbf{Q}$ is defined as $\mathbf{Q}[\mathbf{i},\mathbf{j}] = \mathbf{R}[\mathbf{i},\mathbf{j}]$ if $\mathbf{i} \neq \mathbf{j}$, and $\mathbf{Q}[\mathbf{i},\mathbf{i}] = -\sum_{\mathbf{j} \neq \mathbf{i}} \mathbf{R}[\mathbf{i},\mathbf{j}]$.

Symbolic encoding of R. As is often the case, the main obstacle to the analysis of GSPNs is the size of the reachability set $\mathcal{X}_{rch}$, or even just of its tangible portion $\mathcal{X}_{tan}$, and of the transition rate matrix $\mathbf{R}$. The symbolic generation of $\mathcal{X}_{rch}$ or $\mathcal{X}_{tan}$ can be performed along the lines of the algorithms of Section 2. However, while the weight and rate information can be ignored during reachability-set generation, the priority of immediate transitions over timed transitions must be taken into account. One way to do so is to build the MDD encoding the set of potential vanishing markings $\mathcal{X}_{potvan}$ (i.e., the union of the enabling conditions of any immediate transition) and restrict the forward function $\mathcal{N}_\alpha$ of any timed transition α to $\mathcal{N}_\alpha \cap (\mathcal{X}_{pot} \setminus \mathcal{X}_{potvan}) \times \mathcal{X}_{pot}$. Indeed, this idea can even be extended to enforce multiple priority levels, not just that of timed vs. immediate transitions [41]. An ordinary saturation approach can then be used to find the reachable markings $\mathcal{X}_{rch}$, from which the set of reachable tangible markings can be obtained as $\mathcal{X}_{tan} = \mathcal{X}_{rch} \setminus \mathcal{X}_{potvan}$.

In many practical models, immediate transitions often have very localized effect. Then, a simpler approach is possible, where a forward function MDD encodes the firing of a timed transition α followed by that of any immediate transition that may become enabled because of the firing of α. For example,

consider the **fms** GSPN of Fig. 7, where the firing of transition t_{P1} deposits a token in place P_1wM_1, enabling immediate transition t_{M1} whenever place M_1 is not empty. The forward function $\mathcal{N}_{t_{P1},t_{M1}}$ will then depend on the union of the input and output places for t_{P1}, i.e., P_1 and P_1wM_1, and of t_{M1}, i.e., P_1wM_1, M_1, and P_1M_1. Thanks to our conjunctive encoding of each forward function, the resulting MDD is usually quite manageable. Then, $\mathcal{N}_{t_{P1},t_{M1}}$ can be treated as the forward function of an ordinary timed transition. If this approach is used for all timed transitions that might enable immediate transitions, ordinary saturation can be employed for state-space generation without having to consider priorities, and the resulting algorithm is essentially as efficient as for ordinary Petri nets. This approach is applicable even if a *sequence* of immediate transitions might fire following a timed transition, although in practice it becomes inefficient to build the corresponding forward function if the possible sequences are too complex.

To encode $\mathbf{R}$, real-valued MTBDDs can be employed [36, 38], but, as we have already seen, edge-valued approaches are potentially much more efficient. Our EV*MDDs [54], the multiplicative analogue of EV$^+$MDDs, are particularly well suited to generate and store very large transition rate matrices described by GSPNs. In EV*MDDs, all edge values are real values between 0 and 1 (except for the edge pointing to the root, whose value equals the maximum of the function encoded by the EV*MDD), a node must have at least one outgoing edge with value 1, an edge with value 0 must point to Ω, and the function is evaluated by *multiplying* the edge values encountered along the path from the root to Ω.

The EV*MDD encoding $\mathbf{R}$ can be built with an approach analogous to the one used for the forward functions in GSPN state-space generation. We first build the EV*MDD encoding matrix $\mathbf{R}_{\alpha,*}$, corresponding to firing timed transition α (with the appropriate marking-dependent rates) followed by the firing of any sequence of immediate transitions (with the appropriate marking-dependent probabilities). The main difficulty is that the GSPN does not directly specify the firing probabilities of immediate transitions, only their firing weights, which must then be normalized, for example, by encoding the total marking dependent weight in each (vanishing) marking in an EV*MDD, and performing an element-wise division operation on EV*MDDs. $\mathbf{R}$ may be computed as $\sum_{\alpha\in\mathcal{T}_T} \mathbf{R}_{\alpha,*}$, or we might choose to carry on the required numerical solution leaving $\mathbf{R}$ in disjunct form, i.e., as the collection of the EV*MDDs encoding each $\mathbf{R}_{\alpha,*}$.

Unfortunately, the steady-state solution vector rarely has a compact symbolic representation (the MTMDD or EV*MDD encoding $\boldsymbol{\pi}$ is often close to a tree). Thus, the state-of-the-art for an *exact* numerical solution is a *hybrid* approach where the (huge) tangible reachability set $\mathcal{X}_{tan}$ is stored with an MDD and the (correspondingly huge) transition rate matrix $\mathbf{R}$ is stored with an EV*MDD or an MTBDD, but $\boldsymbol{\pi}$ is stored in a full real-valued vector of size $|\mathcal{X}_{tan}|$, where the mapping between tangible markings and their position in this vector, or *indexing* function $\rho : \mathcal{X}_{tan} \to \{0, ..., |\mathcal{X}_{tan}| - 1\}$, can be encoded in an EV$^+$MDD which has the remarkable property of being isomorphic to the MDD encoding $\mathcal{X}_{tan}$ if the index of marking $\mathbf{i}$ is its lexicographic position in $\mathcal{X}_{tan}$ [17].

While this enables the solution of substantially larger CTMC models, since the memory for $\boldsymbol{\pi}$ is a small fraction of what would be required for an explicit storage of $\mathbf{R}$, it falls short of enabling the analysis of models having similar size as those for which we can answer reachability or CTL queries. In [54], we propose a technique that obviates the need to allocate this full vector, but uses instead memory proportional to the size of the MDD encoding $\mathcal{X}_{tan}$ to store an *approximation* of the solution vector. However, characterizing the quality of the result of this approximation is still an open problem.

In conclusion, saturation helps the analysis of GSPNs when generating the reachability set, and could in principle be used to obtain $\mathbf{R}$ when a fixpoint iteration is required to traverse the sequences of immediate transitions (i.e., if the GSPN gives rise to *transient vanishing loops* [2]) but, at least so far, it has not been employed for the actual numerical solution, even if the traditional solution algorithms are expressed in terms of a fixpoint computation.

4.4 Using Petri net invariants to derive variable order heuristics

We already mentioned that the order of the variables used to encode a particular function with an appropriate form of decision diagrams can affect the resulting size, sometimes exponentially, but finding an optimal variable order is NP-hard [7]. Of course, the problem is even worse in practice, since the fixpoint symbolic algorithms we described manipulate evolving sets of markings, not just a fixed function. Thus, both *static* and *dynamic* heuristics have been proposed, the former to determine a good variable order a priori, the latter to improve the variable order during decision diagram manipulation.

For static orders, a starting point is often to determine groups of two or more variables that should appear "close" in the overall variable order. An obvious example is the relative order of unprimed and primed variables in the representation of the forward function for a transition α; since x'_k is usually a function of x_k (indeed, the two coincide if x_k is independent of α), the size of the MDD encoding $\mathcal{N}_\alpha$ is usually minimized by an interleaved order, and so is the cost of performing a relational product involving $\mathcal{N}_\alpha$. An interleaved order is then almost universally assumed, and the (much harder) remaining problem is that of choosing the order for the L state variables used to encode sets of marking, which then determines the order of the L unprimed-primed state variable pairs used to encode the forward functions.

For this, heuristics such as "inputs to the same logic gate should appear close the output of the logic gate in the variable order" are often employed when verifying digital circuits. Assuming for now that each variable corresponds to a place, the analogous idea for Petri nets is "places connected to a transition should appear close in the variable order", but the problem is more complex since the graph of a Petri net is often cyclic and rather more connected. In [48], we proposed two heuristics to approach this goal in the context of the saturation algorithm for state-space generation: *sum-of-spans* and *sum-of-tops*.

As the name suggests, sum-of-spans aims at finding a variable order that minimizes $SOS = \sum_{\alpha \in \mathcal{E}} Top(\alpha) - Bot(\alpha)$. This tends to "keep together" the

input, output, and inhibitor places of each transition. However, since places can be connected to multiple transitions, and vice versa, minimizing SOS is shown to be NP-hard in general. Nevertheless, even just orders that result in small values of SOS tend to perform well (and enormously better than random orders).

Sum-of-tops aims instead at finding a variable order that simply minimizes $SOT = \sum_{\alpha \in \mathcal{E}} Top(\alpha)$. Also this heuristic tends to reduce spans but, in addition, it tends to "push down" the range of variables affected by each transition. When generating the state-space using saturation, which works bottom-up, this means that more transitions can be applied sooner (recall that $\mathcal{N}_\alpha$ is applied to the MDD encoding the currently known set of markings only when we begin saturating nodes associated with $Top(\alpha) = v_k$). Minimizing SOT is also NP-hard but, again, small values of SOT tend to do quite well.

Both sum-of-spans and sum-of-tops, however, are after all quite localized heuristics that fail to take into account the entire structure of the Petri net. Indeed, they do not prioritize the grouping of places connected to a particular transition over those connected to another transition, they simply delegate this choice to the algorithm performing the minimization of SOS or SOT.

In [14], we showed how Petri net invariants can help in a heuristic derivation of static orders. Informally, if places a, b, and c form the support of an invariant $\mathbf{w}_a \cdot \mathbf{i}_a + \mathbf{w}_b \cdot \mathbf{i}_b + \mathbf{w}_c \cdot \mathbf{i}_c = const$, it is good to keep the corresponding variables close to each other in the MDD. While in principle one of the three places can be *eliminated* because its number of tokens can be derived from the other two and the invariant equation, this can decrease locality, which is essential for saturation to work well (and for a compact encoding of the forward functions, regardless of the iteration approach being used). Thus, we showed instead how the bottom two places (in the chosen variable order) should be *merged* into the same variable, resulting in an increased locality and a provably smaller MDD. Ultimately, the goal is a mixed heuristic to minimize both SOS or SOT and the invariant support spans (ISS). While much work remains to be done (e.g., an important question is "What relative weights should be used when minimizing SOS+ISS?", since a net can have exponentially more invariants than transitions), we are starting to gain an understanding of the intuition behind static variable order heuristics.

5 Decision diagram implementations and packages

Over the past two decades, many decision diagram libraries have been developed and many logic, timing, and probabilistic verification tools making use of decision diagram technology have been built. In the following, we mention just a few representative libraries and tools. This list is by no means complete, and new libraries and tools are developed every year.

CUDD (`http://vlsi.colorado.edu/~fabio/CUDD/`) [49] is by far the most stable and well-known decision diagram library. CUDD implements BDDs, Algebraic Decision Diagrams (similar to MTBDDs), and Zero-suppressed BDDs (the restriction of ZMDDs to boolean domains). Powerful dynamic variable reordering heuristics are provided.

Task	Best suited class of decision diagrams	
State-space generation	MDDs	
CTL model checking	MDDs	for yes/no answers
	EV^+MDDs	for witness generation
Bounded model checking	MDDs	if using BFS
	EV^+MDDs	if using saturation
P-semiflow computation	ZMDDs	
Timed reachability	MDDs	
Earliest reachability	MDDs and EV^+MDDs	
GSPN analysis	MDDs	for state-space generation
	EV^+MDDs	for state indexing
	EV^*MDDs	for transition rate matrix

Fig. 27. Summary: classes of decision diagrams best suited for various analysis tasks.

Meddly (`http://meddly.sourceforge.net/`) [5] is a recent and still very much under development open-source library that implements most (and likely all, eventually) of the classes of decision diagrams discussed in this paper.

NuSMV (`http://nusmv.fbk.eu/`) [21] is an open-source reimplementation of the SMV tool originally developed by McMillan [33]. It uses both decision diagrams and SAT solvers to carry on logic verification. NuSMV is based on CUDD, thus it only uses the classes of decision diagrams supported by it.

GreatSPN (`http://www.di.unito.it/~greatspn/`) [4] is a well known tool for the analysis of GSPNs and related models. Initially developed with explicit analysis techniques in mind, it evolved to include stochastic well-formed nets (SWNs), which tackle state-space growth through symmetry exploitation. More recently, GreatSPN has been enhanced with decision diagram algorithms implemented using Meddly.

PRISM (`http://www.prismmodelchecker.org/`) [35] is an open-source probabilistic model checker. One of its solution engines uses BDDs and MTBDDs for the fully symbolic or hybrid solution of Markovian models.

CADP (`http://www.inrialpes.fr/vasy/cadp.html`) [27] is a tool suite for the design and analysis of communication protocols. Its capabilities include symbolic verification using BDDs.

SMART (`http://www.cs.ucr.edu/~ciardo/SMART/`) [12] is our own tool, developed at the College of William and Mary, University of California at Riverside, and Iowa State University. Initially conceived for explicit analysis of stochastic models, SMART provides now a wide suite of symbolic capabilities. All the results in this paper were obtained using SMART.

6 Conclusion

We have presented a survey of how decision diagram technology, and in particular the saturation algorithm, can be leveraged to greatly improve the size of Petri net models that can be reasonably studied. Fig. 27 summarizes the classes of decision diagrams best suited for the various analysis tasks we discussed.

We believe that this synergy between Petri nets (or similarly structured formalisms) on the one hand and decision diagrams on the other will continue to provide valuable insight into how to advance our ability to analyze increasingly large and complex discrete-state systems.

Acknowledgments: This work was supported in part by the National Science Foundation under Grant CCF-1018057.

References

1. M. Ajmone Marsan, G. Balbo, and G. Conte. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comp. Syst.*, 2(2):93–122, May 1984.
2. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, 1995.
3. S. Baarir, M. Beccuti, D. Cerotti, M. De Pierro, S. Donatelli, and G. Franceschinis. The GreatSPN tool: recent enhancements. *SIGMETRICS Perform. Eval. Rev.*, 36:4–9, March 2009.
4. J. Babar, M. Beccuti, S. Donatelli, and A. S. Miner. GreatSPN enhanced with decision diagram data structures. In *Proc. ICATPN*, LNCS 6128, pages 308–317. Springer, 2010.
5. J. Babar and A. S. Miner. Meddly: Multi-terminal and Edge-valued Decision Diagram LibrarY. In *Proc. QEST*, pages 195–196. IEEE Computer Society, 2010.
6. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS*, pages 193–207. Springer, 1999.
7. B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. Comp.*, 45(9):993–1002, Sept. 1996.
8. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, Aug. 1986.
9. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.
10. E. Çinlar. *Introduction to Stochastic Processes*. Prentice-Hall, 1975.
11. G. Ciardo. Data representation and efficient solution: a decision diagram approach. In *Formal Methods for Performance Evaluation*, LNCS 4486, pages 371–394. Springer, May 2007.
12. G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with SMART. *Perf. Eval.*, 63:578–608, 2006.
13. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In *Proc. TACAS*, LNCS 2031, pages 328–342. Springer, Apr. 2001.
14. G. Ciardo, G. Lüttgen, and A. J. Yu. Improving static variable orders via invariants. In *Proc. ICATPN*, LNCS 4546, pages 83–103. Springer, June 2007.
15. G. Ciardo, R. Marmorstein, and R. Siminiceanu. The saturation algorithm for symbolic state space exploration. *Software Tools for Technology Transfer*, 8(1):4–25, Feb. 2006.
16. G. Ciardo, G. Mecham, E. Paviot-Adet, and M. Wan. P-semiflow computation with decision diagrams. In *Proc. ICATPN*, LNCS 5606, pages 143–162. Springer, June 2009.

17. G. Ciardo and A. S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In *Proc. PNPM*, pages 22–31. IEEE Comp. Soc. Press, Sept. 1999.
18. G. Ciardo, J. K. Muppala, and K. S. Trivedi. On the solution of GSPN reward models. *Perf. Eval.*, 12(4):237–253, 1991.
19. G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In *Proc. FMCAD*, LNCS 2517, pages 256–273. Springer, Nov. 2002.
20. G. Ciardo and K. S. Trivedi. A decomposition approach for stochastic reward net models. *Perf. Eval.*, 18(1):37–59, 1993.
21. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: an open source tool for symbolic model checking. In *Proc. CAV*, LNCS 2404. Springer, July 2002.
22. E. Clarke, M. Fujita, P. C. McGeer, J. C.-Y. Yang, and X. Zhao. Multi-terminal binary decision diagrams: an efficient data structure for matrix representation. In *IWLS '93 International Workshop on Logic Synthesis*, May 1993.
23. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, LNCS 131, pages 52–71. Springer, 1981.
24. E. M. Clarke, J. M. Wing, R. Alur, R. Cleaveland, D. Dill, A. Emerson, S. Garland, S. German, J. Guttag, A. Hall, T. Henzinger, G. Holzmann, C. Jones, R. Kurshan, N. Leveson, K. McMillan, J. Moore, D. Peled, A. Pnueli, J. Rushby, N. Shankar, J. Sifakis, P. Sistla, B. Steffen, P. Wolper, J. Woodcock, and P. Zave. Formal methods: state of the art and future directions. *ACM Comp. Surv.*, 28(4):626–643, Dec. 1996.
25. S. Donatelli. Superposed generalized stochastic Petri nets: definition and efficient solution. In *Proc. ICATPN*, LNCS 815, pages 258–277. Springer, June 1994.
26. J. Farkas. Theorie der einfachen ungleichungen. *Journal für die reine und andgewandte Mathematik*, 124:1–27, 1902.
27. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. Cadp 2010: A toolbox for the construction and analysis of distributed processes. In *Proc. TACAS*, LNCS 6605, pages 372–387. Springer, 2011.
28. S. Graf, B. Steffen, and G. Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Journal of Formal Aspects of Computing*, 8(5):607–616, 1996.
29. W. K. Grassmann. Finding transient solutions in Markovian event systems through randomization. In *Numerical Solution of Markov Chains*, pages 357–371. Marcel Dekker, Inc., 1991.
30. K. Heljanko. Bounded reachability checking with process semantics. In *CONCUR*, volume 2154 of *LNCS*, pages 218–232, 2001.
31. K. Heljanko and I. Niemelä. Answer set programming and bounded model checking. In *Answer Set Programming*, 2001.
32. A. Itai and M. Rodeh. Symmetry breaking in distributed networks. In *22th Annual Symp. on Foundations of Computer Science*, pages 150–158. IEEE Comp. Soc. Press, Oct. 1981.
33. K. L. McMillan. The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.
34. T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.
35. M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. CAV*, LNCS. Springer, 2011. To appear.

36. M. Z. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: a hybrid approach. In *Proc. TACAS*, LNCS 2619, pages 52–66. Springer, Apr. 2003.
37. Y.-T. Lai, M. Pedram, and B. K. Vrudhula. Formal verification using edge-valued binary decision diagrams. *IEEE Trans. Comp.*, 45:247–255, 1996.
38. K. Lampka and M. Siegle. MTBDD-based activity-local state graph generation. In *Proc. PMCCS*, pages 15–18, Sept. 2003.
39. K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.
40. P. M. Merlin. *A study of the recoverability of computing systems*. PhD thesis, Department of Information and Computer Science, University of California, Irvine, 1974.
41. A. S. Miner. Efficient state space generation of GSPNs using decision diagrams. In *Proc. DSN*, pages 637–646, June 2002.
42. S. Ogata, T. Tsuchiya, and T. Kikuno. SAT-based verification of safe Petri nets. In *Proc. ATVA*, volume 3299 of *LNCS*, pages 79–92, 2004.
43. E. Pastor, O. Roig, J. Cortadella, and R. M. Badia. Petri net analysis using boolean manipulation. In *Proc. ICATPN*, LNCS 815, pages 416–435. Springer, June 1994.
44. J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.
45. C. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, 1962.
46. W. Reisig. *Elements of Distributed Algorithms (Modeling and Analysis with Petri Nets)*. Springer, 1998.
47. O. Roig, J. Cortadella, and E. Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri nets. In *Proc. ICATPN*, LNCS 935, pages 374–391. Springer, June 1995.
48. R. Siminiceanu and G. Ciardo. New metrics for static variable ordering in decision diagrams. In *Proc. TACAS*, LNCS 3920, pages 90–104. Springer, Mar. 2006.
49. F. Somenzi. CUDD: CU Decision Diagram Package, Release 2.4.2. http://vlsi.colorado.edu/∼fabio/CUDD/.
50. W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.
51. M. Tilgner, Y. Takahashi, and G. Ciardo. SNS 1.0: Synchronized Network Solver. In *1st Int. Workshop on Manufacturing and Petri Nets*, pages 215–234, June 1996.
52. A. Valmari. A stubborn attack on the state explosion problem. In *Proc. CAV*, pages 156–165. Springer, June 1991.
53. M. Wan and G. Ciardo. Symbolic reachability analysis of integer timed Petri nets. In *Proc. SOFSEM*, LNCS 5404, pages 595–608. Springer, Feb. 2009.
54. M. Wan, G. Ciardo, and A. S. Miner. Approximate steady-state analysis of large Markov models based on the structure of their decision diagram encoding. *Perf. Eval.*, 68:463–486, 2011.
55. T. Yoneda, H. Hatori, A. Takahara, and S.-I. Minato. BDDs vs. zero-suppressed BDDs: for CTL symbolic model checking of Petri nets. In *Proc. FMCAD*, LNCS 1166, pages 435–449, 1996.
56. A. J. Yu, G. Ciardo, and G. Lüttgen. Decision-diagram-based techniques for bounded reachability checking of asynchronous systems. *Software Tools for Technology Transfer*, 11(2):117–131, Apr. 2009.
57. Y. Zhao and G. Ciardo. Symbolic CTL model checking of asynchronous systems using constrained saturation. In *Proc. ATVA*, LNCS 5799, pages 368–381. Springer, Oct. 2009.
58. W. L. Zuberek. Timed Petri nets definitions, properties, and applications. *Microelectronics and Reliability*, 31:627–644, 1991.