Exploiting interleaving semantics in symbolic state-space generation

Gianfranco Ciardo · Gerald Lüttgen · Andrew S. Miner

Abstract Symbolic techniques based on Binary Decision Diagrams (BDDs) are widely employed for reasoning about temporal properties of hardware circuits and synchronous controllers. However, they often perform poorly when dealing with the huge state spaces underlying systems based on *interleaving semantics*, such as communications protocols and distributed software, which are composed of independently acting subsystems that communicate via shared events.

This article shows that the efficiency of state-space exploration techniques using decision diagrams can be drastically improved by exploiting the interleaving semantics underlying many event-based and component-based system models. A new algorithm for symbolically generating state spaces is presented that (i) encodes a model's state vectors with *Multi–valued Decision Diagrams* (MDDs) rather than flattening them into BDDs and (ii) partitions the model's *Kronecker–consistent* next–state function by event and subsystem, thus enabling multiple lightweight next–state transformations rather than a single heavyweight one. Together, this paves the way for a novel iteration order, called *saturation*, which replaces the breadth–first search order of traditional algorithms. The resulting *saturation algorithm* is implemented in the tool SMART, and experimental studies show that it is often several orders of magnitude better in terms of time efficiency, final memory consumption, and peak memory consumption than existing symbolic algorithms.

Keywords Symbolic state-space exploration · Interleaving semantics · Decision diagrams · Kronecker algebra

G. Ciardo (✉)
Department of Computer Science and Engineering, University of California, Riverside, CA, USA
e-mail: ciardo@cs.ucr.edu

G. Lüttgen
Department of Computer Science, University of York, Heslington, York, UK
e-mail: gerald.luettgen@cs.york.ac.uk

A. S. Miner
Department of Computer Science, Iowa State University, Ames, IA, USA
e-mail: asminer@cs.iastate.edu

1 Introduction

The advent of *Binary Decision Diagrams* (BDDs) [3] has had a massive impact on the practicality and adoption of state–based automated verification. It increased the manageable sizes of state spaces from about 10^7 states, with traditional explicit state-space generation techniques, to about 10^{20} states [4]. Today BDD–based symbolic model checkers [15] are able to automatically verify temporal properties of complex hardware circuits and synchronous controllers. However, they often perform poorly on system models that employ interleaving semantics, such as models of communications protocols and distributed software, which often suffer from state-space explosion. It is a widely held belief that decision diagrams are the wrong choice of data structure when generating, storing, and manipulating the huge state spaces underlying interleaving–based system behavior [6]. Indeed, the most popular model checker used on such systems is the explicit–state model checker *Spin* [23] which relies on partial–order reduction techniques to limit state-space explosion.

The aim of this article is to show that the above–stated belief is wrong and that symbolic techniques may well be adapted to cope with the intrinsic complexities of system models based on interleaving. We restrict ourselves to *state-space generation* which is the most fundamental challenge for many formal verification tools, such as model checkers [16]. Traditional symbolic state-space generators store both the set of *initial states* and the global *next–state function*, which together define a system's state space, as BDDs. The BDD–representation of the desired reachable state space is then computed by iteratively applying the BDD–encoded next–state function to the BDD representing the initial states, until a fixed point is reached. In contrast to explicit state-space generators, whose memory requirements increase linearly with the number of explored states, such a symbolic breadth–first–search algorithm sees the BDD storing the reachable state space grow and shrink during execution. In practice, the *peak* BDD size is achieved well before reaching its final size and is frequently so large that it cannot be stored in the main memory of a modern workstation.

1.1 Contributions

Our main contribution is a novel symbolic state-space generation algorithm that exploits the interleaving semantics in event-based concurrent systems and is orders of magnitude more time– and memory–efficient than traditional symbolic algorithms. The effect of firing events in such system models is *local*, a fact that is largely ignored by traditional symbolic state-space exploration tools, with the exception of the disjunctive partitioning of next–state functions [5].

System models equipped with interleaving semantics have both structured states and a structured next–state function. States are described by vectors over finite sets, where each vector entry represents the state of a subsystem in the system under consideration. These subsystems either arise naturally when, e.g., considering models of distributed software or are the result of partitioning a system model that is given in one piece, such as a *Petri net* [30]. Sets of state vectors, i.e., structured sets of states, can be represented naturally by *Multi–valued Decision Diagrams* [24] (MDDs). This is a variant of Binary Decision Diagrams (BDDs) that facilitates the encoding of functions over finite–set variables rather than boolean variables.

Similarly, the next–state function of an event-based system must not be treated as one monolithic function, but can be disjunctively partitioned by event and, in product–form style, by subsystem. The latter is due to the *Kronecker representation* [33] inherently possible in interleaving–based semantics, such as the standard semantics of Petri nets. This means that

firing an event usually updates just a few components of a system's state vector, which permits the use of multiple lightweight next–state transformations that manipulate MDDs locally, rather than a single heavyweight one that manipulates BDDs globally. This results in significant time savings when symbolically exploring state spaces.

The partitioning of the next–state function by events also implies that the reachable state space can be built by firing the system's events in any order, as long as every event is considered often enough. This is in contrast to statements that symbolic state-space generation is "inherently breadth–first" [1]. We exploit the freedom of firing order in our setting by proposing a novel iteration strategy that exhaustively fires all events affecting a given MDD node, thereby transforming it into its final *saturated* shape. Moreover, nodes are considered in depth-first fashion, i.e., whenever a node is processed, all its descendants are already saturated. *Saturation* implies that MDD nodes are updated as soon as safely possible, whereas traditional symbolic techniques constantly generate new BDD nodes and disconnect others [27]. This is important since non–saturated nodes are guaranteed *not* to be part of the final state space, while saturated nodes have a good chance to be part of it. Moreover, the data structures we use to implement MDDs and their operations contain only saturated nodes and are not cluttered with nodes that will later become superfluous. This significantly reduces the peak number of MDD nodes and cache entries required during state-space generation. In addition, the resulting state-space generation algorithm is not only concise, but also allows for an elegant proof of correctness.

1.2 Results

We have implemented our new Saturation algorithm in the tool SMART [8], and experimental studies indicate that it performs on average several orders of magnitude faster than state-space generation in the modern symbolic model checkers Cadence SMV and NuSMV [15], for the class of event-based concurrent system models that rely on interleaving semantics. The considered examples range from the classic problem of dining philosophers to mutual–exclusion protocols, a flexible manufacturing system, and a fault–tolerant multiprocessor system, all of which are taken from the rich literature on state-space exploration [22, 29, 32, 36]. Even more important, and in contrast to related work, the peak memory requirements of the saturation algorithm are often close to its final memory requirements.

Further experiments for which we have replaced the Saturation order with a breadth–first search strategy in our implementation, testify that the impressive performance improvements of Saturation are largely a result of our novel iteration order. In comparison, the use of MDDs and the Kronecker property contribute little to the efficiency improvements, but significantly simplify the presentation of the Saturation algorithm.

Thus, Saturation enables the symbolic verification of larger concurrent systems than ever before, using much less memory and providing faster feedback. Recent work [12], which has applied our Saturation algorithm to implementing a novel symbolic model checker for the temporal logic CTL, further testifies to this statement.

1.3 Organization

The remainder of this article is organized as follows. Section 2 revisits the classical setting of symbolic state-space generation. Section 3 briefly summarizes the modeling formalism of Petri nets, which we have chosen for presenting our work, and introduces a running example. Our approach to exploiting the structure of concurrent systems models with MDD and Kronecker encodings is detailed in Section 4, while Section 5 presents our novel Saturation

algorithm, illustrates it by means of the running example, and proves it correct. The results of our experimental studies are reported in Section 6, related work is discussed in Sections 7 and 8 gives our conclusions and directions for future research.

2　Traditional symbolic state-space generation

Many real–world systems, including digital circuits and software, may be specified as discrete–state models. A state may indicate, for example, which registers of a digital circuit are currently set or which value each variable of a program currently possesses.

Definition 2.1. A discrete–state model is a tuple $(\hat{\mathcal{S}}, \mathcal{N}, \mathcal{S}^{\text{init}})$, where

- $\hat{\mathcal{S}}$ is the *potential state space*, i.e., a set that includes all states **i** that the underlying system may potentially enter;
- $\mathcal{N}: \hat{\mathcal{S}} \to 2^{\hat{\mathcal{S}}}$ is the *next–state function*, i.e., $\mathcal{N}(\mathbf{i})$ specifies the set of states that can be reached from state **i** in one step;
- $\mathcal{S}^{\text{init}} \subseteq \hat{\mathcal{S}}$ is the set of *initial states*.

Intuitively, a discrete–state model of a system is simply a directed graph, in which traversals through the graph correspond to system runs. In practice, discrete–state models are not directly given as graphs, but are extracted from structured high–level system descriptions, e.g., encoded in VHDL or modeled as Petri nets, which often represent an enormous potential state space and a complex next–state function. Typically, a system modeled in a high–level formalism is composed of *K subsystems*, i.e., $\hat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$, in such a way that a *(global) state* **i** can be written as a K-tuple $(i_K, \ldots, i_1)$ of *local states*, each one corresponding to a different subsystem. For example, in a digital circuit, each register may define a subsystem of its own; or, alternatively, several registers may be grouped together. As another example, in distributed software, each thread may be considered as a subsystem. The reason for numbering subsystems backwards from K to 1 instead of forwards will become apparent in Section 4.2.1.

2.1　Computing reachable state spaces

state-space generation consists of finding out which states of a discrete–state model's potential state space are *reachable* from its initial states. The set of its reachable states $\mathcal{S} \subseteq \hat{\mathcal{S}}$ is called the model's *state space*. Computing this state space is conceptually a simple task: one only has to consider the graph corresponding to the model's next–state function and search it starting from each initial state. *Explicit state-space generators* explore this graph state–by–state, thus their time and space complexity is linear in the number of reachable states. The challenge of state-space generation is to handle real–world systems with enormous state spaces; indeed, explicit approaches usually become infeasible for systems with more than about 10^7 states. Systems based on interleaving semantics may have state-space sizes that are exponential in the number of subsystems. Thus, states cannot be practically handled and stored one–by–one but are better processed in chunks of large sets. To achieve this, researchers typically apply a breadth–first algorithm that computes a model's state space as the reflexive and transitive closure of the model's next–state function, starting from the
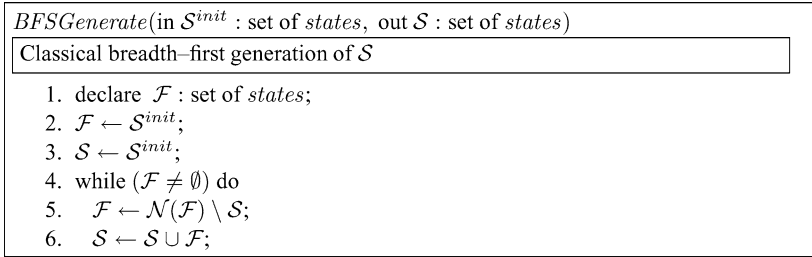
$BFSGenerate(\text{in } \mathcal{S}^{init} : \text{set of } states, \text{ out } \mathcal{S} : \text{set of } states)$

Classical breadth–first generation of $\mathcal{S}$

    1. declare $\mathcal{F}$: set of $states$;
    2. $\mathcal{F} \leftarrow \mathcal{S}^{init}$;
    3. $\mathcal{S} \leftarrow \mathcal{S}^{init}$;
    4. while $(\mathcal{F} \neq \emptyset)$ do
    5. $\mathcal{F} \leftarrow \mathcal{N}(\mathcal{F}) \setminus \mathcal{S}$;
    6. $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{F}$;

Fig. 1 A classical symbolic algorithm for state-space generation

model's initial states:

$$\mathcal{S} = \mathcal{S}^{\text{init}} \cup \mathcal{N}(\mathcal{S}^{\text{init}}) \cup \mathcal{N}^2(\mathcal{S}^{\text{init}}) \cup \cdots = \mathcal{N}^*(\mathcal{S}^{\text{init}}),$$

where we have extended $\mathcal{N}$ to accept sets of states as argument, i.e., $\mathcal{N}(\mathcal{X}) = \bigcup_{\mathbf{i} \in \mathcal{X}} \mathcal{N}(\mathbf{i})$.

This leads to the algorithm *BFSGenerate* of Fig. 1, which performs a breadth–first search and manipulates *sets* of states, rather than individual states. The number of required steps, or iterations, is equal to the the maximum of the (shortest) distances of any reachable state from $\mathcal{S}^{\text{init}}$. To make this algorithm work in practice, one must be able to represent very large sets of states and the next–state function compactly in the memory of a workstation, as well as to efficiently perform union operations and apply the next–state function directly on these representations. In this context, *BFSGenerate* is referred to as a *symbolic*, or *implicit*, state-space generation algorithm [27].

2.2 Binary Decision Diagrams

A popular technique for implicitly representing sets of states is to use *Binary Decision Diagrams* (BDDs) [3]. Informally, BDDs are graph structures that represent boolean functions over boolean variables. While the BDD size can be exponential in the number of variables in the worst case [3], BDDs are an extremely compact representation in many practical cases. Moreover, given a fixed order on the variables, BDD representations are canonical.

To use BDDs for storing state spaces, each state must be expressible via b boolean variables $(x_b, \ldots, x_1)$, for some $b \in \mathbb{N}$. Then, a set of states $\mathcal{X}$ can be represented as a BDD via its characteristic function $f_{\mathcal{X}}$, i.e., $f_{\mathcal{X}}(\mathbf{i}) = 1$ if and only if $\mathbf{i} \in \mathcal{X}$. One approach is to encode a potential state $\mathbf{i} \in \hat{\mathcal{S}}$ by assuming that $|\hat{\mathcal{S}}| = 2^b$ for some $b \in \mathbb{N}$ [16]. Another approach, applicable to structured models, is to represent local states using a "one–hot" encoding with $n_l = |\mathcal{S}_l|$ boolean variables, only one of which possesses value 1 at any given time [32]. Set operations such as union and intersection can be performed via logical operations on the arguments' characteristic functions; for example, union on sets corresponds to disjunction on BDDs. The complexity of these operations depends on the number of nodes in the arguments' BDDs and not on the number of states encoded by the BDDs. For union and intersection, the complexity is proportional to the product of the arguments' BDD sizes, while checks for emptiness can be made in constant time due to the canonicity of BDDs.

The next–state function of a discrete–state model may be represented by a function $f_{\mathcal{N}}$ over $2b$ variables $(x_b, \ldots, x_1, x'_b, \ldots, x'_1)$, where $f_{\mathcal{N}}$ evaluates to 1 if and only if state $(x'_b, \ldots, x'_1)$ can be reached from state $(x_b, \ldots, x_1)$ in exactly one step. Since the chosen ordering of variables can have a significant impact on the number of nodes in the resulting

BDD, $f_\mathcal{N}$ is typically represented using the variable order $(x_b, x'_b, \ldots, x_1, x'_1)$, which usually produces a more compact BDD than the naive order $(x_b, \ldots, x_1, x'_b, \ldots, x'_1)$. Applying $\mathcal{N}$ to a set of states $\mathcal{X}$ encoded by BDD $f_\mathcal{X}$—a process known as *image computation*—is typically implemented on BDDs as $\mathcal{N}(\mathcal{X}) = \exists\{x_b, \ldots, x_1\}(f_\mathcal{X} \wedge f_\mathcal{N})$, where $f_\mathcal{X} \wedge f_\mathcal{N}$ considers the next–state function for only the states in $\mathcal{X}$, and $\exists\{x_b, \ldots, x_1\}$ determines all possible outcomes of $f_\mathcal{X} \wedge f_\mathcal{N}$. Note that the resulting BDD is over variables $(x'_b, \ldots, x'_1)$; these must be converted back to variables $(x_b, \ldots, x_1)$ at the end of each image computation.

2.3 Problems of symbolic state-space generation

Applying algorithm *BFSGenerate* with BDD–encoded sets of states often results in significant time and memory savings for state-space generation. Indeed, the advent of BDDs increased the manageable sizes of state spaces to about 10^{20} states [4], particularly for synchronous hardware models. Nevertheless, many challenges remain.

One particular problem is that, although the final state space might have a very compact BDD representation, the intermediate sets of states built during the execution of algorithm *BFSGenerate* might not, and thus might not fit into a workstation's memory. In other words, it is the *peak*, not the *final* BDD size that limits the applicability of BDD-based methods, and this peak size is almost always reached well before the algorithm terminates. This problem is made even worse by the way the breadth–first iterations operate on the BDD nodes: whenever $\mathcal{S}$ is updated, new BDD nodes are generated as a result of the union operation, rather than modifying existing ones. One way to reduce the peak size is to reorganize BDDs by changing their variable order on–the–fly [20], hoping to achieve a more compact representation.

Another problem is that many next–state functions underlying real–world systems are very complex and frequently cannot be compactly stored as BDDs. Widely employed techniques to reduce the size of the BDD representing a system's next–state function are *conjunctive* or *disjunctive partitioning* [5]. For *synchronous system models* with K submodels, the next–state function can be expressed as the conjunction

$$\mathcal{N}(\mathbf{i}) = \mathcal{N}_K(\mathbf{i}) \times \cdots \times \mathcal{N}_1(\mathbf{i}),$$

where the local next–state function $\mathcal{N}_l(\mathbf{i})$ describes the change in local state i_l for global state $\mathbf{i}$. For many systems, the BDDs representing functions $\mathcal{N}_l$ in conjunctive form are quite compact. Image computation for a conjunctive form can be performed by applying one BDD at a time, thus avoiding the construction of the monolithic and potentially large BDD for $\mathcal{N}$. Experimental studies have shown that this approach can increase the size of manageable state spaces by about one order of magnitude [5].

Similarly, for *asynchronous system models*, where asynchrony is exemplified by interleaving and where only a single submodel changes its local state at a time, the next–state function can be expressed as the disjunction

$$\mathcal{N}(\mathbf{i}) = \bigcup_{l=1}^{K} \mathcal{I}_K \times \cdots \mathcal{I}_{l+1} \times \mathcal{N}_l(\mathbf{i}) \times \mathcal{I}_{l-1} \times \cdots \mathcal{I}_1,$$

where function $\mathcal{N}_l(\mathbf{i})$ is as before, while $\mathcal{I}_g$, for $g \neq l$ indicates that the local state of the gth submodel remains unchanged. Indeed, many concurrent system models, such as event-based system models specified in process algebras or Petri nets, rely on synchronizations between an often small subset of the submodels. This *event locality* requires a more general

disjunctive–partitioning approach than the one shown above or than minor improvements (see, e.g., p. 80 of [16]) to the above approach.

3 Petri nets

While our results and our novel algorithm for symbolic state-space generation which we will present in Sections 4–6 are applicable to general discrete–state models of event-based concurrent systems that rely on interleaving semantics, such as models of communications protocols and distributed software, we focus our discussion on an extended class of *Petri nets* [30]. This is because the Petri net formalism is well known and widely used, it is easy to define, and employs interleaving semantics. In addition, with the appropriate extensions, arbitrarily complex discrete–state concurrent systems can be modeled in Petri nets. In case the state space of the underlying system model is not finite, our state-space generation algorithm will not terminate; this is the same as for most other state-space generators. However, in special cases, but not in general, it can be decided whether a Petri net with the extensions we use in our study, has a finite state space [30].

Definition 3.1. A *Petri net* $(\mathcal{P}, \mathcal{T}, f, s)$ is a directed bipartite multigraph with two finite sets of nodes: *places* $\mathcal{P}$ and *transitions* $\mathcal{T}$, such that $f: (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P}) \to \mathbb{N}$ defines the *cardinality* of *input arcs* (from places to transitions) and *output arcs* (from transition to places), while $s \in \mathbb{N}^{|\mathcal{P}|}$ defines an initial *marking*, i.e., an assignment of *tokens* to each place. The net evolves according to two rules, the *enabling rule* and the *firing rule*. The enabling rule states that a transition t is enabled in marking $\mathbf{i}$ if, for all places $p \in \mathcal{P}$, $i_p \geq f(p, t)$. The firing rule states that an enabled transition t in marking $\mathbf{i}$ may fire, leading to a marking $\mathbf{j}$ given by $j_p = i_p - f(p, t) + f(t, p)$, for all places $p \in \mathcal{P}$.

It is easy to see that a Petri net defines a discrete–state model where the potential state space is the set of potential markings $\mathbb{N}^{|\mathcal{P}|}$ and the next–state function is determined by the enabling and firing rules for f.

3.1 Extended Petri nets

Various extensions have been proposed to the standard Petri net formalism [30].

Definition 3.2. A Petri net with *inhibitor* arcs specifies, in addition, a function $h: \mathcal{P} \times \mathcal{T} \to \mathbb{N} \cup \{\infty\}$, and its enabling rule is modified as follows: t is enabled in marking $\mathbf{i}$ if, for all places $p \in \mathcal{P}$, $i_p \geq f(p, t)$ and $i_p < h(p, t)$.

Definition 3.3. A Petri net with *reset* arcs specifies, in addition, a function $r: \mathcal{P} \times \mathcal{T} \to \{0, 1\}$, and its firing rule is modified as follows: an enabled transition t in marking $\mathbf{i}$ may fire, leading to a marking $\mathbf{j}$ given by $j_p = r(p, t)(i_p - f(p, t)) + f(t, p)$, for all places $p \in \mathcal{P}$.

Definition 3.4. A *self–modifying* net allows the arc cardinalities to be a function of the current marking, $f: ((\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})) \times \mathbb{N}^{|\mathcal{P}|} \to \mathbb{N}$, and modifies the enabling and firing rules accordingly: a transition t is enabled in marking $\mathbf{i}$ if, for all places $p \in \mathcal{P}$, $i_p \geq f(p, t)(\mathbf{i})$ and, if enabled, it can fire, leading to marking $\mathbf{j}$ that satisfies $j_p = i_p - f(p, t)(\mathbf{i}) + f(t, p)(\mathbf{i})$, for all places $p \in \mathcal{P}$.

In other words, an inhibitor arc with cardinality c from p to t disables t whenever p contains c or more tokens; a reset arc from p to t empties p immediately before adding the tokens specified by the output arc from t to p, if any, when t fires; and, in self–modifying nets, the effect of an input or output arc is not constant, but depends instead on the entire marking.

Petri net formalisms can be classified according to the class of languages they can generate if transitions are labeled with events α from some event alphabet $\mathcal{E}$, which are emitted when a transition fires. Petri nets with reset arcs are known to be strictly more expressive than standard Petri nets, while Petri nets with inhibitor arcs and self–modifying Petri nets are Turing equivalent. In this paper we restrict ourselves to Petri nets defining finite state spaces, i.e., Petri nets with a finite number of reachable markings. In this case, the above extensions can simply be seen as a way to define more *compact* models, in the sense that model sizes are small.

3.2 Structuring Petri nets

Although Petri nets appear to be rather monolithic models, they offer a rich and flexible structure, regarding both potential state spaces and next–state functions.

To represent the potential state space $\hat{\mathcal{S}}$ of a Petri net as a cross–product $\mathcal{S}_K \times \cdots \times \mathcal{S}_1$, one may partition the net's places into K disjoint sets. For example, given a Petri net where each place is considered to be a submodel, $\mathcal{S}_l$ is the set of possible values that the number of tokens in place p_l can assume. For convenience, we may think of $\mathcal{S}_l$ as the set $\{0, 1, \ldots, m_l\}$, where m_l is the maximum number of tokens place p_l might have, even if some number $j_l < m_l$ might not occur, as no marking $\mathbf{i} \in \mathcal{S}$ with $i_l = j_l$ exists. Knowing a priori the value of m_l or even just an upper bound on it, e.g., through the computation of *place invariants*, can be expensive, and it is undecidable if inhibitor arcs are present [30]. Furthermore, if multiple places are grouped into the same submodel, $\mathcal{S}_l$ may be defined as the cross–product of the possible values for the number of tokens in those places but, again, many combinations of these values might not occur. Note that using a finite but larger–than–needed set $\mathcal{S}_l$ does not affect the correctness of the state-space generation algorithms considered here, but it may make them less efficient.

Determining the smallest possible sets $\mathcal{S}_l$ is a problem of practical interest and depends on the underlying modeling language, here Petri nets. In this article, we simply assume that $\mathcal{S}_l$ is finite and known; hence, given its size n_l, we can map its elements to $\{0, \ldots, n_l - 1\}$. In our experiments in Section 6 we derived the exact bounds of the submodels of the chosen Petri nets by considering place invariants. However, computing a priori the sets $\mathcal{S}_l$ is, in practice, not necessary for computing the underlying model's overall state space. Both computations can be interleaved, i.e., submarkings, or local state spaces, can be computed on–the–fly, as demonstrated in [11].

Regarding the structure of the next–state function $\mathcal{N}$ encoded by a Petri net, it is easy to see that $\mathcal{N}$ can be represented as the union of one next–state function per event $\alpha \in \mathcal{E}$; formally, $\mathcal{N}(\mathbf{i}) = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha(\mathbf{i})$. We then have that α is *enabled* in state $\mathbf{i}$ if $\mathcal{N}_\alpha(\mathbf{i}) \neq \emptyset$, and that it is *disabled* otherwise. Moreover, if $\mathbf{j} \in \mathcal{N}_\alpha(\mathbf{i})$, we say that the Petri net can progress from state $\mathbf{i}$ to state $\mathbf{j}$ when event, i.e., transition, α fires. As the effect of firing a Petri net transition is deterministic, $\mathcal{N}_\alpha(\mathbf{i})$ can contain at most one state. This union representation of $\mathcal{N}$ is a form of disjunctive partitioning. However, there is more structure to the next–state function of a Petri net; since this is less obvious, we defer a further discussion to Section 4.1.
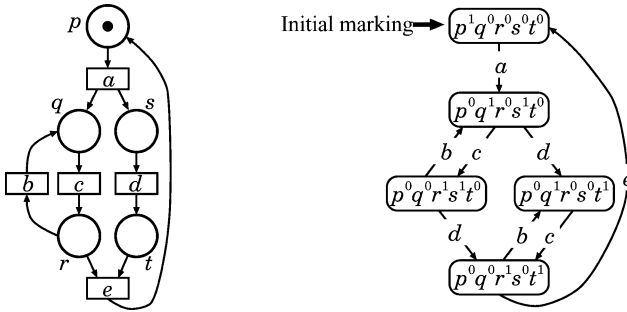
Fig. 2 The Petri net of our running example (left) and its discrete–state model (right)

3.3 Running example

We use the simple Petri net shown in Fig. 2 as a running example to illustrate our techniques. This model is often studied in the literature, as it contains important behaviors that can be modeled with Petri nets: *sequentialization*, *concurrency*, *conflict*, and *fork–and–join*.

The events of the model are the *transitions* of the Petri net: $\mathcal{E} = \{a, b, c, d, e\}$. If the initial marking is $(p^1 q^0 r^0 s^0 t^0)$, i.e., there is one token in place p and zero tokens in places q, r, s, and t, the state space $\mathcal{S}$ contains only five states, as shown in Fig. 2 on the right. However, the size of $\mathcal{S}$ grows with the cube of the number of tokens initially in place p: if the initial marking is $(p^N q^0 r^0 s^0 t^0)$, then $\mathcal{S}$ has $(N + 1)(N + 2)(2N + 3)/6$ markings.

The structure and size of the potential state space depend on how we define the submodels for this net. In our approach, a Petri net is decomposed into submodels by partitioning its places. If we assign each place to a different class of the partition, i.e., we use the partition $(\{p\}, \{q\}, \{r\}, \{s\}, \{t\})$, we have $K = 5$ submodels. Then it is easy to see that $\mathcal{S}_5 = \{(p^0), (p^1), \ldots, (p^N)\}$, $\mathcal{S}_4 = \{(q^0), (q^1), \ldots, (q^N)\}$, and so on, thus $\hat{\mathcal{S}}$ has $(N + 1)^5$ states. If we use instead the partition $(\{p\}, \{q, r\}, \{s\}, \{t\})$, we have $K = 4$ submodels, where, for $k \in \{4, 2, 1\}$, the set $\mathcal{S}_k$ still has $N + 1$ states, but $\mathcal{S}_3$ has instead $(N + 1)(N + 2)/2$ states, corresponding to all the ways to put a total of up to N tokens in places q and r, namely $\{(q^0 r^0), (q^1 r^0), (q^0 r^1), (q^2 r^0), (q^1 r^1), (q^0 r^2), \ldots, (q^0 r^N)\}$. With this second partition, the potential state space $\hat{\mathcal{S}}$ is smaller, as it contains only $(N + 1)^4 (N + 2)/2$ states. Of course, the state space $\mathcal{S}$ is the same, regardless of how we define $\hat{\mathcal{S}}$.

4 Exploiting interleaving semantics in concurrent system models

To benefit from the interleaving semantics in event-based concurrent system models for symbolic state-space generation, we depart from the classical encoding of next–state functions and state spaces as BDDs. Instead, we propose to employ *Kronecker expressions on sparse boolean matrices* to store next–state functions and *Multi–valued Decision Diagrams* (MDDs) to store state spaces. This allows us to exploit the event locality implied by interleaving semantics and forms the foundation for the symbolic state-space generation algorithm we present in Section 5.

4.1 Sparse boolean matrices to store next–state functions

In contrast to the wealth of related work starting with McMillan's thesis [27], we do not represent the next–state function of an event-based concurrent system model with interleaving

semantics via decision diagrams. Instead we split the next–state function by event α and level l to obtain a nested disjunctive–conjunctive form, which can be represented as a collection of (at most) $K \cdot |\mathcal{E}|$ sparse matrices, where $\mathcal{E}$ is the set of considered events.

This splitting is inspired by techniques to compactly represent the transition rate matrix of a continuous–time Markov chain by means of a sum of *Kronecker products* [33]. Of course, since our next–state functions are concerned only with which states can be reached, and not with the *rate* at which they can be reached, we use boolean matrices instead of real–valued matrices. The application of Kronecker–based encodings to state-space generation has been proposed by Buchholz and Kemper in [25], but for *explicit* rather than symbolic state-space generation. In this section we briefly review Kronecker products and show that many next–state functions, including those defined by Petri nets, permit a *practical* Kronecker–based encoding.

4.1.1 Kronecker products, consistency, and representation

We start from a given next–state function $\mathcal{N}$ of an event-based concurrent system model with interleaving semantics, with $\hat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$ and $n_l = |\mathcal{S}_l|$, which is decomposed by event, i.e., $\mathcal{N} = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha$. In a Kronecker–based encoding scheme, $\mathcal{N}_\alpha$ is represented by a square boolean matrix $\mathbf{N}_\alpha$ of size $|\hat{\mathcal{S}}|$, where $\mathbf{N}_\alpha[\mathbf{i}, \mathbf{j}] = 1$ if and only if $\mathbf{j} \in \mathcal{N}_\alpha(\mathbf{i})$ and states $\mathbf{i}$ and $\mathbf{j}$, when used as global state indices, are interpreted as mixed–base natural numbers $\mathbf{i} = \sum_{K \geq l \geq 1} i_l \cdot \prod_{g=1}^{l-1} n_g$, and $\mathbf{j} = \sum_{K \geq l \geq 1} j_l \cdot \prod_{g=1}^{l-1} n_g$, respectively. However, matrix $\mathbf{N}_\alpha$ is not stored explicitly but as the *Kronecker product* $\mathbf{N}_\alpha = \mathbf{N}_{K,\alpha} \otimes \cdots \otimes \mathbf{N}_{1,\alpha}$, where each square matrix $\mathbf{N}_{l,\alpha}$ has dimension n_l. Such a representation always exists for generalized Kronecker products [19], but this may require functional elements in the matrices $\mathbf{N}_{l,\alpha}$. To eliminate the need for functional elements and use ordinary Kronecker products, we must ensure that the representation of $\mathbf{N}_{K,\alpha} \otimes \cdots \otimes \mathbf{N}_{1,\alpha}$ is possible using matrices $\mathbf{N}_{l,\alpha}$ having constant boolean entries.

Definition 4.1. An event α has a *Kronecker representation* for a given model partition if its next–state function $\mathcal{N}_\alpha$ can be written as the cross–product of K local functions, i.e., $\mathcal{N}_\alpha = \mathcal{N}_{K,\alpha} \times \cdots \times \mathcal{N}_{1,\alpha}$, where $\mathcal{N}_{l,\alpha} \colon \mathcal{S}_l \to 2^{\mathcal{S}_l}$, for $K \geq l \geq 1$. Moreover, a partition of a given system model into submodels is *Kronecker–consistent* if every event α has a Kronecker representation for that partition.

A Kronecker representation guarantees that the entries of each matrix $\mathbf{N}_{l,\alpha}$ are boolean constants, since $\mathbf{N}_{l,\alpha}[i_l, j_l] = 1$ if and only if $j_l \in \mathcal{N}_{l,\alpha}(i_l)$. Indeed, the matrix $\mathbf{N}_{l,\alpha}$ can be seen as a way to represent the local next–state function $\mathcal{N}_{l,\alpha}$.

Intuitively, a Kronecker representation reflects the interleaving semantics of event-based concurrent systems. For an event to be enabled, it must not be disabled by any subsystem: $\mathcal{N}_{l,\alpha}(i_l) = \emptyset$ implies $\mathcal{N}_\alpha(\mathbf{i}) = \emptyset$, for any global state $\mathbf{i}$ whose lth component is i_l. Moreover, the outcome of firing an event is decided by each subsystem independently: if $\mathcal{N}_\alpha(\mathbf{i})$ contains both $(j_K, \ldots, j_1)$ and $(j'_K, \ldots, j'_1)$, then it must contain all states of the form $(j''_K, \ldots, j''_1)$, where each j''_l can be either j_l or j'_l.

An important observation given a Kronecker–consistent partition is that, if the state of submodel l does not affect the enabling of event α and if the firing of α does not change the state of subsystem l, then $\mathcal{N}_{l,\alpha}(i_l) = \{i_l\}$ for all $i_l \in \mathcal{S}_l$, i.e., $\mathbf{N}_{l,\alpha} = \mathbf{I}$, the identity matrix of dimension n_l. Since this property features prominently in those concurrent system models that are equipped with interleaving semantics, we introduce the following notational conventions. We say that event α *depends* on level l if $\mathbf{N}_{l,\alpha} \neq \mathbf{I}$, i.e., if the local state at level l affects the

enabling of α or if it is changed by the firing of α. Let $Top(\alpha)$ and $Bot(\alpha)$ be the highest and lowest levels, respectively, on which α depends. An event α such that $Top(\alpha) = Bot(\alpha) = l$ is said to be a *local event* for level l.

4.1.2 Existence of Kronecker representations

The aim of this section is to show that our Kronecker–consistency requirement is not restrictive in practice. It is in fact quite natural for concurrent system models; indeed, it is automatically satisfied by many modeling formalisms, such as by Petri nets, even in the presence of inhibitor and reset arcs. For other models, such as self–modifying Petri nets, Kronecker consistency can always be achieved, albeit at the price of introducing additional system events or combining subsystems. To substantiate these statements we establish some theoretical properties of Kronecker representations.

Lemma 4.1. *An event that is enabled in exactly one state and whose firing leads to exactly one state has a Kronecker representation in any partition.*

Proof: If $\mathbf{i}$ is the only state in which event α is enabled, and $\mathcal{N}_\alpha(\mathbf{i}) = \{\mathbf{j}\}$, then we have $\mathcal{N}_{l,\alpha}(i_l) = \{j_l\}$ and $\mathcal{N}_{l,\alpha}(i_l') = \emptyset$ for any submodel l and any local state $i_l' \neq i_l$. $\qquad\square$

Another way to achieve Kronecker consistency is to coarsen the considered partition by merging submodels. In the extreme case where all submodels are merged, the resulting partition is trivially a Kronecker representation but no longer exhibits any structure.

Lemma 4.2. *An event α with a Kronecker representation for a given partition also has a Kronecker representation for any* coarser *partition, i.e., a partition obtained by merging any number of submodels.*

For the proof of this lemma and the next we use the shorthand $\mathcal{N}_{l:g,\alpha}$ to represent the effect of α on substates, i.e., $\mathcal{N}_{l:g,\alpha}((i_l, \ldots, i_g)) = \mathcal{N}_{l,\alpha}(i_l) \times \cdots \times \mathcal{N}_{g,\alpha}(i_g)$, for $K \geq l \geq g \geq 1$.

Proof: Since the ordering of submodels does not determine whether an event has a Kronecker representation, it suffices to show that the theorem holds when the topmost submodels from K down to $K' < K$ are merged. Given that event α has a Kronecker representation, we have $\mathcal{N}_\alpha = \mathcal{N}_{K:1,\alpha} = \mathcal{N}_{K:K',\alpha} \times \mathcal{N}_{(K'-1):1,\alpha}$. Thus, in the coarsened partition with K' levels, where level K' corresponds to the old levels $K, K-1, \ldots, K'$, the local next–state function for submodel K' is $\mathcal{N}_{K:K',\alpha}$. $\qquad\square$

As an example, consider some model that is partitioned into four submodels, where every event has a Kronecker representation except for α, i.e., $\mathcal{N}_\alpha = \mathcal{N}_{4,\alpha} \times \mathcal{N}_{(3,2),\alpha} \times \mathcal{N}_{1,\alpha}$, but $\mathcal{N}_{(3,2),\alpha} : \mathcal{S}_3 \times \mathcal{S}_2 \to 2^{\mathcal{S}_3 \times \mathcal{S}_2}$ cannot be expressed as the product $\mathcal{N}_{3,\alpha} \times \mathcal{N}_{2,\alpha}$. We can achieve Kronecker consistency by either partitioning the model into three submodels—since, then, α would now have a Kronecker representation and Lemma 4.2 ensures that every other event would retain a Kronecker representation—or by replacing event α with a set of events $\{\alpha_{i,j} : i \equiv (i_3, i_2) \in \mathcal{S}_3 \times \mathcal{S}_2 \ \wedge \ j \equiv (j_3, j_2) \in \mathcal{N}_{(3,2),\alpha}(i)\}$—so that $\mathcal{N}_{3,\alpha_{i,j}}(i_3) = \{j_3\}$ and $\mathcal{N}_{2,\alpha_{i,j}}(i_2) = \{j_2\}$. In the worst case α can be enabled in each combination of local states. For a formalism such as Petri nets, where the effect of α is deterministic, this splits α into

$n_3 \cdot n_2$ events. However, it may be possible to merge some of these events using the following lemma.

Lemma 4.3. *For a given partition, two events α_1 and α_2 with Kronecker representations can be merged into a single event α with a Kronecker representation, if the local next–state functions for α_1 and α_2 differ in at most one submodel. The local functions for the merged event α are given by the union of the local functions for events α_1 and α_2.*

Proof: Suppose we have a partition of K submodels and $\mathcal{N}_{g,\alpha_1} = \mathcal{N}_{g,\alpha_2}$ for all $g \neq l$, for some l with $K \geq l \geq 1$. Then we have $\mathcal{N}_\alpha = \mathcal{N}_{\alpha_1} \cup \mathcal{N}_{\alpha_2}$ and

$$
\begin{aligned}
&\mathcal{N}_{\alpha_1} \cup \mathcal{N}_{\alpha_2} \\
&= \left(\mathcal{N}_{K:(l+1),\alpha_1} \times \mathcal{N}_{l,\alpha_1} \times \mathcal{N}_{(l-1):1,\alpha_1}\right) \cup \left(\mathcal{N}_{K:(l+1),\alpha_2} \times \mathcal{N}_{l,\alpha_2} \times \mathcal{N}_{(l-1):1,\alpha_2}\right) \\
&= \left(\mathcal{N}_{K:(l+1),\alpha_1} \times \mathcal{N}_{l,\alpha_1} \times \mathcal{N}_{(l-1):1,\alpha_1}\right) \cup \left(\mathcal{N}_{K:(l+1),\alpha_1} \times \mathcal{N}_{l,\alpha_2} \times \mathcal{N}_{(l-1):1,\alpha_1}\right) \\
&= \mathcal{N}_{K:(l+1),\alpha_1} \times \left(\mathcal{N}_{l,\alpha_1} \cup \mathcal{N}_{l,\alpha_2}\right) \times \mathcal{N}_{(l-1):1,\alpha_1}.
\end{aligned}
$$

Thus, we get $\mathcal{N}_{g,\alpha} = \mathcal{N}_{g,\alpha_1} = \mathcal{N}_{g,\alpha_2} = \mathcal{N}_{g,\alpha_1} \cup \mathcal{N}_{g,\alpha_2}$ for all $g \neq l$, and $\mathcal{N}_{l,\alpha} = \mathcal{N}_{l,\alpha_1} \cup \mathcal{N}_{l,\alpha_2}$.
□

Since the next–state functions of local events for submodel l differ only at level l, we can always apply Lemma 4.3 and merge such events into a single *macro–event* λ_l. This improves the efficiency of the state-space generation algorithm we introduce in Section 5.

Many modeling languages guarantee Kronecker consistency for *any* partition of a given system model. The following theorem shows that any partition of a non–self–modifying Petri net is Kronecker–consistent, even in the presence of inhibitor arcs and reset arcs.

Theorem 4.1. *Given a Petri net with inhibitor arcs and reset arcs $(\mathcal{P}, \mathcal{T}, f, h, r, s)$, any partition of its places $\mathcal{P}$ into $K \leq |\mathcal{P}|$ submodels is Kronecker–consistent.*

Proof: By Lemma 4.2, it suffices to show that the theorem holds for $K = |\mathcal{P}|$, since any other partitioning is a coarsening of that one. In this case, each place is a submodel, and we may name both with the integers from K down to 1. Then, $\mathcal{S}_l$ contains the numbers of tokens that place l can hold, and the local functions for transition α are given by

$$
\mathcal{N}_{l,\alpha}(i_l) = \begin{cases} \emptyset & \text{if } i_l < f(l,\alpha) \text{ or } i_l \geq h(l,\alpha) \\ \{r(l,\alpha)(i_l - f(l,\alpha)) + f(\alpha,l)\} & \text{otherwise}. \end{cases}
$$

This matches Definition 3.1, i.e., event α is enabled in marking **i** if and only if, for all places l, $\mathcal{N}_{l,\alpha}(i_l) \neq \emptyset$ and, if α fires, the new marking **j** satisfies $j_l = r(l,\alpha)(i_l - f(l,\alpha)) + f(\alpha,l)$. Thus, every transition α has a Kronecker representation.   □

This proof suggests the following variant of Theorem 4.1 for a specific class of self–modifying Petri nets.

Theorem 4.2. *Given a self–modifying Petri net $(\mathcal{P}, \mathcal{T}, f, s)$, a transition $\alpha \in \mathcal{T}$ has a Kronecker representation in a partition $\mathcal{P}_K, \ldots, \mathcal{P}_1$ of $\mathcal{P}$ if, for each submodel $\mathcal{P}_l$ and for each*

place $p \in \mathcal{P}_l$, the functions $f(\alpha, p)$ and $f(p, \alpha)$ depend only on the lth *component of the global state, i.e., only on the numbers of tokens in the places of* $\mathcal{P}_l$.

Proof: *(Sketch)* The proof is analogous to that of Theorem 4.1. The restriction on the form of $f(\alpha, p)$ and $f(p, \alpha)$ ensures that $\mathcal{N}_{l,\alpha}$ can be properly defined as a function of $\mathcal{P}_l$ alone, i.e., $\mathcal{N}_{l,\alpha} : \mathbb{N}^{|\mathcal{P}_l|} \to 2^{\mathbb{N}^{|\mathcal{P}_l|}}$. □

Theorem 4.1 also suggests our desired representation of the next–state function $\mathcal{N}$ of a discrete–state model as a collection of boolean matrices, one for each event and submodel.

Corollary 4.1. *The next–state function $\mathcal{N}$ of a discrete–state model defined by a Petri net $(\mathcal{P}, \mathcal{T}, f, h, r, s)$ with inhibitor arcs and reset arcs can be encoded by $|\mathcal{P}| \cdot |\mathcal{T}|$ square boolean matrices $\mathbf{N}_{l,\alpha}$, for $\alpha \in \mathcal{T}$ and $l \in \mathcal{P}$ satisfying: (i) $\mathbf{N}_{l,\alpha}$ is of size n_l, where n_l is the number of different token counts i_l that l may contain in any reachable marking; and (ii) $\mathbf{N}_{l,\alpha}[i_l, j_l] = 1$ if and only if $(i_l \geq f(l, \alpha)) \wedge (j_l = r(l, \alpha)(i_l - f(l, \alpha)) + f(\alpha, l)) \wedge (i_l < h(l, \alpha))$.*

We conclude this section by observing that, for practical purposes, we can automatically extract the finest Kronecker–consistent partition of an arbitrary self–modifying Petri net. In our tool SMART [8], we simply have to parse the expressions used to specify $f(\alpha, p)$ and $f(p, \alpha)$, for each transition α. Alternatively, if we are also given a partition, we can automatically find the finest Kronecker–consistent coarsening of that partition, by applying Lemma 4.2, or the coarsest refinement of the transitions in the net so that the given partition is Kronecker–consistent, by applying Lemma 4.1. These statements must be qualified, however, since our approach is purely syntactic, i.e., we must assume that, if the number of tokens for some place appears in an expression, then the expression truly depends on it.

4.1.3 Storing the next–state function of our running example

Returning to our example of Fig. 2, we can now show how its next–state function may be encoded using Kronecker representations. For the partition into five submodels, the corresponding boolean matrices are given in the top table of Fig. 3, when indexing the local states as shown on the left–hand side of Fig. 5. An empty cell at level l for event α signifies that $\mathbf{N}_{l,\alpha}$ is the identity matrix. Observing that both events b and c depend on levels 4 and 3, one could decide to merge these two levels, i.e., assign places q and r to the same submodel, as discussed at the end of Section 3.3. The resulting matrices are displayed in the middle of Fig. 3. Finally, since events b and c now depend only on the revised level 3, they are both local events for level 3 and can be merged into the single macro–event λ_3. This yields the matrices at the bottom of Fig. 3. Note that these matrices coincide with the ones in the previous case, except that the new column for the macro–event λ_3 is obtained as the boolean sum of the old columns for events b and c.

4.1.4 Implementation of Kronecker representations

A naive implementation for the Kronecker representation of $\mathcal{N}$ according to Corollary 4.1 uses a number of boolean matrices equal to $|\mathcal{P}| \cdot |\mathcal{T}|$, and is thus quadratic in the size of the Petri net, where the $|\mathcal{T}|$ matrices for level l are themselves quadratic in the size n_l of the corresponding local state space $\mathcal{S}_l$. However, by exploiting the presence of identity matrices

$K = 5$, total of 5 events

Event→ Level↓	$Top(a)=5$ $Bot(a)=2$	$Top(b)=4$ $Bot(b)=3$	$Top(c)=4$ $Bot(c)=3$	$Top(d)=2$ $Bot(d)=1$	$Top(e)=5$ $Bot(e)=1$
5	$\mathbf{N}_{5,a}=\begin{bmatrix}0&1\\0&0\end{bmatrix}$				$\mathbf{N}_{5,e}=\begin{bmatrix}0&0\\1&0\end{bmatrix}$
4	$\mathbf{N}_{4,a}=\begin{bmatrix}0&1\\0&0\end{bmatrix}$	$\mathbf{N}_{4,b}=\begin{bmatrix}0&1\\0&0\end{bmatrix}$	$\mathbf{N}_{4,c}=\begin{bmatrix}0&0\\1&0\end{bmatrix}$		
3		$\mathbf{N}_{3,b}=\begin{bmatrix}0&0\\1&0\end{bmatrix}$	$\mathbf{N}_{3,c}=\begin{bmatrix}0&1\\0&0\end{bmatrix}$		$\mathbf{N}_{3,e}=\begin{bmatrix}0&0\\1&0\end{bmatrix}$
2	$\mathbf{N}_{2,a}=\begin{bmatrix}0&1\\0&0\end{bmatrix}$			$\mathbf{N}_{2,d}=\begin{bmatrix}0&0\\1&0\end{bmatrix}$	
1				$\mathbf{N}_{1,d}=\begin{bmatrix}0&1\\0&0\end{bmatrix}$	$\mathbf{N}_{1,e}=\begin{bmatrix}0&0\\1&0\end{bmatrix}$

$K = 4$, total of 5 events

Event→ Level↓	$Top(a)=4$ $Bot(a)=2$	$Top(b)=3$ $Bot(b)=3$	$Top(c)=3$ $Bot(c)=3$	$Top(d)=2$ $Bot(d)=1$	$Top(e)=4$ $Bot(e)=1$
4	$\mathbf{N}_{4,a}=\begin{bmatrix}0&1\\0&0\end{bmatrix}$				$\mathbf{N}_{4,e}=\begin{bmatrix}0&0\\1&0\end{bmatrix}$
3	$\mathbf{N}_{3,a}=\begin{bmatrix}0&1&0\\0&0&0\\0&0&0\end{bmatrix}$	$\mathbf{N}_{3,b}=\begin{bmatrix}0&0&0\\0&0&0\\0&1&0\end{bmatrix}$	$\mathbf{N}_{3,c}=\begin{bmatrix}0&0&0\\0&0&1\\0&0&0\end{bmatrix}$		$\mathbf{N}_{3,e}=\begin{bmatrix}0&0&0\\0&0&0\\1&0&0\end{bmatrix}$
2	$\mathbf{N}_{2,a}=\begin{bmatrix}0&1\\0&0\end{bmatrix}$			$\mathbf{N}_{2,d}=\begin{bmatrix}0&0\\1&0\end{bmatrix}$	
1				$\mathbf{N}_{1,d}=\begin{bmatrix}0&1\\0&0\end{bmatrix}$	$\mathbf{N}_{1,e}=\begin{bmatrix}0&0\\1&0\end{bmatrix}$

$K = 4$, total of 4 events (b and c merged into λ_3)

Event→ Level↓	$Top(a)=4$ $Bot(a)=2$	$Top(\lambda_3)=3$ $Bot(\lambda_3)=3$	$Top(d)=2$ $Bot(d)=1$	$Top(e)=4$ $Bot(e)=1$
4	$\mathbf{N}_{4,a}=\begin{bmatrix}0&1\\0&0\end{bmatrix}$			$\mathbf{N}_{4,e}=\begin{bmatrix}0&0\\1&0\end{bmatrix}$
3	$\mathbf{N}_{3,a}=\begin{bmatrix}0&1&0\\0&0&0\\0&0&0\end{bmatrix}$	$\mathbf{N}_{\lambda_3,3}=\begin{bmatrix}0&0&0\\0&0&1\\0&1&0\end{bmatrix}$		$\mathbf{N}_{3,e}=\begin{bmatrix}0&0&0\\0&0&0\\1&0&0\end{bmatrix}$
2	$\mathbf{N}_{2,a}=\begin{bmatrix}0&1\\0&0\end{bmatrix}$		$\mathbf{N}_{2,d}=\begin{bmatrix}0&0\\1&0\end{bmatrix}$	
1			$\mathbf{N}_{1,d}=\begin{bmatrix}0&1\\0&0\end{bmatrix}$	$\mathbf{N}_{1,e}=\begin{bmatrix}0&0\\1&0\end{bmatrix}$

Fig. 3 Storing $\mathcal{N}$ for our running example using a Kronecker encoding

and the subsystem structure of the model, a linear number of matrices, each itself of linear size, suffices.

To see this, note that Corollary 4.1 implies that each $\mathbf{N}_{l,\alpha}$ contains at most one entry equal to 1 per row, thus it can be stored using an integer vector of size n_l, whose entry i_l is either the unique local state j_l satisfying $\mathcal{N}_{l,\alpha}(i_l) = \{j_l\}$, or "$-1$" if $\mathcal{N}_{l,\alpha}(i_l) = \emptyset$. The same corollary also implies that $\mathbf{N}_{l,\alpha}$ is the identity if $f(l, \alpha) = 0$, $f(\alpha, l) = 0$, $h(l, \alpha) = \infty$, and $r(l, \alpha) = 1$, i.e., if place l and transition α are not connected by any input, output, inhibitor, or reset arc. Since identity matrices do not need to be represented explicitly, we can store the next–state function $\mathcal{N}$ for a Petri net of this class using at most $|\mathcal{P}| + |\mathcal{T}| + \sum_{(l,\alpha)\in\mathcal{F}}(n_l + 1)$ integers, where "$|\mathcal{P}|$" corresponds to a vector needed to record the sizes of the local state spaces, "$|\mathcal{T}|$" corresponds to pointers to a sparse transition–wise addressing scheme for the
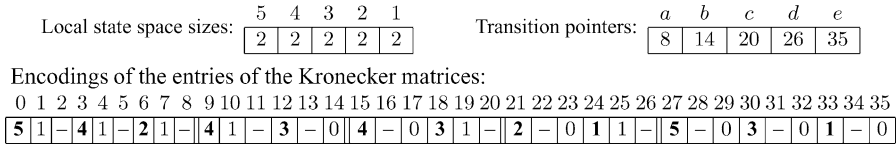
Local state space sizes:

5	4	3	2	1
2	2	2	2	2

Transition pointers:

a	b	c	d	e
8	14	20	26	35

Encodings of the entries of the Kronecker matrices:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35

| **5** | 1 | – | **4** | 1 | – | **2** | 1 | – | **4** | 1 | – | **3** | – | 0 | **4** | – | 0 | **3** | 1 | – | **2** | – | 0 | 1 | 1 | – | **5** | – | 0 | **3** | – | 0 | **1** | – | 0 |

Fig. 4 Storing $\mathcal{N}$ for our running example (with $K = 5$) using $|\mathcal{P}| + |\mathcal{T}| + 3|\mathcal{F}|$ integers

non–identity matrices, "$(n_l + 1)$" corresponds to an integer vector of size n_l for each matrix for submodel l plus an integer to record the submodel index, and $\mathcal{F}$ describes the transition–place pairs connected by some arc in the net, i.e., $\mathcal{F} = \{(l, \alpha) : f(\alpha, l) \neq 0 \vee f(l, \alpha) \neq 0 \vee r(l, \alpha) \neq 1 \vee h(l, \alpha) \neq \infty\}$. In particular, we obtain the following theorem for *safe* Petri nets, i.e., for those Petri nets for which no place ever contains more than one token.

Theorem 4.3. *For a safe Petri net, possibly with inhibitor arcs and reset arcs, the next–state function can be encoded in linear space; it requires at most $|\mathcal{P}| + |\mathcal{T}| + 3|\mathcal{F}|$ integers, where $\mathcal{F}$ is the set of arcs in the net.*

Proof: *(Sketch)* The proof is a special case of the discussion above, when $n_l = 2$. □

As a consequence of this theorem, the set of matrices in the top portion of Fig. 3 (case $K = 5$ and 5 events), can be encoded by the three vectors shown in Fig. 4, where $|\mathcal{P}| + |\mathcal{T}| + 3|\mathcal{F}| = 5 + 5 + 3 \cdot 12 = 46$. Each entry in the vector of transition pointers points to the end of the encoding of the sequence of matrices for the corresponding transition. For readability, the numbers corresponding to the level indices are in boldface, "–" represents "-1", i.e., the transition is disabled, and the separators "||" have been added (their position in the vector encoding the entries of the Kronecker matrices can be inferred from the local state-space sizes and the transition–pointer vector).

4.2 MDDs to store state spaces

Concurrent systems consisting of multiple subsystems give rise to state spaces whose characteristic function is of the form $\mathcal{S}_K \times \mathcal{S}_{K-1} \times \cdots \times \mathcal{S}_1 \to \{0, 1\}$. Since we assume that a system's local state spaces $\mathcal{S}_l$ are finite, we may identify each local state with an integer in the range $\{0, 1, \ldots, n_l - 1\}$. State spaces may thus be represented naturally via *Multi–valued Decision Diagrams*. These were proposed by Kam et al. [24] and encode functions of the form

$$\{0, 1, \ldots, n_K - 1\} \times \{0, 1, \ldots, n_{K-1} - 1\} \times \cdots \times \{0, 1, \ldots, n_1 - 1\} \to \{\mathbf{0}, \mathbf{1}\}.$$

Intuitively, Multi–valued Decision Diagrams generalize BDDs by extending the constant–two fan–out of BDD nodes to larger fan–outs, namely fan–out n_l for nodes at level l. The particular variant of Multi–valued Decision Diagrams we use is defined as follows.

Definition 4.2. A *Multi–valued Decision Diagram*, or MDD for short, is a directed acyclic edge–labeled multi–graph with the following properties:

- Nodes are organized into $K + 1$ *levels*. We write $p.lvl$ to denote the level of node p.
- Level K contains only a single *non–terminal* node r, the *root*, whereas levels $K - 1$ through 1 contain one or more non–terminal nodes.
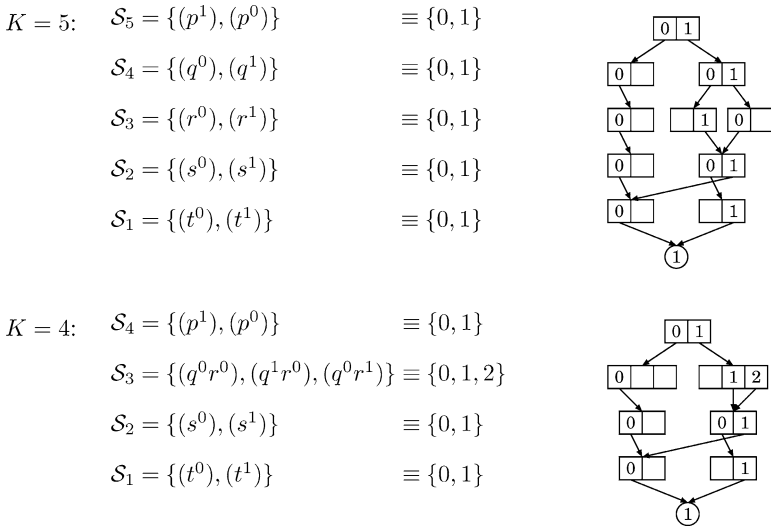
$K = 5:$ $\mathcal{S}_5 = \{(p^1), (p^0)\}$ $\equiv \{0, 1\}$

$\mathcal{S}_4 = \{(q^0), (q^1)\}$ $\equiv \{0, 1\}$

$\mathcal{S}_3 = \{(r^0), (r^1)\}$ $\equiv \{0, 1\}$

$\mathcal{S}_2 = \{(s^0), (s^1)\}$ $\equiv \{0, 1\}$

$\mathcal{S}_1 = \{(t^0), (t^1)\}$ $\equiv \{0, 1\}$

$K = 4:$ $\mathcal{S}_4 = \{(p^1), (p^0)\}$ $\equiv \{0, 1\}$

$\mathcal{S}_3 = \{(q^0 r^0), (q^1 r^0), (q^0 r^1)\} \equiv \{0, 1, 2\}$

$\mathcal{S}_2 = \{(s^0), (s^1)\}$ $\equiv \{0, 1\}$

$\mathcal{S}_1 = \{(t^0), (t^1)\}$ $\equiv \{0, 1\}$

Fig. 5 Storing $\mathcal{S}$ for our running example using MDDs (top: $K = 5$; bottom: $K = 4$)

- Level 0 consists of the two *terminal* nodes, **0** and **1**.
- A non–terminal node p at level l has n_l arcs pointing to nodes at level $l-1$. An arc from position $i_l \in \mathcal{S}_l$ to node q is denoted by $p[i_l] = q$.
- No two nodes are *duplicates*. Two distinct non–terminal nodes p and q at level l are *duplicates* if $p[i_l] = q[i_l]$ for all $0 \le i_l < n_l$.

In contrast to [24], our MDDs are not fully–reduced but *quasi–reduced* [26], as our definition permits *redundant* nodes, i.e., non–terminal nodes p at level l such that $p[i_l] = p[0]$ for all $0 \le i_l < n_l$. As shown for quasi–reduced ordered BDDs, which correspond to the special case of our MDDs where $n_l = 2$ for all $K \ge l \ge 1$, this does not affect canonicity; quasi–reduced MDDs may simply be understood as minimized deterministic automata [26].

Given a node p at level l, we recursively define the node reached from it through a *substate* starting at level l, i.e., a sequence σ of local states $(i_l, \ldots, i_{l'})$, as

$$p[\sigma] = \begin{cases} p & \text{if } \sigma = (), \text{ the empty sequence} \\ p[i_l][\sigma'] & \text{if } \sigma = (i_l, \sigma'), \quad \text{with } i_l \in \mathcal{S}_l. \end{cases}$$

Given a node p at level l, the substates encoded by p or reaching p, are then, respectively,

$$\mathcal{B}(p) = \{\sigma \in \mathcal{S}_l \times \cdots \times \mathcal{S}_1 : p[\sigma] = \mathbf{1}\} \quad \text{``below'' } p;$$
$$\mathcal{A}(p) = \{\sigma \in \mathcal{S}_K \times \cdots \times \mathcal{S}_{l+1} : r[\sigma] = p\} \quad \text{``above'' } p.$$

Consequently, $\mathcal{B}(p)$ contains the substates that, prefixed by a substate in $\mathcal{A}(p)$, form a global state encoded by the MDD. For technical convenience, we reserve a special node z_l at level l to encode the empty set: $\mathcal{B}(z_l) = \emptyset$. Note that node z_l has all arcs pointing to node z_{l-1}, with z_0 corresponding to terminal node **0**.

4.2.1 Storing the state space of our running example with MDDs

The state space for the Petri net of Fig. 2, when the initial state is $(p^1q^0r^0s^0t^0)$ and the model is partitioned into five submodels, is encoded by the 5–level MDD in Fig. 5, top; the elements of the local state spaces $\mathcal{S}_l$ and their mapping to $\{0, \ldots, n_l-1\}$ are shown to the left of the MDD. We use a *discovery order* for the mapping: submarkings are indexed in the order they might be found by a breadth–first exploration from the initial marking, which then always corresponds to the global state where all local states have index 0. In particular, the indices of local states (p^1) and (p^0) in $\mathcal{S}_K$ are 0 and 1, respectively, not 1 and 0.

Graphically, we display a node p at level l as an array of size n_l indexed from 0 to n_l-1; to improve readability, we write the indices inside the cell. If $p[i_l] = q \neq z_{l-1}$, we draw an arc from index i_l of the array to node q. If $p[i_l] = z_{l-1}$, we omit the arc and the index, since only terminal node **1** is displayed.

The MDD resulting when partitioning our example Petri net into four submodels, with places q and r assigned to the same submodel, is displayed in Fig. 5, bottom. One of the local state spaces, $\mathcal{S}_3$, contains now three, not two, states. Had we included state (q^1r^1) in the definition of $\mathcal{S}_3$, the arrays depicting the nodes at level 3 of the the MDD would have had a fourth entry, corresponding to this local state, but each associated arc would have been to node z_2, i.e., this entry would not have been used to encode any reachable global state.

This mapping of concurrent systems onto MDD levels is the reason why we label subsystems "backwards", i.e., from K to 1. When employing this convention, as has been done in the literature before (see, e.g., [40]), the MDD terminal nodes always reside at level 0.

4.2.2 Implementation of MDDs

While the Multi-valued Decision Diagrams proposed in [24] were implemented through BDDs, we showed in [29] that implementing MDDs directly may result in greater efficiency during state-space generation. This is because accessing and manipulating a local state requires us to work on a single MDD node only, rather than on multiple BDD nodes that need to be recursively traversed. Implementing MDDs directly offers a few additional challenges with respect to the more commonly used BDDs.

The main and most obvious challenge is a consequence of nodes at different levels having different sizes, which implies that it might not be possible to reuse them across levels. Our solution is to manage K separate pools of nodes, one per level. To avoid duplicate nodes, each pool is managed by a separate *unique table* stored using *extensible arrays*, whose sizes can be increased or reduced during the execution and yet provide constant–time access to any element. More specifically, we store the nodes at a given level l using a node array and an arc array, as shown in Fig. 6. The former is organized according to the MDD node indices, i.e., the constant portion of the data for node p at level l is stored in $node[l][p]$. As such, the index of node p is unique only within its level l; an arbitrary node must therefore be referred to as the pair (l, p). The arc array is indexed by the *start* and *size* field of the node array, i.e., if $node[l][p].start = a$ and $node[l][p].size = b$, then $p[i]$ is stored in $arcs[l][a + i]$, for $0 \leq i < b$. The arcs are stored in *truncated full format*, i.e., $b < n_l$ means that $p[b] = \cdots = p[n_l-1] = z_{l-1}$, where n_l is the node size at level l. Our implementation in SMART [8] also provides a *sparse format*, which stores each arc $p[i_l] \neq z_{l-1}$ using two successive positions in the arc array, one for the local state i_l and one for the value of $p[i_l]$. The format that saves the most memory is chosen individually for each node, and a flag in the node is used to discriminate between the two choices. For convenience, we reserve the index 0 at each level for the special node z_l. This can be physically stored in the node

MDD nodes at a given level l:
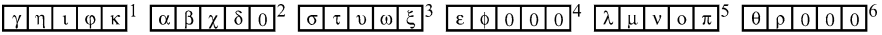
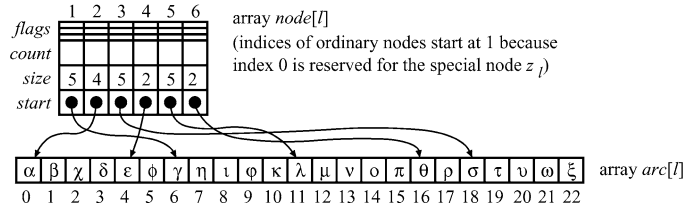Their encoding using array $node[l]$ and array $arc[l]$:

Fig. 6 Implementing MDD nodes using extensible arrays

array, with a size of 0, and inserted into the unique table. Alternatively, node $(l, 0)$ can be a virtual node, not stored in the node array, but this requires ensuring that each newly–created node is not a duplicate of z_l before checking the unique table. SMART uses the latter approach.

As mentioned before, we use quasi–reduced instead of fully–reduced decision diagrams. Indeed, having arcs span only one level has several important implementation advantages. First, any operation on MDDs frequently used in our Saturation algorithm of Section 5, such as the union of two MDDs, is performed on nodes at the same level. As for BDDs, each MDD operation uses a cache to reduce computational complexity. Since our caches are organized by level, there is no need to store level information as part of the key or the result. Analogously, each entry in array $arcs[l]$ stores only the index of the node being pointed to, since its level is known to be $l-1$.

Second, field *count* of $node[l][p]$ represents the number of arcs pointing to node p. When an arc pointing to p is redirected, we reduce $node[l][p].count$ and, if it reaches 0, we know that p has become disconnected. However, index p might still appear in cache entries, thus we cannot recycle p right away. Combined with our level–based node storage, the quasi–reduced form allows us to recycle nodes in array $node[l]$ and compact array $arcs[l]$ independently and efficiently, on a level-by-level basis. To *recycle* all nodes p at level l for which $node[l][p].count = 0$, we simply remove the corresponding entries from the unique table for level l and scan all operation caches at level l to eliminate any entry referring to them. If we also want to *compact* node array $node[l]$, we must update the pointers to the nodes. In our MDDs, this only requires us to update the entries in $arcs[l+1]$. Independently, we can choose to *compact* arc array $arcs[l]$ at any time. This only requires us to update the *start* field of the entries in array $node[l]$.

Third, as will be clear in the discussion of our Kronecker representation of the next–state function and of our Saturation algorithm, greater efficiency is achieved by firing events not from the root node of the MDD, but from each node at the highest level l affected by an event. This includes redundant nodes, which would then have to be recognized, by examining each arc "jumping over" level l, and reintroduced in the MDD, had we used fully–reduced MDDs.

A final reason to use the quasi–reduced form is that, in practice, our experiments confirm that very few, if any, nodes in the MDD encoding the state space of the discrete–state models we studied are redundant. Indeed, it is easy to see that, if a local state space $\mathcal{S}_l$ contains even just one unreachable local state, then the only possible redundant node at level l is the special node z_l. As discussed above, our implementation does not actually store z_l.

5 Saturation–based state-space generation

Our novel algorithm for generating the reachable state space $\mathcal{S}$ of an event-based concurrent system model relies on storing reached states as an MDD and on encoding the next–state function as a Kronecker expression on sparse boolean matrices, as shown in the previous section.

In contrast to related work, we iterate multiple local next–state functions rather than a single global next–state function to obtain state space $\mathcal{S}$. This has two distinct advantages. First, it enables the *local* manipulation of the underlying data structures, which is the key to eliminating computational overheads. Second, it gives us a choice for the order in which to iterate the local next–state functions, while still computing the desired fixed point, due to the inherent monotonicity of state-space generation. The only requirement is that each event must be considered in each state, or at least in each state where it *might* be enabled and it *might* create new states if it fires. In our setting, firing an event is an extremely lightweight operation because of event locality. Event locality ensures that most events affect only a few components of a global state, i.e., a few local states, and this is often even more so as models scale up; indeed, our firing operation exploits the many identity matrices that are part of a Kronecker representation. In turn, monotonicity and event locality pave the road for modifying MDD nodes in place, which is unique to our approach and significantly reduces memory requirements while increasing time efficiency.

5.1 The idea of node saturation

We exploit the freedom of iteration order by suggesting a novel iteration order based on exhaustively firing all events affecting a given MDD node and its descendants, thereby bringing the node to its final, *saturated* shape. Moreover, nodes are considered in a bottom–up fashion, i.e., when a node is processed, all its descendants are already saturated. To aid our presentation, it is convenient to introduce the following notational conventions. We extend the next–state function $\mathcal{N}_{l:g,\alpha}$ for an event α to sets of substates: $\mathcal{N}_{l:g,\alpha}(\mathcal{X}) = \bigcup_{\mathbf{i}\in\mathcal{X}} \mathcal{N}_{l:g,\alpha}(\mathbf{i})$, for $\mathcal{X} \subseteq \mathcal{S}_l \times \cdots \times \mathcal{S}_g$, and to sets of events: $\mathcal{N}_{l:g,\mathcal{E}'}(\mathcal{X}) = \bigcup_{\alpha\in\mathcal{E}'} \mathcal{N}_{l:g,\alpha}(\mathcal{X})$, for $\mathcal{E}' \subseteq \mathcal{E}$. We omit the range of levels when it is clear from the context; in particular, we write $\mathcal{N}_{\leq l}$ as a shorthand for $\mathcal{N}_{l:1,\{\alpha:Top(\alpha)\leq l\}}$. We may now formalize node saturation as follows:

Definition 5.1. An MDD node p at level l is *saturated* if it encodes a set of (sub)states that is a fixed point with respect to the firing of any event affecting only its level or lower levels, i.e., if $\mathcal{B}(p) = \mathcal{N}_{\leq l}^*(\mathcal{B}(p))$ holds.

It can easily be shown by contradiction that, if node p is saturated, any node on a path from p to $\mathbf{1}$ must also be saturated.

Our choice of iteration order improves both memory and execution–time efficiency, for several reasons. First, it ensures that the firing of an event $\alpha \in \mathcal{E}_l = \{\beta \in \mathcal{E}: Top(\beta) = l\}$ at an MDD node p adds as many new states as possible, since all descendants of p have already been saturated. Then, since each node in the final encoding of the state space $\mathcal{S}$ is saturated by definition, any node inserted in the unique table has at least a chance of being still part of the final MDD, while any unsaturated node inserted by a traditional symbolic approach is *guaranteed* to be eventually deleted and replaced with another node encoding a larger subset of states. Finally, once we saturate a node p at level l, we never need to fire any event $\alpha \in \mathcal{E}_l$ in p again, while, in classic symbolic approaches [27], $\mathcal{N}$ is applied to the entire decision diagram at *every iteration*.

The resulting state-space generation algorithm is on average several orders of magnitude more efficient in time *and* memory than existing BDD–based algorithms, as the data in Section 6 will show. Most importantly, the peak memory requirements of our algorithm are often close to its final memory requirements. When compared to our own previous work [9, 29] on MDD–based state-space generation, saturation eliminates much administration overhead, reduces the average number of event firings, and enables a simpler and more efficient cache management.

5.2 The saturation algorithm

Our algorithm implementing the idea of node saturation is shown in Fig. 7. It consists of routine *Generate* for initialization, routine *Saturate* to control node saturation, and routine *Fire* to recursively fire events at lower levels. Routines *Saturate* and *Fire* are mutually recursive. Obviously, saturating a node requires firing events on this node, whence *Saturate* calls *Fire*. The mutual recursion is necessary to implement our desired invariant that the descendants of a node being saturated are already saturated; every new node generated during the execution of *Fire* will immediately be saturated by calling *Saturate* on this new node. However, before commenting on our routines in more detail, we introduce the underlying data types and data structures, as well as some generic routines for manipulating them which are adapted from the literature.

5.2.1 Data structures

Our algorithm's basic data types are *evnt* (event), *lcl* (local state), *lvl* (MDD level), and *node* (MDD node), which in practice are simply integers in appropriate ranges. Just as in traditional symbolic state-space generation algorithms, we use a *unique table*, to detect duplicate nodes, and *operation caches*, in particular a *union cache* and a *firing cache* to avoid potentially expensive re–computations of operations already carried out. However, our approach is distinguished by the fact that only saturated nodes are checked in the unique table or referenced in the caches. As mentioned in Section 4.2.2, the table and cache data structures are organized along MDD levels. For $K \geq l \geq 1$, $UT[l]$ is a unique table for nodes at level l; it is used to retrieve the node p given the key $p[0], \ldots, p[n_l - 1]$. For $K > l \geq 1$, $UC[l]$ is the *union cache* for nodes at level l; it is used to retrieve the node s given the nodes $\{p, q\}$ (as an unordered set, since the union operation is commutative), where $\mathcal{B}(s) = \mathcal{B}(p) \cup \mathcal{B}(q)$. For $K > l \geq 1$, $FC[l]$ is the *firing cache* for nodes at level l; it is used to retrieve the node s given node p and event α, where $Top(\alpha) > l \geq Bot(\alpha)$ and $\mathcal{B}(s) = \mathcal{N}^*_{\leq l}(\mathcal{N}_\alpha(\mathcal{B}(p)))$. Note that $FC[l]$ does not contain entries for any event $\alpha \in \mathcal{E}_l$, since our approach saturates a node by modifying it in place, as will be shown below; for the same reason, $UC[K]$ and $FC[K]$ are not needed at all.

5.2.2 Routine generate

The call *Generate*(s) starts off our algorithm by creating the MDD that encodes the initial state s and by immediately saturating each MDD node p it creates, in a bottom–up fashion, by calling *Saturate*(p). It is straightforward to generalize our algorithm from taking a *single* initial state to *multiple* initial states that are represented as an MDD. We simply traverse this MDD in a depth-first fashion and use a flag to recognize MDD nodes that have already been saturated.

$Generate$(in $\mathbf{s}$: array $[1..K]$ of lcl) : $node$

Build an MDD rooted at r, in $UT[K]$, encoding $\mathcal{N}_{\mathcal{E}}^*(\mathbf{s})$ and return r.

1. declare r, p : $node$;
2. declare l : lvl;
3. $p \leftarrow 1$;
4. for $l = 1$ to K do
5. $\quad r \leftarrow NewNode(l)$;
6. $\quad r[\mathbf{s}[l]] \leftarrow p$;
7. $\quad Saturate(r)$;
8. $\quad p \leftarrow r$;
9. return r;

$Saturate$(in p : $node$)

Saturate node p and put it in $UT[p.lvl]$.

1. declare α : $evnt$;
2. declare f, u : $node$;
3. declare i, j : lcl;
4. declare $\mathcal{F}$: set of $evnt$;
5. $\mathcal{F} \leftarrow \mathcal{E}_{p.lvl}$;
6. while $\mathcal{F} \neq \emptyset$ do
7. $\quad \alpha \leftarrow Pick(\mathcal{F})$;
8. $\quad$ foreach $i \in Locals(\alpha, p)$ do
9. $\quad\quad f \leftarrow Fire(\alpha, p[i])$;
10. $\quad\quad$ foreach $j \in \mathcal{N}_{p.lvl,\alpha}(i)$ do
11. $\quad\quad\quad u \leftarrow Union(f, p[j])$;
12. $\quad\quad\quad$ if $u \neq p[j]$ then
13. $\quad\quad\quad\quad p[j] \leftarrow u$;
14. $\quad\quad\quad\quad \mathcal{F} \leftarrow \mathcal{E}_{p.lvl}$;
15. $p \leftarrow Check(p)$;

$Locals$(in α : $evnt$, p : $node$) : set of lcl

Return $\{i \in \mathcal{S}_l : p[i] \neq z_{l-1} \wedge \mathcal{N}_{l,\alpha}(i) \neq \emptyset\}$, the local states in p, a node at level l, that are on a path to $\mathbf{1}$ and locally enable α. In particular, if p is a duplicate of z_l, return $\emptyset$.

$Check$(in p : $node$) : $node$

If $p[0] = \cdots = p[n_l-1] = z_{l-1}$, delete p, a node at level l, and return z_l, since $\mathcal{B}(p)$ is $\emptyset$. If p duplicates q in $UT[l]$, delete p and return q. Otherwise, insert p in $UT[l]$ and return p.

$Pick$(inout $\mathcal{F}$: set of $evnt$) : $evnt$

Remove and return an element from $\mathcal{F}$.

$Fire$(in α : $evnt$, q : $node$) : $node$

Return s s.t. $\mathcal{B}(s) = \mathcal{N}_{\leq q.lvl}^*(\mathcal{N}_\alpha(\mathcal{B}(q)))$. It is called with $q.lvl < Top(\alpha)$.

1. declare f, s : $node$;
2. declare i, j : lcl;
3. if $q.lvl < Bot(\alpha)$ then return q;
4. if $Find(FC[q.lvl],(q,\alpha),s)$ then return s;
5. $s \leftarrow NewNode(q.lvl)$;
6. foreach $i \in Locals(\alpha, q)$ do
7. $\quad f \leftarrow Fire(\alpha, q[i])$;
8. $\quad$ foreach $j \in \mathcal{N}_{l,\alpha}(i)$ do
9. $\quad\quad s[j] \leftarrow Union(f, s[j])$;
10. $Saturate(s)$;
11. $Insert(FC[q.lvl], (q, \alpha), s)$;
12. return s;

$Union$(in p : $node$, q : $node$) : $node$

Assuming that $p.lvl = q.lvl = l$, build an MDD rooted at s encoding $\mathcal{B}(p) \cup \mathcal{B}(q)$, and return s, which is in $UT[l]$.

1. declare i : lcl;
2. declare s : $node$;
3. if $p = z_{p.lvl}$ or $p = q$ then return q;
4. if $q = z_{q.lvl}$ then return p;
5. if $Find(UC[p.lvl], \{p,q\},s)$ then return s;
6. $s \leftarrow NewNode(p.lvl)$;
7. for $i = 0$ to $n_{p.lvl}-1$ do
8. $\quad s[i] \leftarrow Union(p[i], q[i])$;
9. $s \leftarrow Check(s)$;
10. $Insert(UC[p.lvl], \{p, q\}, s)$;
11. return s;

$NewNode$(in l : lvl) : $node$

Create node p at level l with arcs set to z_{l-1}, return p.

$Insert$(inout tab, in key, v)

Insert (key, v) in hash table tab.

$Find$(in tab, key, out v) : $bool$

If (key, x) is in hash table tab, set v to x and return $true$. Otherwise, return $false$.

Fig. 7 Pseudo–code for the Saturation algorithm

5.2.3 Routine saturate

The call $Saturate(p)$ saturates node p at level l. Since our algorithm guarantees that all children of p are already saturated, saturating p involves exhaustively firing all events $\alpha \in \mathcal{E}_l$ affecting level l and possibly lower levels. Node p itself is saturated "in place". For each $i \in Locals(\alpha, p)$, that is, for each $i \in \mathcal{S}_l$ used to currently encode states in this node (i.e., $p[i] \neq 0$) and locally enabling α (i.e., $\mathcal{N}_{l,\alpha}(i) \neq \emptyset$) we call $Fire(\alpha, p[i])$ to compute the result f of firing α in the node pointed by $p[i]$, and then recursively saturating it in place. This set of
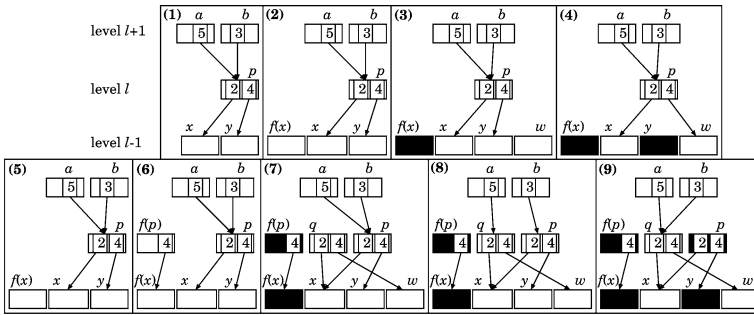
Consider node p in the above scenario **(1)**, and an event α, with $Top(\alpha) = l$ and $\mathcal{N}_{l,\alpha}(2) = \{4\}$.

In-place updates. When saturating p at level l, we **(2)** recursively build node $f(x)$ encoding the result of firing α on x, which is pointed by $p[2]$ (we actually saturate $f(x)$ before returning it, but let us assume that this does not change anything). Then **(3)**, we build node w encoding the union of $f(x)$ and y, which is pointed by $p[4]$. Finally **(4)**, we update p in place, i.e., we set $p[4]$ to w. Note that $f(x)$ can be deleted after step 3, unless it happens to be a previously existing node, while y can be deleted after step 4, unless some other node points to it.

Traditional symbolic approach. When p is reached through arc $a[5]$, one recursively builds **(5)** node $f(x)$ encoding the result of firing α on x, as before. Then **(6)**, one builds node $f(p)$ encoding the result of firing α in p, i.e., $f(p)[4] = f(x)$. Then **(7)**, one builds node q encoding the union of p and $f(p)$; this of course causes the creation of node w as before. Then **(8)**, one sets $a[5]$ to q, which encodes the same set as node p at the end of our approach. Later on in **(9)**, when the recursion reaches b, one descends along $b[3]$ and reach again p, repeating the entire process. The operation caches stop the recursion short, but the cost of cache lookups is not negligible. Furthermore, this approach has higher memory overhead, since node p is *guaranteed* to become disconnected. This is unlike the other nodes marked in black, which might or might not be deleted, depending on whether they are pointed to by other nodes. Finally, note that nodes a and b in steps **8** and **9**, respectively, are modified in our approach as well, whereas the traditional approach would again create new nodes.

Fig. 8 In–place updates vs. ordinary node manipulation during state-space generation

states is added directly to each $\mathcal{B}(p[j])$, for each $j \in \mathcal{N}_{l,\alpha}(i)$, and $p[j]$ is updated accordingly. This is correct because, according to our Kronecker representation of $\mathcal{N}_\alpha$, the enabling and the outcome of firing event α depend only on the states of submodels $Top(\alpha)$ through $Bot(\alpha)$ and is, in particular, independent of the ones in submodels K through $Top(\alpha) + 1$. Updating node p in place means that the effect of firing α benefits all global states through node p, i.e., all paths in $\mathcal{A}(p)$, thus avoiding repetitions of the same work; this is illustrated in Fig. 8.

Whenever new states are found, i.e., the union of $\mathcal{B}(p[j])$ and $\mathcal{B}(f)$ is not $\mathcal{B}(p[j])$, every event in $\mathcal{E}_l$ must be fired again to test for further reachable states. This iteration continues until no more states are found, i.e., until we attempted the firing of each event $\alpha \in \mathcal{E}_l$ once without changing node p. It should be pointed out again that routines *Saturate* and *Fire* are actually mutually recursive. This is because the firing of event $\alpha \in \mathcal{E}_l$ might lead to the creation of new MDD nodes at levels lower than l, which are saturated before the saturation of node p continues.

5.2.4 Routine fire

The second fundamental routine in our algorithm is $Fire(\alpha, q)$, where $q.lvl = l < Top(\alpha)$. Like *Saturate*, *Fire* finds out whether event α is enabled in a node q and its descendants. However, unlike *Saturate*, routine *Fire* operates on a fresh node s instead of modifying q in place, since (i) q is already saturated and in $UT[l]$, (ii) might be already referenced by $UC[l]$ and $FC[l]$, and (iii) might be pointed to by other nodes at level $l + 1$ along paths that do *not*

locally enable α. The third argument shows that updating such a node s, with $l < Top(\alpha)$, is incompatible with interleaving semantics and would thus be incorrect in general.

Accordingly, $Fire(\alpha, q)$ proceeds as follows. First some trivial cases are done away with, namely that $l < Bot(\alpha)$ (Line 3) or that the result of firing α in q has been computed before and is stored in the firing cache (Line 4). The actual work of $Fire$ occurs in Lines 6–9. For each local state $i \in Locals(\alpha, q)$ in which α is enabled, we recursively call $Fire$ (Line 7) and join the resulting set of states represented by node f to each of the i–successor states of node s, as given by the next–state function for event α at level l. This is done by invoking the standard union operation on MDDs (Line 9). If a recursive call determines that event α is not enabled at some lower level $l > g \geq Bot(\alpha)$, i.e., if $Locals(\alpha, q) = \emptyset$ when $q.lvl = g$, then $Fire$ correctly returns z_l, the node at level l representing the empty set. Before returning, node s is saturated by calling $Saturate(s)$ and the result is stored in the firing cache (Lines 10–11), thus maintaining our invariant that lower–level nodes are saturated before higher–level nodes.

Summarizing, the Saturation algorithm brings MDD nodes into their final shapes as early as possible, from the bottom to the top. The firing of events exploits the structure of the partitioned model at hand, as well as the Kronecker representation of the underlying local next–state functions. In–place updates help us reuse MDD nodes as often as possible, rather than generating many new nodes that soon become disconnected, as classical BDD–based approaches do. Together, these features allow us to compute reachable state spaces very efficiently, while keeping peak memory requirements small.

5.3 Proof of correctness

Before presenting an example of our algorithm we prove its correctness. To begin with, we show that the algorithm terminates. The mutual recursion between routines $Saturate$ and $Fire$ must eventually terminate: assuming $p.lvl = l$, $Saturate(p)$ calls $Fire(\alpha, q)$, with $q.lvl = l - 1$ (Line 9), and $Fire$ calls itself only on one level below the one it has been called (Line 7) and calls $Saturate$ at the same level $l - 1$ (Line 10). A similar argument holds for the $Union$ routine, which is taken from the literature [3, 24]. The only other reason for non–termination could be due to an infinite while–loop in $Saturate$ (Line 6). However, the only updates to node p are unions, which are monotonically non-decreasing. Since $\mathcal{B}(p) \subseteq \mathcal{S}_l \times \cdots \times \mathcal{S}_1$ and since each local state space is finite and fixed, we can only increase $\mathcal{B}(p)$, thus modify p and reset the value of $\mathcal{F}$ to $\mathcal{E}_l$, a finite number of times, after which the loop must be exited.

To see that our algorithm correctly computes the reachable state space, we focus closer on the two key routines $Saturate$ and $Fire$. First of all, it is easy to check that the algorithm preserves the invariant that $Fire$ can be invoked only on saturated nodes and that $Saturate$ can be invoked only on nodes whose children are already saturated. This is because routine $Generate$ invokes $Saturate$ in a bottom–up fashion, whereas any fresh node generated in $Fire$ is immediately saturated before returning it (Line 10), and the union of saturated nodes, called by $Saturate$ (Line 10) and $Fire$ (Line 9), is saturated by definition.

Theorem 5.1. *Let p be a node at level l, where $K \geq l \geq 1$, with saturated children, and let*

- *q be one of its children at level $g = l - 1$, satisfying $q \neq 0$;*
- *$\mathcal{U}$ stand for $\mathcal{B}(q)$ before the call $Fire(\alpha, q)$, for some event α with $g < Top(\alpha)$, and $\mathcal{V}$ represent $\mathcal{B}(f)$, where f is the node at level g returned by this call;*
- *$\mathcal{X}$ and $\mathcal{Y}$ denote $\mathcal{B}(p)$ before and after calling $Saturate(p)$, respectively.*

Then, the algorithm guarantees the following two properties:

1. $\mathcal{V} = \mathcal{N}^*_{\leq g}(\mathcal{N}_\alpha(\mathcal{U}))$ *and*
2. $\mathcal{Y} = \mathcal{N}^*_{\leq l}(\mathcal{X})$.

By choosing, for node p, the root node r of the MDD representing the initial state s, we obtain $\mathcal{Y} = \mathcal{N}^*_{\leq K}(\mathcal{B}(r)) = \mathcal{N}^*_{\leq K}(\{s\}) = \mathcal{S}$, as desired.

Proof: To prove both of the above statements we employ a simultaneous induction on l.

- **Inductive base ($l = 1$):** Since local events affecting only a single submodel l are bundled into a single event λ_l, the event set $\mathcal{E}_1$ is either empty or the singleton set $\{\lambda_1\}$. We concentrate on the latter case, as the former is trivial.

  The only possible call $Fire(\lambda_1, 1)$ immediately returns 1 because of the test on $1.lvl$, which has value 0, in line 3. Then, $\mathcal{U} = \mathcal{V} = \{()\}$ and $\{()\} = \mathcal{N}^*_{\leq 0}(\mathcal{N}_{\lambda_1}(\{()\}))$. Hence, Proposition (1) holds.

  The call $Saturate(p)$ repeatedly explores λ_1, the only event in $\mathcal{E}_1$, in every local state i for which, according to the definition of routine $Locals$, (i) $\mathcal{N}_{1,\lambda_1}(i) \neq \emptyset$ and (ii) either $p[i] = 1$, or $p[i]$ has been modified from 0 to 1 (cf. line 13) in a previous iteration of the outermost foreach–loop, because of some other local state i' satisfying $i \in \mathcal{N}_{1,\lambda_1}(i')$ and $p[i'] = 1$. The iteration stops when further attempts to fire λ_1 do not add any new state to $\mathcal{B}(p)$. At this point, $\mathcal{Y} = \mathcal{N}^*_{\lambda_1}(\mathcal{X}) = \mathcal{N}^*_{\leq 1}(\mathcal{X})$, which is Proposition (2).

- **Inductive step ($l-1$ implies l):** We first verify Proposition (1). A call $Fire(\alpha, q)$ can be resolved in three ways. If $q.lvl = g < Bot(\alpha)$, the returned value is $f = q$ and $\mathcal{N}_{g,\alpha}(\mathcal{U}) = \mathcal{U}$ for any set $\mathcal{U}$; as q is saturated and $\mathcal{N}_{g:1,\alpha}$ is the identity, $\mathcal{B}(q) = \mathcal{N}^*_{\leq g}(\mathcal{B}(q)) = \mathcal{N}^*_{\leq g}(\mathcal{N}_\alpha(\mathcal{B}(q)))$. If $g \geq Bot(\alpha)$ but $Fire$ has been called previously with the same parameters, then the call $Find(FC[g], \{q, \alpha\}, s)$ is successful. Since node q is in the unique table, it is saturated and has thus not been modified further. Hence, the value s in the cache is still valid and can be safely used. Finally, we need to consider the case where the call $Fire(\alpha, q)$ performs real work. First, a new node s at level g is created, having all its arcs initialized to 0. We explore the firing of α in each state i satisfying $q[i] \neq 0$ and $\mathcal{N}_{g,\alpha}(i) \neq \emptyset$. According to the induction hypothesis, the recursive call $Fire(\alpha, q[i])$ returns $\mathcal{N}^*_{\leq g-1}(\mathcal{N}_\alpha(\mathcal{B}(q[i])))$. Hence, we have $\mathcal{B}(s) = \bigcup_{i \in \mathcal{S}_g} \mathcal{N}_{g,\alpha}(i) \times \mathcal{N}^*_{\leq g-1}(\mathcal{N}_\alpha(\mathcal{B}(q[i]))) = \mathcal{N}^*_{\leq g-1}(\mathcal{N}_\alpha(\mathcal{B}(q)))$ when the outer loop terminates, i.e., all children of s are saturated. By induction hypothesis, the call $Saturate(s)$ saturates s. Consequently, we have $\mathcal{B}(s) = \mathcal{N}^*_{\leq g}(\mathcal{N}^*_{\leq g-1}(\mathcal{N}_\alpha(\mathcal{B}(q)))) = \mathcal{N}^*_{\leq g}(\mathcal{N}_\alpha(\mathcal{B}(q)))$ after the call.

  Regarding Proposition (2), $Saturate(p)$ repeatedly explores the firing of each event α that is locally enabled in $i \in \mathcal{S}_l$, by calling $Fire(\alpha, p[i])$ which, as shown above and since $g = l - 1$, returns $\mathcal{N}^*_{\leq l-1}(\mathcal{N}_\alpha(\mathcal{B}(p[i])))$. Further, $Saturate(p)$ terminates when firing the events in $\mathcal{E}_l = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$ does not add any state to $\mathcal{B}(p)$, and the set $\mathcal{Y}$ encoded by p is the fixed point of the iteration

$$\mathcal{Y}^{(m+1)} \leftarrow \mathcal{Y}^{(m)} \cup \mathcal{N}^*_{\leq l-1}(\mathcal{N}_{\alpha_1}(\mathcal{N}^*_{\leq l-1}(\mathcal{N}_{\alpha_2}(\cdots \mathcal{N}^*_{\leq l-1}(\mathcal{N}_{\alpha_m}(\mathcal{Y}^{(m)}))\cdots)))),$$

initialized with $\mathcal{Y}^{(0)} \leftarrow \mathcal{X}$. Hence, $\mathcal{Y} = \mathcal{N}^*_{\leq l}(\mathcal{X})$, as desired.

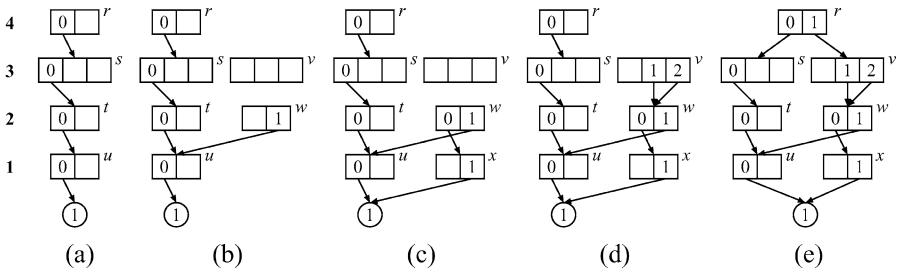This completes the correctness proof of our Saturation algorithm.   □

Fig. 9 Applying the Saturation algorithm to our running example

5.4 Applying the Saturation algorithm to our running example

We now demonstrate the Saturation algorithm on our running example, when the Petri net is partitioned into four subsystems, leading to a four–level MDD to store the set of reachable states. Moreover, the event set $\mathcal{E} = \{a, b, c, d, e\}$ is split into the sets $\mathcal{E}_1 = \emptyset$, $\mathcal{E}_2 = \{d\}$, $\mathcal{E}_3 = \{\lambda_3\}$, and $\mathcal{E}_4 = \{a, e\}$; recall that $\mathcal{E}_l$ denotes the set of events α for which $Top(\alpha) = l$, and that event λ_3 combines the local events b and c. The lowest levels affected by the events in $\mathcal{E}$ are as follows: $Bot(a) = 2$, $Bot(\lambda_3) = 3$ and $Bot(d) = Bot(e) = 1$. Recall that the local next–state functions of our example are stored by the matrices shown at the bottom of Fig. 3.

We begin the Saturation algorithm by invoking *Generate* on the initial marking (0000). This creates the MDD depicted in Fig. 9(a) whose nodes are successively saturated bottom–up. Note that nodes r, s, and t are actually created later on, but we show them here from the beginning for clarity. The levels of the nodes are given at the very left of the MDD figures.

Starting with node u, we observe that this is trivially saturated since $\mathcal{E}_1 = \emptyset$. Thus, we move one level up and consider node t. This node, too, is saturated, since d is the only event in $\mathcal{E}_2$, and $Locals(d, t) = \emptyset$. The same is true for node s at the next higher level since $Locals(\lambda_3, s) = \emptyset$.

Saturating node r turns out to be more interesting, as event $a \in \mathcal{E}_4$ is enabled because $Locals(a, r) = \{0\}$, $Locals(a, s) = \{0\}$, and $Locals(a, t) = \{0\}$. When moving down the MDD recursively, *Fire* generates the fresh nodes v and w at levels 3 and 2, respectively. The recursive calls end at level 1, since $Bot(a) = 2$, and return node u. Since $\mathcal{N}_{2,a}(0) = \{1\}$, the 1–successor of w is pointed to u, as shown in Fig. 9(b).

Before continuing with saturating node r, we must saturate node w. Since $Locals(d, w) = \{1\}$ and $Locals(d, u) = \{0\}$, the only event d in $\mathcal{E}_2$ is enabled, and *Fire* generates a fresh node x at level 1. Further, since $\mathcal{N}_{1,d}(0) = \{1\}$ and $\mathcal{N}_{2,d}(1) = \{0\}$, the 1–successor of x points to the terminal node **1** and the 0–successor of node w to node x, respectively. Note that the fresh node x is already saturated as $\mathcal{E}_1 = \emptyset$. Further firings of d in node w are not possible, whence the node is saturated, too. The resulting MDD is depicted in Fig. 9(c).

We continue with the saturation of node r in the middle of a call to *Fire* with event a, and study the effect on the fresh node v. Since $\mathcal{N}_{3,a}(0) = \{1\}$, the 1–successor of v is set to point to node w, and *Saturate*(v) is invoked. This leads us to considering v twice, until it is saturated. First, $\mathcal{N}_{3,\lambda_3}(1) = \{2\}$, which updates node v in place by inserting an arc from the 2–successor of v to w. Then, the local event λ_3 is enabled again in the local state 2 that has just been added; however, since $\mathcal{N}_{3,\lambda_3}(2) = \{1\}$, firing λ_3 a second time does not reveal any new reachable local states and thus no further in–place update is made. This leaves us with node v saturated and the MDD depicted in Fig. 9(d). Again, we return to the saturation of

node r with respect to event a, which we can now complete. Since $\mathcal{N}_{4,a}(0) = \{1\}$, we add an arc from the 1–successor of node r to node v, which results in the MDD shown in Fig. 9(e).

We continue saturating node r with respect to event e, which is now enabled via the sequence $r[1]$, $v[2]$, $w[0]$, and $x[1]$. This generates the fresh nodes y, y', and y'' (not shown) at levels 3, 2, and 1, respectively, strictly less than $Top(e) = 4$. As $\mathcal{N}_{1,e}(1) = \{0\}$, the 0–successor of the fresh node y'' is pointed to the terminal node $\mathbf{1}$. Herewith, node y'' is already saturated and, since it is identical to u, node y'' is discarded and instead u is returned as the result of $Fire(e, x)$. As a consequence and since $\mathcal{N}_{2,e}(0) = \{0\}$, the 0–successor of fresh node y' is pointed to node u. Hence, y' becomes identical to the saturated node t and is thus itself saturated. Similar to above, node y' is discarded and node t is returned as the result of $Fire(e, w)$. Moving up one level to the fresh node y, $\mathcal{N}_{3,e}(2) = 0$, and consequently its 0–successor is now set to node t. Again, this means that node y becomes identical to saturated node s; thus, node y is discarded and s returned as the result of calling $Fire(e, v)$. Finally, $\mathcal{N}_{4,e}(1) = 0$, but the 0–successor of node r already points to node s, so the in–place update of the 0–successor of node r with the union of nodes s and s is trivial. This concludes the call of $Fire(e, r)$. Further firings of events $a, e \in \mathcal{E}_4$ do not reveal any new reachable states, thus the root node r is saturated. This terminates the Saturation algorithm, i.e., the state space of our example Petri net has been fully generated and is encoded by the MDD of Fig. 9(e), which coincides with the desired MDD given in Fig. 5.

To summarize, since MDD nodes are saturated as soon as they are created, each node will either be part of the final diagram or will eventually become disconnected, but never be modified. It is worth pointing out that no node becomes disconnected in our simple example. Once all events in $\mathcal{E}_l$ are exhaustively fired in a node p at level l, any further state discovered that uses p for its encoding benefits in advance from the "knowledge" encapsulated in p and its descendants. This reduces the amount of work needed to construct reachable state spaces and contributes to the Saturation algorithm's time–efficiency and memory–efficiency.

6 Experimental results

This section evaluates the time and memory efficiency of our Saturation algorithm, as implemented in the SMART verification tool [8], by applying Saturation to construct the reachable state spaces of a large suite of examples taken from the literature. Within SMART, we compare the Saturation algorithm to the traditional breadth–first algorithm. We also carefully compare SMART's efficiency to some of the leading symbolic model checkers, namely McMillan's (Cadence) SMV and NuSMV [15] (which is built on top of the CUDD BDD library [37]), and attempt a brief comparison to the popular explicit–state model checker Spin [23].

The chosen examples for our performance studies, ranging from the well–known dining philosophers to a fault–tolerant multiprocessor system, exhibit a wide range of characteristics regarding the locality of events as well as the numbers and sizes of subsystems. Each example is parametric, further allowing us to explore scalability issues with respect to different model structures. Due to space constraints we cannot reproduce the formal models here but restrict ourselves to an informal description of each. All models used in our experiments can be downloaded from the SMART website located at (www.cs.ucr.edu/∼ciardo/SMART/). SMART itself is also freely available for non-commercial use upon request.

6.1 Toggling bits

This simple model describes the status of N bits, b_1 through b_N, which can be *set* (from 0 to 1) and *reset* (from 1 to 0), according to the following rules. Bit b_1 can be set whenever it is

0, or reset whenever it is 1, independently of the other bits. Bit b_i, for $2 \leq i \leq N$, can be set or reset only together with b_1, that is, if both b_1 and b_i are 0, they can both be set to 1, and if they are both 1, they can both be reset to 0. No other change is possible. The MDD has $K = N$ levels, with bit i at level i, thus all events affect some level i plus level 1, except for the two local events that independently toggle bit 1. The state space contains all possible 2^N combinations of bit values, but the bit toggling rules restrict the possible transitions between states.

6.2 Dining philosophers

This classic model is obtained by connecting N identical submodels, one per philosopher, in a circular fashion. Each philosopher starts in the idle state and occasionally decides to eat; to do so he must acquire both his left and right forks, which he then releases when he has finished eating. Since forks can be acquired in any order, this model is known to have two deadlock states, the one where each philosopher has acquired his left fork and waits for his right fork (which, being also the left fork of his right neighbor, will never be released), and the symmetric state where each philosopher has acquired his right fork. Our model assigns one philosopher per level. We also experimented with assigning P philosophers per level, so that the height of the MDD is $K = N/P$ when N is a multiple of P, but this results in larger local state spaces and is less efficient overall (for example, $|\mathcal{S}_l| = 8, 34, 144$, and 610 for $P = 1, 2, 3$, and 4, respectively). Events affecting a level l are either completely local to l or they affect also a neighboring level, $l + 1$ or $l - 1$; the only exceptions are levels 1 and K, since their neighbors are 2 and K, or 1 and $K - 1$, respectively, due to the circular arrangement of the philosophers.

6.3 Round–robin mutex [22]

This protocol solves a specific type of mutual exclusion problem among N processes organized in a circular fashion, requiring access to a shared resource. Each process is mapped to a different level, from $N + 1$ down to 2, while the shared resource corresponds to level 1. All local state spaces are of size 7 except for $|\mathcal{S}_1| = N + 1$, since it encodes the identity of the process that has been granted access to the resource, if any. There are no local events and, in addition to events synchronizing neighboring processes, there are events synchronizing levels n and 1, for $2 \leq n \leq N + 1$. The results from a similar model where the shared resource is instead at level $N + 1$, while the processes are mapped to levels from N down to 1, are essentially the same.

6.4 Queens

The classic N *queens problem* consists in finding all ways to place N queens on a $N \times N$ chess board, with rows and columns indexed from 1 to N, so that they do not attack each other. Since any such configuration of queens must have exactly one queen per row, the state of the model can be encoded by a vector of size N, whose nth entry is the index m of the column containing the queen for row n, or 0 if that queen has not yet been placed on the board. The initial state is thus the vector of 0s and, at each step, a queen is placed on an empty row, avoiding any square on the row that is attacked, vertically or diagonally, by previously placed queens; thus, the evolution halts in exactly N steps, or fewer if it is not possible to place further queens. We generate all legal configurations having *up to N* queens on the board, and consider two versions of this model: one where queens are placed in *sequential*

(SEQ) order from row 1 to row N, and one where they are placed in *random (RND)* order. Of course, the number of *final* configurations having all N queens on the board is the same for the two models, but the *total* number of configurations is larger for the latter. The levels are in one–to–one correspondence to the queens, where level $K = N$ stores the position of queen N, and level 1 that of queen 1. In this model, locality is extremely poor, since each event placing the queen on a row n must consider the state of rows 1 through $n − 1$ for the *SEQ* model, or of all other rows for the *RND* model.

6.5 Fault–tolerant multiprocessor [36]

This models a system of N interconnected computers whose components can fail. The system fails as soon as all N computers fail. A computer fails once two of its memory modules fail, or two of its CPUs fail, or two of its I/O modules fail, or one of its error–handling modules fails. A memory module fails once three of its RAM chips have failed or one of its interface chips has failed. Uncovered failures are possible, e.g., failure of a RAM chip can cause its memory module to fail even if spare RAM chips are available, which can in turn cause the computer to fail even if its other memory modules are operational, which can cause the entire system to fail even if other computers are operational. The state of a memory module is described by the number of failed RAM chips and the number of failed interface chips. The state of a computer is given by the state of its three memory modules together with the number of failed memory modules, CPUs, I/O modules, and error–handling modules. Thus, the overall system state contains $K = 10N + 1$ variables: ten for each computer plus one to keep track of the number of failed computers. We partition this model so that each state variable corresponds to an MDD level, with level 1 used for the number of failed computers. Every event depends on level 1, since all events are disabled once N computers have failed. Thus, every event is synchronizing, but only between level 1 and some of the ten levels corresponding to state variables for a given computer. All local state spaces are of size 2, 3, or 4, except for $|S_1| = N + 1$. We faithfully follow the model in [36], except for splitting some of the transitions to achieve Kronecker–consistency for the SMART model.

6.6 Flexible manufacturing system [14]

This model describes the movement of pallets carrying parts of three different types to be machined and assembled in an automated factory. The model is decomposed into a fixed number of submodels, $K = 16$, but is parameterized by the number N of pallets initially present in three repositories, which affects the size of the local state spaces. Thus, $|S_l| = N + 1$ for all levels l except for some levels corresponding to stations with an admission control policy that limits the number of parts being processed at any given time: $|S_{17}| = 4$, $|S_{11}| = 3$, and $|S_7| = 2$. The model exhibits a moderate degree of locality, as events span from two to six levels.

6.7 Kanban [38]

This model describes a manufacturing system with four similar stations where "kanbans" (tags) control the flow of parts. The overall flow of parts is as follows: a part enters station 1 first, then it is split into two parts that enter stations 2 and 3, then the two parts are joined into a single part again that enters station 4, from which the part leaves the system. A part can enter a station only if a kanban is available at that station and, after it is processed, it undergoes a check to see whether it needs to be re–processed by the same machine. In

particular, a part can leave station 1 only if there is a kanban at both stations 2 and 3, and a part can leave station 2 (resp. 3) only if there is also a part ready to leave station 3 (resp. 2) and a kanban in station 4. Our partition has 16 levels and each local state space has size $N + 1$, where N is the number of kanbans available in each station; all events affect multiple levels. We also considered an alternative partition with just one level per station, where each local state space has size $(N+3)(N+2)(N+1)/6$ and all events are local except the one that splits a part, which affects levels 1, 2, and 3, as well as the one that joins two parts, which affects levels 2, 3, and 4. For small values of N, this rougher partition is more efficient in peak memory consumption, but not in runtime. However, as N grows, the cubic growth of the local state spaces hurts its memory performance: for $N > 6$, the rougher partition becomes less memory efficient, and for $N = 30$, its final and peak memory is about 12 times that of the finer partition. The runtime for the rougher partition is more than 40 times that of the finer partition for $N = 30$.

6.8 Leader election [17]

This randomized asynchronous leader election protocol designates a unique leader among N participants, each with a unique identifier. Each stage of the protocol requires $2N$ messages, sent along a unidirectional ring. Participants can become passive at each stage and the protocol completes when only one participant remains active, at which point it can broadcast the identity of the leader to the other participants. In SMART, the most natural way to model the exchange of integer messages between participants in this protocol is to use a self–modifying net. However, to ensure Kronecker consistency, we must either *merge* places belonging to different participants into a single submodel, or *split* events according to how many tokens they move (according to the integer value being exchanged). The former approach results in $N + 2$ levels and $14N + 2$ events, but one of the local state spaces grows exponentially in N. We therefore concentrate on the latter approach, and on producing a very fine partition with small local state spaces; we use a model with $\mathcal{O}(N^2)$ levels and $\mathcal{O}(N^3)$ events (for each of the N participants, several events must be split into the disjunction of N^2 "smaller" events). The largest local state space has $11N - 1$ states, while most local state spaces have N, 3, or 2 states.

6.9 Comparison within SMART

Table 1 reports the final memory usage for the next–state function $\mathcal{N}$ and the reachable state space $\mathcal{S}$, the peak MDD memory required to construct $\mathcal{S}$, and the CPU time required to construct $\mathcal{N}$ and $\mathcal{S}$ using algorithms *Saturate* or *BFSGenerate* in SMART. All experiments are run on a 3.2 GHz Pentium IV with 1 GB of memory and 512 KB L2 cache. Our breadth–first search implementation performs garbage collection after every G iterations, where G can be modified by the user; the table reports results for running the garbage collector after *every* iteration (columns BFS[1]) and after every 64 iterations (columns BFS[64]). We instead use a "lazy" garbage collection policy for the Saturation algorithm: disconnected nodes are cleaned up only at the end of the computation. Both Saturation and breadth–first search/iteration construct each local state space $\mathcal{S}_k$ in isolation (using explicit local state-space exploration of the kth submodel) before constructing $\mathcal{N}$, and both algorithms use Kronecker encoding for the next–state function $\mathcal{N}$, which is quite compact, except for the leader election model. In this model, a large number of split events is required to achieve Kronecker consistency; alternative approaches would be to generate local state spaces $\mathcal{S}_k$ and the next–state function $\mathcal{N}$ *on–the–fly*, using the technique described in [11] (this allows for using a coarse partition

Table 1 Generating $\mathcal{N}$ and $\mathcal{S}$: Saturation versus BFS in SMART

| | | Memory (Kb = 1,024 bytes) | | | | | CPU Time (seconds) to Generate $\mathcal{N}$ and $\mathcal{S}$ | | |
| | | Final | | Peak MDD memory | | | | | |
N	$\lvert\mathcal{S}\rvert$	$\mathcal{N}$	$\mathcal{S}$	Sat.	BFS[1]	BFS[64]	Sat.	BFS[1]	BFS[64]
Toggling bits: $K = N$, $\lvert\mathcal{S}_l\rvert = 2$ for all l									
128	3.4×10^{38}	17	4	7	567	926	0.2	375.3	17.7
256	1.2×10^{77}	34	7	14	2,254	3,276	0.6	9,758.0	497.1
4,096	$1.0\times10^{1,233}$	544	112	224	–	–	260.5	–	–
Dining philosophers: $K = N$, $\lvert\mathcal{S}_l\rvert = 13$ for all l									
100	5.0×10^{62}	67	28	35	6,920	10,332	0.4	192.9	9.8
400	6.1×10^{250}	265	113	141	112,728	131,528	1.8	100,359	2,989.7
5,000	$6.5\times10^{3,134}$	3,301	1,047	1,758	–	–	98.1	–	–
Round robin mutex: $K = N + 1$, $\lvert\mathcal{S}_l\rvert = 10$ for all l except $\lvert\mathcal{S}_K\rvert = N + 1$									
60	1.6×10^{20}	68	134	144	1,373	3,697	0.4	108.6	24.1
120	3.6×10^{38}	191	506	527	9,424	18,875	2.5	5,489.6	1,143.6
400	2.3×10^{123}	1,510	5,406	5,476	–	–	202.6	–	–
Queens-SEQ: $K = N$, $\lvert\mathcal{S}_l\rvert = N + 1$ for all l									
11	1.7×10^{5}	106	4,150	4,150	4,150	4,159	2.4	3.4	2.6
12	8.6×10^{5}	151	19,454	19,455	19,455	19,466	22.0	34.9	30.9
13	4.7×10^{6}	208	97,469	97,469	97,470	97,483	443.3	798.8	774.7
Queens-RND: $K = N$, $\lvert\mathcal{S}_l\rvert = N + 1$ for all l									
8	1.2×10^{5}	56	591	2,436	4,795	10,981	1.9	5.1	8.2
9	9.2×10^{5}	91	3,276	14,213	29,450	71,693	36.2	112.1	311.1
10	7.5×10^{6}	139	19,222	88,360	182,062	–	1,668.1	5,987.1	–
Fault-tolerant multiprocessor: $K = 10N + 1$, $2 \leq \lvert\mathcal{S}_l\rvert \leq 4$ for all l except $\lvert\mathcal{S}_1\rvert = N + 1$									
10	3.0×10^{26}	143	47	166	11,720	16,754	21.0	7,130.9	253.4
12	5.1×10^{31}	188	66	230	19,836	27,159	40.0	29,497.9	1,049.6
36	3.3×10^{94}	1,157	554	1,773	–	–	3,926.5	–	–
Flexible manufacturing system: $K = 19$, $\lvert\mathcal{S}_l\rvert = N + 1$ for all l except $\lvert\mathcal{S}_7\rvert = 2$, $\lvert\mathcal{S}_{11}\rvert = 3$, $\lvert\mathcal{S}_{17}\rvert = 4$									
30	7.7×10^{14}	18	210	376	47,082	85,710	0.2	4,035.4	324.6
40	2.6×10^{16}	24	428	782	127,483	212,181	0.3	30,207.0	1,964.6
250	3.5×10^{26}	142	67,270	132,029	–	–	42.1	–	–
Kanban: $K = 16$, $\lvert\mathcal{S}_l\rvert = N + 1$ for all l									
40	9.9×10^{14}	26	97	104	38,847	55,772	0.1	2,862.2	171.6
60	7.2×10^{16}	39	206	221	125,225	172,613	0.3	25,591.8	1,577.7
1,000	1.4×10^{30}	626	51,149	55,083	–	–	22.7	–	–
Leader: $K = \mathcal{O}(N^2)$, $\lvert\mathcal{S}_l\rvert \leq \mathcal{O}(N)$ for all l									
12	3.4×10^{10}	8,630	421	582	16,067	40,305	49.1	755.5	174.0
14	1.9×10^{12}	15,576	612	834	29,312	63,724	133.9	2,139.7	474
24	1.0×10^{21}	125,801	2,342	3,022	–	–	3,826.2	–	–

rather than splitting events, since the local state spaces generated in isolation are much larger than the local states actually reached), or to use a version of Saturation that does not require a Kronecker encoding for $\mathcal{N}$, such as the one described in [28]. Finally, since the same model decomposition and variable ordering is used for Saturation and breadth–first iteration, the algorithms generate isomorphic "final" MDDs for $\mathcal{S}$.

Several important conclusions can be reached from the results compiled in Table 1. As expected, increasing the frequency of garbage collection for breadth–first iteration leads to a smaller peak memory size at the cost of significantly increased runtime. The difference

between our Saturation algorithm and the traditional breadth–first iteration is apparent by considering the peak memory consumption, which goes hand–in–hand with the runtimes. Depending on the model, Saturation requires one–to–three orders of magnitude less memory, and one–to–five orders of magnitude less computation time. This difference in peak memory allows Saturation to analyze models for which the breadth–first iteration fails due to excessive peak memory requirements (designated by dashes in the table).

The only exception to Saturation's excellent performance is the queens problem. In the sequential version of this model, Saturation and breadth–first iteration require roughly the same memory and similar computation time. In the random version, the Saturation memory requirements are less than half that of breadth–first, and computation time is less than one third that of breadth–first iteration. We argue that this model represents a worst–case scenario not only for Saturation, since event locality is very poor in the sequential version and non–existent in the random version, but also for symbolic approaches in general, since the memory and time requirements for Saturation and breadth–first iteration are considerably worse than for a good explicit implementation. For example, the explicit decision–tree storage and state-space generation algorithm implemented in SMART requires only 149 seconds and 147 MB for $N = 13$ queens (sequential) and 136 seconds and 74 MB for $N = 10$ queens (random).

In summary, our results testify that exploiting the interleaving semantics of event-based concurrent systems is the key for making their automated verification algorithms truly efficient.

6.10 Comparison with SMV and NuSMV

To illustrate that the breadth–first implementation in SMART is competitive with other symbolic model checkers, we (manually) translated each example model into the SMV input language and ran experiments using NuSMV version 2.3.1 and Cadence SMV. We empirically verified that the resulting models are equivalent, by checking that they have the same initial state and transition relation, and produce the same sets of reachable states. The SMV models used *process* modules to handle asynchronous behavior, with the notable exception of the dining philosophers model, where we specified the transition relation "by hand", using an input file of size $O(N^3)$ for N philosophers; our process version of this model has instead an input file of size $O(N)$, but requires significantly longer runtimes. Finally, we use the same ordering of state variables for the SMV model as the SMART model (both Cadence SMV and NuSMV further decompose state variables into booleans; we use the default ordering for this). We disabled dynamic variable reordering to ensure that the same variable ordering is used throughout.

Table 2 reports the peak memory and the runtimes required to build the next–state function $\mathcal{N}$ and the reachable state space $\mathcal{S}$, using breadth–first symbolic exploration, in SMART, NuSMV, and Cadence SMV. The peak memory consumption for SMART is measured as the memory required for $\mathcal{N}$ plus the peak MDD memory, while the peak memory consumption for NuSMV and Cadence SMV is measured as the peak number of BDD nodes multiplied by the node size (which we assume is 16 bytes). We believe this measurement includes the nodes required for $\mathcal{N}$. The memory for the operation caches or other data structures required to parse and build the model is not included in the measurements. In the case of SMART, the runtimes include the time to generate each local state space $\mathcal{S}_k$ in isolation, to generate and store the required Kronecker matrices for $\mathcal{N}$, and to generate $\mathcal{S}$ using standard symbolic breadth–first iterations. In the case of NuSMV and Cadence SMV, the runtimes include the time to build the BDD encoding of the transition relation (we hand–specified the sizes of the the local state spaces $\mathcal{S}_k$ in the input file) and to generate $\mathcal{S}$ using breadth–first iterations. For NuSMV, we

Table 2 Generating $\mathcal{N}$ and $\mathcal{S}$ using BFS in various tools

		Peak memory (Kb = 1,024 bytes) SMART				CPU Time (seconds) SMART			
N	#Its	BFS^1	BFS^{64}	NuSMV	Cadence	BFS^1	BFS^{64}	NuSMV	Cadence
Toggling bits									
128	255	584	943	7,602	3,519	375.3	17.7	249.3	241.9
256	511	2,288	3,310	7,710	13,654	9,758.0	497.1	12,756.8	4,256.6
Dining philosophers									
80	161	4,455	7,344	294,676	17,461	86.4	4.8	47.5	181.1
100	201	6,986	10,398	566,904	25,943	192.9	9.8	159.4	385.5
Round robin mutex									
50	395	916	2,578	11,564	51,372	44.3	10.3	3,404.6	4,639.1
60	475	1,440	3,764	19,158	82,132	108.6	24.1	8,428.9	11,516.6
Queens-SEQ									
11	12	4,256	4,265	9,975	86,358	3.4	2.6	14.5	20.1
12	13	19,605	19,616	42,359	398,638	34.8	30.9	108.0	125.4
13	14	97,677	97,691	192,862	–	798.8	774.7	1,786.8	–
Queens-RND									
8	9	4,851	11,037	2,874	37,621	5.1	8.2	6.5	10.1
9	10	29,540	71,783	14,570	234,334	112.1	311.1	74.0	79.9
10	11	182,201	–	75,590	–	5,987.1	–	1,184.5	–
Fault-tolerant multiprocessor									
10	81	11,863	16,896	8,939	29,338	7,130.9	253.4	7,714.0	2,166.9
12	97	20,024	27,347	9,042	47,545	29,497.9	1,049.6	28,150.3	5,125.4
Flexible manufacturing system									
20	281	12,256	24,686	8,781	56,525	375.3	30.9	2,843.5	1,635.7
30	421	47,100	85,728	15,950	141,356	4,035.4	324.6	98,357.0	7,713.8
Kanban									
30	421	17,122	25,626	12,188	61,029	730.1	40.8	3,720.9	2,649.4
40	561	38,872	55,797	23,769	135,824	2,862.2	171.6	69,391.1	8,794.9
Leader									
12	144	24,696	48,935	102,056	118,321	755.5	174.0	1,120.3	5,394.7
14	168	44,888	79,299	190,139	211,403	2,139.7	474.1	3,026.3	15,543.0

use a monolithic BDD encoding of $\mathcal{N}$, since the partitioned encodings (using options "-m Threshold" or "-m Iwls95CP") always resulted in longer generation times for $\mathcal{S}$ or the models we considered. Conversely, for Cadence SMV, we use the partitioned encoding of $\mathcal{N}$ since the monolithic encoding (using option "-nopr") led to longer generation times. We note that, for all the models we checked, the three tools required exactly the same number of iterations, as expected.

As seen from the results compiled in Table 2, the S<small>MART</small> breadth–first iteration is significantly faster (up to three orders of magnitude) than both NuSMV and Cadence SMV for all models except the random queens problem. In terms of peak memory, there is no clear winner among the three tools we tested. Again, dashes in the table correspond to runs that could not complete due to excessive peak memory requirements. Perhaps more surprising is the model–dependent difference in performance between NuSMV and Cadence SMV, which use exactly the same input model and variable ordering: for some models, NuSMV has significantly lower runtimes than Cadence SMV, while for other models, the reverse is true.

In summary, we find that the breadth–first iteration in S<small>MART</small> is quite competitive with other symbolic model checking tools. While Saturation is not included in Table 2 due to lack of space, we see from Table 1 that Saturation is one–to–six orders of magnitude faster

than NuSMV and one–to–five orders of magnitude faster than Cadence SMV, except for the random queens problem. Also, the peak memory requirements for Saturation are less than the peak memory requirements for NuSMV and Cadence SMV for all models, and for all models except queens, the difference is one–to–three orders of magnitude.

6.11 Comparison with Spin

The most widely used model checker for verifying concurrent systems, in particular communications protocols and distributed algorithms, is the Spin model checker developed by Holzmann [23] over the past two decades.

Spin is an *explicit–state* model checker, i.e., state spaces are represented explicitly so that each state corresponds to an entity of the underlying data structure. To make this approach work for systems exhibiting interleaving behavior, it is complemented by *partial–order reduction* [39]. This technique exploits the independence of transitions, which implies that many traces of a system model may be equivalent with respect to the properties to be verified. Hence, it is sufficient to explore a single trace of each equivalence class, i.e., only a subset of the globally reachable state space. The success of Spin in commercial applications is due in no small part to its efficient implementation of partial–order reduction, which often results in greatly reduced memory requirements.

Comparing the state-space generator of Spin to Saturation in SMART is quite difficult as their algorithms are very different in nature and seem to have rather complementary strengths. For example, the toggling bits model is built in such a way that all transitions affect each other, as they all toggle bit one. This means that partial–order techniques do not enable any reduction, and the size of the reachable state space is exponential in the number of bits in its model. In contrast, our Saturation algorithm requires memory that is only linear in the number of bits (cf. Table 1), since the transitions involving setting and resetting bit one, located at the bottom level of the MDD, are fired first. The firings of the set and reset transitions for bit i, at level $i > 1$, then fully reuse the Saturation work done at lower levels.

The situation is quite different for the leader election model. This example, taken from the distribution of the Spin model checker, is one where Spin's partial–order reduction algorithm provides maximal benefit: it reduces the state-space growth from exponential to linear in the number of processes participating in the election. We translated this model into SMART's Petri net language. This is a non–trivial task since Spin's modeling language, called *Promela* [23], is at a higher level than Petri nets, combining features of C with Dijkstra's guarded command language and message channels. Our translation encodes message channels, local Promela variables, and program counters using Petri net places. The state space of the resulting model has the same size as for the Promela model in Spin (when run without partial–order reduction), thus we are confident of the exact equivalence of the two models. As shown in Table 1, our Saturation algorithm does not perform as well for the leader election model; its memory and runtimes still grow exponentially in the number of processes. Nevertheless, Saturation is optimal among symbolic approaches for this model, as its final and peak memory consumptions are essentially the same (the peak number of nodes is just two greater than the final number of nodes), which is a much better behavior than breadth–first search.

Hence, partial–order reduction seems to have complementary strength and weaknesses to our Saturation algorithm. A more insightful comparison between SMART and Spin would require implementing Saturation in Spin or equipping SMART with a Promela front–end and partial–order reduction algorithms. Then, the very same models could be used to thoroughly evaluate the two approaches. However, such a major implementation effort is beyond the scope of this article. It should only be mentioned here, for the sake of completeness, that

researchers have tried to integrate partial–order techniques in BDD–based model–checking algorithms [1], with some success. Whether our Saturation algorithm allows for such an integration remains to be seen.

7 Related work, history and outlook

There exists a vast amount of literature on state-space generation, which is impossible to discuss here in full. We restrict ourselves to symbolic techniques that aim at coping with the state–explosion problem inherent in concurrent models that rely on interleaving semantics.

7.1 Related work

Symbolic (or implicit) state-space generation techniques based on decision diagrams, such as BDDs, do not aim at storing fewer states and transitions, as, e.g., partial–order reduction does, but at storing them more compactly, in sublinear space [3]. As early as a decade ago, it has been pointed out that BDDs, although well suited for reasoning about synchronous systems such as hardware circuits, are less successful when dealing with systems such as Petri nets or distributed software [6]. The reason is that BDD–representations of state spaces do not solve the state–explosion problem; in practice, the problem often shifts to a BDD–node explosion problem. To tackle node explosion in BDD–based state-space generators, i.e., to reduce the peak number of BDD nodes, two lines of research have been investigated in the literature: *partitioned representations of the next–state function* and *heuristics for iteration orders*.

7.1.1 Disjunctive partitioning

As mentioned in Section 2, disjunctively partitioning a next–state function and encoding each disjunctive element as a BDD rather than the whole function leads to much smaller encodings. Indeed, experimental studies have shown that this approach can increase the size of manageable state spaces by about one order of magnitude [5]. Our approach also uses a form of disjunctive partitioning of the global next–state function, although the resulting local next–state functions are not encoded as MDDs but further partitioned conjunctively and stored as sparse matrices [29]. This allows us to easily recognize and exploit the presence of identity transformations when an event occurs, and has recently been adopted by other researchers [2]. In addition, our Saturation algorithm does not involve classical MDD operations when applying next–state functions, but direct Kronecker–based manipulations. This leads to considerable efficiency gains.

Rather than partitioning a system's global next–state function, researchers have also experimented with partitioning a system's state space [7, 31]. Each state-space partition can then be represented by a small BDD using its own variable order, even when the overall state space cannot be encoded compactly by a BDD under any variable order.

7.1.2 Iteration order

It is well known in the automated verification community that computing state spaces in breadth–first order often produces intermediate sets of states that fail to have small BDD representations. This situation is normally not prevented but reacted to by changing the variable order underlying BDDs on the fly, hoping for a more compact encoding [20]. In contrast, the success of our Saturation algorithm lies in its iteration order, which tries to

prevent rather than to react to the problem. Complementing our Saturation algorithm with variable–reordering techniques is an interesting possibility that remains to be investigated.

Related work has experimented with different iteration orders combining breadth–first and depth-first searches, with mixed results. Some examples of such iteration orders are *chaining* [35], *guided search* [34], and *partial traversal* [7]. Another approach to iteration orders *freezes* some subsystems in their initial states while symbolically exploring the state space for the other subsystems of a given concurrent system model [21]. In this approach, which can also be combined with disjunctively partitioned next–state functions, subsystems are successively unfrozen until the entire state space has been constructed. While the "freezing" iteration order requires more iterations to generate state spaces than the standard breadth–first search, the produced intermediate BDDs are often smaller. This improves not only memory efficiency but also time efficiency by about one order of magnitude on realistic examples [21]. In contrast, Saturation gives improvements by one or two further orders of magnitude for similar examples. This is because Saturation brings MDD nodes quickly and cheaply into their final shape, without the need to store unsaturated nodes that are guaranteed to become disconnected later.

7.2 History of our approach

The history of our own approach leading to the Saturation algorithm is documented in a series of three conference papers that form the basis of this article. The earliest paper [29] introduced the ideas to use MDDs rather than BDDs for storing the state spaces of partitioned Petri nets, to represent the Kronecker–consistent next–state function of a partitioned Petri net via sparse boolean matrices instead of MDDs, and to optimize the traversal of MDDs for local events that affect only a single partition or MDD level. Our second paper [9] then generalized the optimizations conducted for local events to all system events, thus exploiting the local effect of firing events in concurrent systems, i.e., exploiting that the firing of an event frequently depends on and updates only very few of a system's subsystems. We also showed in [9] that, as a consequence of this, the heuristics for the underlying iteration order has an important impact on the efficiency of state-space generation and in particular on the number of peak MDD nodes. Finally, our third paper [10] focused on a specific heuristics for the iteration order, namely node saturation, and proposed the Saturation algorithm that is the focus of this article.

7.3 Outlook—model checking with Saturation

Much related work employs the computation of the reachable state space, such as performed by our Saturation algorithm, for checking temporal properties of concurrent systems. Indeed, recent work has seen the Saturation algorithm being used for symbolic CTL–based model checking [12], where it has proved to efficiently model check the important class of interleaving–based concurrent systems which had been thought to be out of the reach of symbolic model checkers before. If the property under consideration is violated, such a Saturation–based model checker can terminate early and return a counterexample, in the form of a shortest path leading to an error state [13]. However, to compute such shortest counterexamples, edge-valued decision diagrams are employed, which may be thought of as extended form of MDDs that stores, for each globally reachable state, its distance from the initial states.

8 Conclusions and future work

The novel Saturation algorithm presented in this article proves that symbolic techniques based on decision diagrams can be successfully applied to efficiently generating state spaces of event-based concurrent systems with interleaving semantics. The key for achieving efficiency is to systematically exploit the local effects of firing events. First, the Saturation algorithm employs *Multi–valued Decision Diagrams* (MDDs) which naturally reflect the structure of the state vectors defined by a component-based, concurrent system model. Second, it partitions the model's *Kronecker–consistent* next–state function by both event and subsystem and stores it as a collection of sparse boolean matrices. This enables lightweight operations to manipulate MDDs, which drastically improves the time–efficiency of state-space generation when compared to traditional approaches that rely on iterating a single, heavyweight next–state function.

Both MDDs and the partitioned next–state function are the prerequisite for our unique iteration order, *Saturation*, that efficiently computes the reachable state space of an event-based concurrent system model with interleaving semantics. Indeed, symbolic state-space generation is not inherently breadth–first, as is widely believed [1]. Instead, Saturation fires all events affecting a given MDD node exhaustively and saturates nodes in a bottom–up fashion. This not only avoids duplicating work at lower MDD levels but also brings nodes into their final shape quickly. Most importantly and in contrast to classical BDD–based state-space generators, Saturation does not constantly create nodes that become disconnected later on. Hence, the peak memory consumption of Saturation drastically improves on earlier work.

Our experimental results testify to the cumulative effect of systematically exploiting the interleaving semantics in event-based concurrent system models, by consistently showing improvements of several orders of magnitude in run–time efficiency—from hours to seconds—and peak memory consumption—from megabytes to kilobytes—when compared to traditional BDD–based state-space generators. Moreover, our evaluation shows that by far the largest efficiency improvements result from our novel iteration order. While being very helpful in presenting our algorithm, the use of MDDs and Kronecker–consistency is not essential [28] and contributes comparatively little to the achieved performance gains.

8.1 Future work

Future work should proceed along several directions. First, we intend to develop heuristics for *partitioning* large systems specified by Petri nets into subsystems. As our experimental studies have shown, the size and order of the submodels in the partition have a significant impact on the compactness, i.e., the size of their MDD representation, in much the same way as the variable ordering has on the compactness of the BDD representation. It is worth investigating how existing variable re–ordering algorithms for BDDs can be adapted to our setting.

The second direction is to interface our Saturation algorithm to other modeling languages, most importantly the language *Promela* used in Holzmann's *Spin* tool [23]. This would enable a better comparison of the saturation algorithm to the Spin model checker, which is an explicit–state model checker that exploits interleaving semantics via *partial–order reduction* techniques. We leave it to future work to check whether our idea of node saturation can be successfully combined with partial–order reduction techniques. In addition, the Promela language includes advanced data structures such as message queues. It remains an interesting research question as to how those data structures can best be mapped onto MDD variables.

The third direction concerns a more extensive comparison of SMART to Spin. To do so, one could write a translator from Spin's Promela language to SMART's input language. However, writing such a translator would not automatically enable a fair comparison. This is because a translator, when fed a well–designed Promela model, does not necessarily produce a Petri net and a partition that are in the right form in order for Saturation to be efficient. Writing "fair" translators is a difficult craft and requires much further research.

The fourth and final direction for future research aims at *parallelizing* the Saturation algorithm. Many approaches to parallelizing symbolic model checkers utilize the huge main memory available in parallel machines in order to store larger state spaces, but have seen the model–checking problem shift from a memory–bound to a time–bound problem. For model checkers to be used by engineers working under strict deadlines, time efficiency is of paramount importance. We believe that the locality of manipulating MDD nodes in our Saturation algorithm might prove to be the key for achieving efficient parallelizations of symbolic model checkers for verifying communications protocols and distributed software. Preliminary findings on parallelizing Saturation are reported in [18].

References

1. Alur R, Brayton R, Henzinger T, Qadeer S, Rajamani S (2001) Partial-order reduction in symbolic state-space exploration. Formal Methods Syst Design 18:97–116
2. Barner S, Rabinovitz I (2003) Efficient symbolic model checking of software using partial disjunctive partitioning. In Correct Hardware Design and Verification Methods, LNCS 2860, Springer-Verlag, pp 35–50
3. Bryant RE (1986) Graph-based algorithms for boolean function manipulation. IEEE Trans Comp 35(8):677–691
4. Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang L (1992) Symbolic model checking: 10^{20} states and beyond. Inf Comput 98(2):142–170
5. Burch JR, Clarke EM, Long DE (1991) Symbolic model checking with partitioned transition relations. In Very Large Scale Integration. IFIP Transactions, North-Holland, pp 49–58
6. Burch JR, Clarke EM, Long DE, McMillan KL, Dill DL (1994) Symbolic model checking for sequential circuit verification. IEEE Trans Comput-Aided Design Integrated Circuits Syst 13(4):401–424
7. Cabodi G, Camurati P, Quer S (1999) Improving symbolic traversals by means of activity profiles. In Design Automation Conference. IEEE Comp Soc Press, pp 306–311
8. Ciardo G, Jones RL, Miner AS, Siminiceanu R (2003) Logical and stochastic modeling with SMART. In Modeling Techniques and Tools for Computer Performance Evaluation, LNCS 2794. Springer-Verlag, pp 78–97
9. Ciardo G, Lüttgen G, Siminiceanu R (2000) Efficient symbolic state-space construction for asynchronous systems. In Applications and Theory of Petri Nets, LNCS 1825. Springer-Verlag, pp 103–122
10. Ciardo G, Lüttgen G, Siminiceanu R (2001) Saturation: An efficient iteration strategy for symbolic state space generation. In Tools and Algorithms for the Construction and Analysis of Systems, LNCS 2031. Springer-Verlag, pp 328–342
11. Ciardo G, Marmorstein R, Siminiceanu R (2003) Saturation unbound. In Tools and Algorithms for the Construction and Analysis of Systems, LNCS 2619. Springer-Verlag, pp 379–393
12. Ciardo G, Siminiceanu R (2003) Structural symbolic CTL model checking of asynchronous systems. In Computer-Aided Verification, LNCS 2725. Springer-Verlag, pp 40–53

13. Ciardo G, Siminiceanu R (2002) Using edge-valued decision diagrams for symbolic generation of shortest paths. In Formal Methods in Computer-Aided Design, LNCS 2517. Springer-Verlag, pp 256–273
14. Ciardo G, Trivedi KS (1993) A decomposition approach for stochastic reward net models. Performance Evaluation 18(1):37–59
15. Cimatti A, Clarke EM, Giunchiglia F, Roveri M (1999) NuSMV: A new Symbolic Model Verifier. In Computer-Aided Verification, LNCS 1633. Springer-Verlag, pp 495–499
16. Clarke EM, Grumberg O, Peled DA (1999) Model Checking. MIT Press
17. Dolev D, Klawe M, Rodeh M (1982) An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. J Algorithms 3(3):245–260
18. Ezekiel J, Lüttgen G (2006) Can Saturation be parallelised? On the parallelisation of a symbolic state-space generator. In Parallel and Distributed Methods in Verification, LNCS. Springer-Verlag, To appear.
19. Fernandes P, Plateau B, Stewart WJ (1998) Efficient descriptor-vector multiplication in stochastic automata networks. J ACM 45(3):381–414
20. Fujita M, Fujisawa H, Matsunaga Y (1993) Variable ordering algorithms for ordered binary decision diagrams and their evaluation. IEEE Trans Comput-Aided Design Integrated Circuits Syst 12(1):6–12
21. Geldenhuys J, Valmari A (2001) Techniques for smaller intermediary BDDs. In Concurrency Theory, LNCS 2154. Springer-Verlag, pp 233–247
22. Graf S, Steffen B, Lüttgen G (1996) Compositional minimisation of finite state systems using interface specifications. Formal Aspects Comput 8(5):607–616
23. Holzmann G (2003) The Spin Model Checker: Primer and Reference Manual. Addison Wesley
24. Kam T, Villa T, Brayton R, Sangiovanni-Vincentelli A (1998) Multi-valued decision diagrams: Theory and applications. Multiple-Valued Logic 4(1–2):9–62
25. Kemper P (1996) Numerical analysis of superposed GSPNs. IEEE Trans. Software Eng. 22(4):615–628
26. Kimura S, Clarke EM (1990) A parallel algorithm for constructing binary decision diagrams. In Computer Design. IEEE Comp Soc Press, pp 220–223
27. McMillan KL (1992) Symbolic model checking: An approach to the state explosion problem. PhD thesis, Carnegie Mellon University. CMU–CS–92–131
28. Miner AS (2006) Saturation for a general class of models. IEEE Trans Software Eng 32(8):559–570
29. Miner AS, Ciardo G (1999) Efficient reachability set generation and storage using decision diagrams. In Applications and Theory of Petri Nets, LNCS 1639. Springer-Verlag, pp 6–25
30. Murata T (1989) Petri Nets: Properties, analysis and applications. Proc IEEE 77(4):541–579
31. Narayan A, Isles AJ, Jain J, Brayton RK, Sangiovanni-Vincentelli A (1997) Reachability analysis using Partitioned-ROBDDs. In Computer-Aided Design. ACM and IEEE Comp Soc Press, pp 388–393
32. Pastor E, Roig O, Cortadella J, Badia R (1994) Petri net analysis using boolean manipulation. In Applications and Theory of Petri Nets, LNCS 815. Springer-Verlag, pp 416–435
33. Plateau B (1985) On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In ACM SIGMETRICS. ACM, pp 147–153
34. Ravi K, Somenzi F (1999) Hints to accelerate symbolic traversal. In Correct Hardware Design and Verification Methods, LNCS 1703. Springer-Verlag, pp 250–264
35. Roig O, Cortadella J, Pastor E (1995) Verification of asynchronous circuits by BDD-based model checking of Petri nets. In Applications and Theory of Petri Nets, LNCS 935. Springer-Verlag, pp 374–391
36. Sanders WH, Malhis LM (1992) Dependability evaluation using composed SAN-based reward models. J Parallel Distributed Comput 15(3):238–254
37. Somenzi F (2001) CUDD: CU Decision Diagram Package, Release 2.3.1. University of Colorado at Boulder.
38. Tilgner M, Takahashi Y, Ciardo G (1996) SNS 1.0: Synchronized Network Solver. In Manufacturing and Petri Nets, pp 215–234
39. Valmari A (1990) A stubborn attack on the state explosion problem. In Computer-Aided Verification. AMS, pp 25–42
40. Yoneda T, Hatori H, Takahara A, Minato S-I (1996) BDDs vs. zero-suppressed BDDs for CTL symbolic model checking of Petri nets. In Formal Methods Comput Aided Design, LNCS 1166. Springer-Verlag, pp 435–449