

Implicit data structures for logic and stochastic systems analysis

Gianfranco Ciardo¹

Andrew S. Miner²

Abstract

Both logic and stochastic analysis have strong theoretical underpinnings, but they have been traditionally relegated to separate areas of computer science, the former focusing on logic and discrete algorithms, the latter on exact or approximate numerical methods. In the last few years, though, there has been a convergence of research in these two areas, due to the realization that data structures used in one area can benefit the other and that, by merging the goals of the two areas, a more integrated approach to system analysis can be derived. In this paper, we describe some of the beneficial interactions between the two, and some of the research challenges ahead.

1 Introduction

In this section, we introduce some notions related to the logical analysis of discrete-state systems, in particular CTL model checking, and to the numerical solution of Markov models, traditionally used for performance and reliability evaluation.

1.1 Discrete-state systems

Many interesting systems, including synchronous and asynchronous circuits, computer systems, communication networks, and manufacturing systems, fall into the category of discrete-state systems. Broadly speaking, these can be described by a collection of discrete state variables, whose *local states* collectively define the (*global*) *state* of the system, and by a set of *events* that, when *enabled* in a state, can occur, or *fire*, causing a change of state. If several actions can occur in a given state, the choice of which event fires first can be made either nondeterministically or according to some probability distribution. Formally, a discrete-state system is defined by:

- A set of possible global states, $\widehat{\mathcal{S}}$. We assume that a global state consists of K local states, $\widehat{\mathcal{S}} = \mathcal{S}_K \times \dots \times \mathcal{S}_1$, where $\mathcal{S}_k = \{0, 1, \dots, n_k - 1\}$ is the set of possible values for the k^{th} local state variable x_k .
- A non-empty set of initial states, $\mathcal{S}^{\text{init}} \subseteq \widehat{\mathcal{S}}$. The system begins in some state $\mathbf{s}^{\text{init}} \in \mathcal{S}^{\text{init}}$; if $\mathcal{S}^{\text{init}}$ contains several states, \mathbf{s}^{init} can again be selected either nondeterministically or according to some probability distribution.

¹Department of Computer Science and Engineering, University of California, Riverside, ciardo@cs.ucr.edu. Work supported in part by the National Science Foundation under grants CNS-0501748 and CNS-0501747.

²Department of Computer Science, Iowa State University, asminer@cs.iastate.edu

- A finite set of events \mathcal{E} .
- A function $\mathcal{N}_e : \widehat{\mathcal{S}} \rightarrow 2^{\widehat{\mathcal{S}}}$ describing, for each event $e \in \mathcal{E}$, the possible state changes due to e . If $\mathcal{N}_e(\mathbf{s}) = \emptyset$, e is not enabled in state \mathbf{s} . The *one-step reachability* function $\mathcal{N} : \widehat{\mathcal{S}} \rightarrow 2^{\widehat{\mathcal{S}}}$, which specifies the set of states reachable in one step from the current state, is then $\mathcal{N}(\mathbf{s}) = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e(\mathbf{s})$. For convenience, we apply this function also to sets of states, i.e., $\mathcal{N}(X) = \bigcup_{\mathbf{s} \in X} \mathcal{N}(\mathbf{s})$.

$\mathcal{S}^{\text{init}}$ and \mathcal{N} implicitly define the (*actual*) *state space* of the system, $\mathcal{S} = \mathcal{S}^{\text{init}} \cup \mathcal{N}(\mathcal{S}^{\text{init}}) \cup \mathcal{N}(\mathcal{N}(\mathcal{S}^{\text{init}})) \cup \dots = \mathcal{N}^*(\mathcal{S}^{\text{init}})$. From now on, we assume that \mathcal{S} is finite. In this case, building and storing \mathcal{S} is a conceptually simple task. In practice, however, it can be a major challenge due to its large size: this is the familiar *state explosion* problem.

1.2 Model checking discrete-state systems

Model checking is used to determine if certain desired behavioral properties are satisfied by a system. Properties are expressed in a suitable logic; different logics have different expressive power and more expressive logics are more computationally complex to verify. Temporal logics such as LTL (linear temporal logic) [29] and CTL (computation tree logic) [14] are often used, as they are fairly expressive and relatively straightforward to verify. These can be used to express properties such as *absence of deadlocks* or *responsiveness* (e.g., a service request is always eventually satisfied).

We focus on CTL, where properties of interest are expressed in terms of *state formulas* that can hold for a given state $\mathbf{s} \in \widehat{\mathcal{S}}$. State formulas can be combined with the usual boolean logic operators, \vee, \wedge, \neg , or with the ten operators AX, EX, AF, EF, AG, EG, AU, EU, AR, ER, which are obtained by pairing one of the two *path quantifiers*, A (for all paths) and E (for at least one path), with one of the five *temporal operators*, X (a property holds in the next state), F (a property holds eventually), G (a property holds always), U (a second property holds eventually and, before that, a first property always holds), and R, the dual of U (a second property holds either forever or until a state where it holds together with a first property). It can be shown that the ten operators can be expressed in terms of only the three operators EX, EG, and EU; thus, a CTL model checking tool must implement only these three operators.

In principle, the CTL model checking algorithm can generate the entire underlying state transition graph of the discrete-state system and recursively build sets of graph nodes, i.e., states, that satisfy each subformula of a CTL formula, starting from the *atomic* formulas whose truth value can be ascer-

tained by simply examining the individual state (e.g., $x_7 = 5$, or $x_2 > x_1 \wedge x_4 = x_1$). The operators EX, EU, and EG can be implemented using basic graph algorithms: the set of states satisfying EXp can be built by finding all states from which a transition to a state satisfying p is possible; the set of states satisfying Ep_1Up_2 can be built by performing a backward search of the graph from the states satisfying p_2 , adding only states that satisfy p_1 ; the set of states satisfying EGp can be built from cycles of states satisfying p (obtained from the strongly-connected components in the subgraph of states satisfying p) and performing a backward search from those cycles and adding only states that satisfy p . The system satisfies the CTL formula if the set of states satisfying the overall formula contains the initial states. However, this *explicit* approach is limited by the size of the state transition graph of the system, which can easily be too large in practice.

1.3 Discrete-state Markov systems

When actual timing aspects of the system are of interest, model checking alone is not enough. Rather, constant or random durations, or *firing times*, are attached to each event, and a *race semantic* is assumed, where the *remaining firing time (RFT)* of each enabled event decreases as time passes and the first event whose RFT elapses is the one to fire next. Furthermore, if the model admits the possibility of multiple RFTs elapsing at the same time, a *selection probability* must be specified to determine which event fires next. A particularly used assumption, on which we focus, is that of exponentially-distributed firing times. These give rise to the widely adopted formalism of continuous-time Markov chains (CTMCs), which admit conceptually simple solution algorithms for the study of the transient and long-term behavior.

Formally, a CTMC is a stochastic process $\{X_t : t \in \mathbb{R}\}$ with a discrete state space \mathcal{S} satisfying the memoryless property: $\forall r \geq 1, \forall t > t_r > \dots > t_1, \forall i, i_r, \dots, i_1 \in \mathcal{S}$,

$$\Pr\{X_t = i | X_{t_r} = i_r, \dots, X_{t_1} = i_1\} = \Pr\{X_t = i | X_{t_r} = i_r\}.$$

Considering only *homogeneous* CTMCs, the above probability depends on t and t_r only through the difference $h = t - t_r$, i.e., it equals $\Pr\{X_h = i | X_0 = i_r\}$. The state of a CTMC at time $t \geq 0$ is specified by a *probability vector* π_t , where $\pi_t[i] = \Pr\{X_t = i\}$. A homogeneous CTMC is then described by its *initial probability vector* π_0 , and by its *transition rate matrix* \mathbf{R} , defined by

$$\forall i, j \in \mathcal{S}, \mathbf{R}[i, j] = \begin{cases} 0 & \text{if } i = j \\ \lim_{h \rightarrow 0} \Pr\{X_h = j | X_0 = i\} / h & \text{if } i \neq j \end{cases}$$

or its *infinitesimal generator matrix* \mathbf{Q} , defined by,

$$\forall i, j \in \mathcal{S}, \mathbf{Q}[i, j] = \begin{cases} -\sum_{l \neq i} \mathbf{R}[i, l] & \text{if } i = j \\ \mathbf{R}[i, j] & \text{if } i \neq j \end{cases}.$$

The short-term, or *transient*, behavior of the CTMC is found by computing the transient probability vector π_t , which is the solution of the ordinary differential equation $d\pi_t/dt = \pi_t \mathbf{Q}$ with initial condition π_0 , thus it is given by the *matrix exponential* expression $\pi_t = \pi_0 e^{\mathbf{Q}t}$. The long-term, or *steady-state*,

behavior is found by computing the steady-state probability vector $\pi = \lim_{t \rightarrow \infty} \pi_t$; if the CTMC is *irreducible* (for finite \mathcal{S} , this means that \mathcal{S} is a strongly-connected component), π is independent of π_0 and is the unique solution of the homogeneous linear system $\pi \mathbf{Q} = \mathbf{0}$ subject to $\sum_{i \in \mathcal{S}} \pi[i] = 1$. The vector π or π_t is typically used to evaluate expected *instantaneous reward measures*: a reward function $r : \mathcal{S} \rightarrow \mathbb{R}$ specifies the rate at which a “reward” is generated in each state, and its expected value is computed as $\sum_{i \in \mathcal{S}} \pi[i] r(i)$.

It is also possible to evaluate *accumulated reward measures* over a time interval $[t_1, t_2]$, in either the transient ($t_1 < t_2 < \infty$) or the long term ($t_1 < t_2 = \infty$). The numerical algorithms and the issues they raise are similar to those for instantaneous rewards discussed above, thus we omit them for lack of space.

1.4 Numerical solution of discrete-state Markov systems

In practice, for exact steady-state analysis, the linear system $\pi \mathbf{Q} = \mathbf{0}$ is solved using iterative methods such as Jacobi or Gauss-Seidel, since the matrix \mathbf{Q} is typically extremely large and quite sparse. If \mathbf{Q} is stored by storing matrix \mathbf{R} , in sparse row-wise or column-wise format, and the diagonal of \mathbf{Q} , as a full vector, the operations required for these iterative solution methods are vector-matrix multiplications, i.e., vector-column (of a matrix) dot products. In addition to \mathbf{Q} , the solution vector π must be stored, and most iterative methods (such as Jacobi) require one or more auxiliary vectors of the same dimension as π , $|\mathcal{S}|$. For extremely large CTMCs, these auxiliary vectors may impose excessive memory requirements.

If matrix \mathbf{R} is stored in sparse column-wise format, and vector \mathbf{h} contains the expected “holding time” for each state, where $\mathbf{h}[i] = -1/\mathbf{Q}[i, i]$, then the Jacobi method can be written as in Fig. 1, where $\pi^{(new)}$ is an auxiliary vector and the matrix \mathbf{R} is accessed by columns, although the algorithm can be rewritten to access matrix \mathbf{R} by rows instead. The method of Gauss-Seidel, also shown in Fig. 1, is similar to Jacobi, except the newly computed vector entries are used immediately; thus only the solution vector π is stored, with newly computed entries overwriting the old ones. The Gauss-Seidel method can also be rewritten to access \mathbf{R} by rows, but this is not straightforward, and requires an auxiliary vector [16].

For transient analysis, the *uniformization* method is most often used, as it is numerically stable and uses vector-matrix multiplication as its primary operation (Fig. 1). The CTMC is *uniformized* with a rate $q \geq \max_{i \in \mathcal{S}} \{\|\mathbf{Q}[i, i]\|\}$ to obtain a discrete-time Markov chain with transition probability matrix $\mathbf{P} = \mathbf{Q}/q + \mathbf{I}$. The number of iterations M must be large enough to ensure that $\sum_{k=0}^M e^{-qt} (qt)^k / k!$ is very close to 1.

The remainder of the paper is organized as follows: Section 2 presents key technologies that have allowed substantial, but separate, advancements in the area of logic and stochastic analysis of discrete-state (Markov) systems. Section 3 discusses how the two areas have a common intersection and can improve each other. Finally, Section 4 concludes by examining some of the fundamental challenges that still remain.

<p><i>Jacobi</i>(in: $\pi^{(old)}$, \mathbf{h}, \mathbf{R}; out: $\pi^{(new)}$) is</p> <ol style="list-style-type: none"> 1 repeat 2 for $j = 1$ to \mathcal{S} 3 $\pi^{(new)}[j] \leftarrow \mathbf{h}[j] \cdot \sum_{i: \mathbf{R}[i,j] > 0} \pi^{(old)}[i] \cdot \mathbf{R}[i, j];$ 4 $\pi^{(new)} \leftarrow \pi^{(new)} / (\pi^{(new)} \cdot \mathbf{1});$ 5 $\pi^{(old)} \leftarrow \pi^{(new)};$ 6 until “converged”; 7 return $\pi^{(new)};$
<p><i>GaussSeidel</i>(in: \mathbf{h}, \mathbf{R}; inout: π) is</p> <ol style="list-style-type: none"> 1 repeat 2 for $j = 1$ to \mathcal{S} 3 $\pi[j] \leftarrow \mathbf{h}[j] \cdot \sum_{i: \mathbf{R}[i,j] > 0} \pi[i] \cdot \mathbf{R}[i, j];$ 4 $\pi \leftarrow \pi / (\pi \cdot \mathbf{1});$ 5 until “converged”; 6 return $\pi;$
<p><i>Uniformization</i>(in: π_0, \mathbf{P}, q, t, M; out: π_t) is</p> <ol style="list-style-type: none"> 1 $\pi_t \leftarrow \mathbf{0};$ 2 $\gamma \leftarrow \pi_0;$ 3 <i>Poisson</i> $\leftarrow e^{-qt};$ 4 for $k = 1$ to M do 5 $\pi_t \leftarrow \pi_t + \gamma \cdot \text{Poisson};$ 6 $\gamma \leftarrow \gamma \cdot \mathbf{P};$ 7 <i>Poisson</i> $\leftarrow \text{Poisson} \cdot q \cdot t / k;$ 8 return $\pi_t;$

Figure 1: Numerical solution algorithms for CTMCs.

2 Key advancements in logic and stochastic analysis

2.1 Symbolic model checking

To overcome limitations of explicit approaches to model checking, researchers turned to *symbolic* approaches, which utilize binary decision diagrams (BDDs) [4] and their variants, following the initial success of [9]. Instead of BDDs, which encode boolean functions on multidimensional boolean domains, we present multivalued decision diagrams (MDDs) [21], which encode boolean functions on the domain $\widehat{\mathcal{S}}$ we already introduced in Sect. 1.3. A K -level MDD on the domain $\widehat{\mathcal{S}} = \mathcal{S}_K \times \dots \times \mathcal{S}_1$ is a directed acyclic graph *rooted* at a node r , such that each non-terminal node is labeled with a level $k \in \{K, \dots, 1\}$ and has $n_k = |\mathcal{S}_k|$ outgoing edges labeled with the values of \mathcal{S}_k , respectively, while the terminal nodes, labeled with level 0, can only be the special nodes 0 or 1, corresponding to the boolean functions 0 and 1. Let $lvl(p)$ be the level of node p , and let $p[i]$ be the node pointed by the i^{th} -edge of node p , for $i \in \mathcal{S}_k$. We restrict ourselves to reduced ordered MDDs: an MDD is *ordered* if $lvl(p) > lvl(p[i])$ for each $i \in \mathcal{S}_k$, and is *reduced* if it contains no *redundant* node p (such that $p[0] = \dots = p[n_k - 1]$) nor *duplicate* nodes p and q (such that $lvl(p) = lvl(q)$ and $p[i] = q[i]$, for each $i \in \mathcal{S}_k$).

An MDD encodes a function $f_r: \widehat{\mathcal{S}} \rightarrow \{0, 1\}$, according to the definition of the function encoded by a node p at level k :

$$f_p(x_K, \dots, x_1) = \begin{cases} f_q(x_K, \dots, x_1) & \text{if } k > 0 \text{ and } p[x_k] = q \\ p & \text{if } k = 0 \end{cases}.$$

MDDs are *canonical*: two boolean functions over the same domain $\widehat{\mathcal{S}}$ encoded using MDDs with *shared nodes* to avoid duplicate nodes, are identical iff they have the same root.

Given a discrete-state system with potential state space $\widehat{\mathcal{S}}$,

<p><i>BfsGen</i>(\mathcal{S}^{init}, \mathcal{N})</p> <ol style="list-style-type: none"> 1 $\mathcal{S} \leftarrow \mathcal{S}^{init};$ 2 $\mathcal{U} \leftarrow \mathcal{S}^{init};$ 3 while $\mathcal{U} \neq \emptyset$ do 4 $\mathcal{X} \leftarrow \mathcal{N}(\mathcal{U});$ 5 $\mathcal{U} \leftarrow \mathcal{X} \setminus \mathcal{S};$ 6 $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{U};$ 7 return $\mathcal{S};$ 	<ul style="list-style-type: none"> •known states •unexplored known states •there are still unexplored states •possibly new states •truly new states
--	--

Figure 2: An MDD-based breadth-first generation algorithm.

<p><i>SymbolicEX</i>($\mathcal{N}^{-1}(\mathcal{P})$)</p> <ol style="list-style-type: none"> 1 return $\mathcal{N}^{-1}(\mathcal{P});$ 	<ul style="list-style-type: none"> •go one step backward
<p><i>SymbolicEU</i>($\mathcal{N}^{-1}(\mathcal{Q}, \mathcal{P})$)</p> <ol style="list-style-type: none"> 1 $\mathcal{X} \leftarrow \mathcal{P};$ 2 $\mathcal{U} \leftarrow \mathcal{X};$ 3 while $\mathcal{U} \neq \emptyset$ do 4 $\mathcal{U} \leftarrow (\mathcal{N}^{-1}(\mathcal{U} \cap \mathcal{Q}) \setminus \mathcal{X});$ 5 $\mathcal{X} \leftarrow \mathcal{X} \cup \mathcal{U};$ 6 return $\mathcal{X};$ 	<ul style="list-style-type: none"> •start from the states satisfying p •unexplored states •go backward enforcing q •add the new states to explore
<p><i>SymbolicEG</i>($\mathcal{N}^{-1}(\mathcal{P})$)</p> <ol style="list-style-type: none"> 1 $\mathcal{X} \leftarrow \mathcal{P};$ 2 repeat 3 $\mathcal{O} \leftarrow \mathcal{X};$ 4 $\mathcal{X} \leftarrow \mathcal{N}^{-1}(\mathcal{X}) \cap \mathcal{P};$ 5 until $\mathcal{O} = \mathcal{X};$ 6 return $\mathcal{X};$ 	<ul style="list-style-type: none"> •start from the states satisfying p •go backward, eliminate states

Figure 3: Symbolic implementation of the CTL operators.

we can store any set of global states $\mathcal{X} \subseteq \widehat{\mathcal{S}}$ by encoding its indicator function $f_r(x_K, \dots, x_1)$, which evaluates to 1 iff $(x_K, \dots, x_1) \in \mathcal{X}$, in a K -level MDD rooted at r . Analogously, we can store any relation over $\widehat{\mathcal{S}}$, or function from $\widehat{\mathcal{S}}$ to $2^{\widehat{\mathcal{S}}}$, such as \mathcal{N} , in a $2K$ -level MDD. Then, we can generate the reachability set \mathcal{S} using the symbolic algorithm of Fig. 2. Starting from the set \mathcal{S}^{init} , we compute the set of new states reachable in one step, and iterate until the set \mathcal{U} of unexplored states is empty. Of course, \mathcal{S} , \mathcal{U} , \mathcal{X} , and \mathcal{N} are encoded by MDDs. Operations on sets (e.g., $\mathcal{P} \cup \mathcal{R}$) become boolean operations on indicator functions (e.g., $f_p \vee f_r$), and are implemented by recursive traversals of the corresponding MDDs; this can be done with complexity $O(|p| \cdot |r|)$, where $|p|$ indicates the number of nodes for the MDD rooted at p .

Just like symbolic state-space generation, symbolic model checking of CTL formulas is done using $2K$ -level MDDs to store relations between states and K -level MDDs to store sets of states. However, instead of “moving forward”, the algorithms to compute *EXp*, *EqUp*, and *EGp* (Fig. 3), start from \mathcal{P} , the set of states satisfying property p , and “move backward”, thus they require \mathcal{N}^{-1} , the inverse of \mathcal{N} : $(i_K, \dots, i_1, j_K, \dots, j_1) \in \mathcal{N}^{-1}$ iff $(j_K, \dots, j_1, i_K, \dots, i_1) \in \mathcal{N}$.

2.2 Kronecker-based approaches

Also in the mid ’80s, researchers working on discrete-state Markov systems that produce enormous CTMCs turned to implicit representations for the matrix \mathbf{Q} . A successful idea championed by Plateau [28], based on a representation of \mathbf{Q} in terms of Kronecker products [15], has received much attention in the last decade [6, 8, 18].

The key to the Kronecker approach is to represent matrix $\widehat{\mathbf{R}}_e$, which describes the contribution of event e to transition rate matrix $\widehat{\mathbf{R}}$ for a CTMC with state space $\widehat{\mathcal{S}}$, in a structured way:

$$\widehat{\mathbf{R}}_e[\mathbf{i}, \mathbf{j}] = \prod_{k=K}^1 \mathbf{R}_{e,k}[i_k, j_k]$$

where $\mathbf{i} = (i_K, \dots, i_1) \in \widehat{\mathcal{S}}$ and $\mathbf{j} = (j_K, \dots, j_1) \in \widehat{\mathcal{S}}$. If the matrices $\mathbf{R}_{e,k}$, of size $|\mathcal{S}_k| \times |\mathcal{S}_k|$, have constant entries, then we say event e is *Kronecker-consistent*, and $\widehat{\mathbf{R}}_e$ can be expressed as the ordinary Kronecker product $\widehat{\mathbf{R}}_e = \mathbf{R}_{e,K} \otimes \dots \otimes \mathbf{R}_{e,1}$; otherwise, some $\mathbf{R}_{e,k}$ contains *functional* elements and $\widehat{\mathbf{R}}_e$ must be expressed using a generalized Kronecker product [19]. The total storage required for matrices $\mathbf{R}_{e,k}$ is much less than for $\widehat{\mathbf{R}}$, but the numerical solution poses several challenges when \mathbf{R} , or, rather, $\widehat{\mathbf{R}}$ is stored using a Kronecker approach.

Diagonal of $\widehat{\mathbf{Q}}$. The diagonal \mathbf{q} of $\widehat{\mathbf{Q}}$ can be expressed as the sum of Kronecker products, $\mathbf{q} = -\sum_{e \in \mathcal{E}} \otimes_{k=K}^1 \mathbf{r}_{e,k}$, where $\mathbf{r}_{e,k}[i_k] = \sum_{j_k \in \mathcal{S}_k, j_k \neq i_k} \mathbf{R}_{e,k}[i_k, j_k]$. Vector \mathbf{q} can either be stored explicitly and computed once, to speed up numerical solution, or computed as needed, to reduce memory requirements.

Local events. Event e *depends* on state variable x_k if its enabling depends on x_k , its firing changes x_k , or its firing time depends on x_k ; when none of these conditions holds, $\mathbf{R}_{e,k}$ is an identity matrix. An important special case is when an event e depends only on x_k ; then, all such events *local* to level k can be merged into a single “macro-event”, since

$$(\mathbf{I} \otimes \mathbf{R}_{e_1,k} \otimes \mathbf{I}) + (\mathbf{I} \otimes \mathbf{R}_{e_2,k} \otimes \mathbf{I}) = \mathbf{I} \otimes (\mathbf{R}_{e_1,k} + \mathbf{R}_{e_2,k}) \otimes \mathbf{I}.$$

Exploiting this property and using Kronecker sums to deal with local events improves the solution efficiency [6, 28].

Potential vs. actual-sized vectors. In principle, the numerical solution of $\widehat{\pi} \widehat{\mathbf{Q}} = \mathbf{0}$, where $\widehat{\pi}$ and any other auxiliary vector is of size $|\widehat{\mathcal{S}}|$, is straightforward. Jacobi or Gauss-Seidel only need an appropriate algorithm for vector-matrix or vector-column multiplication [6]. Furthermore, since the real state space of the CTMC is $\mathcal{S} \subseteq \widehat{\mathcal{S}}$, vector $\widehat{\pi}$ must be initialized so that $\widehat{\pi}[\mathbf{i}] = 0$ when $\mathbf{i} \notin \mathcal{S}$, this guarantees that the same property holds upon convergence. However, this simple approach has high computational and storage costs when $|\widehat{\mathcal{S}}| \gg |\mathcal{S}|$. Using instead vectors of size $|\mathcal{S}|$ can save much storage, but introduces additional challenges: Jacobi and Gauss-Seidel must visit only *reachable* rows and columns of $\widehat{\mathbf{Q}}$, and the indexing difference between $\widehat{\pi}$ and π must be accounted for. The resulting vector-matrix and vector-column multiplication algorithms are considerably more complex [6].

3 Cross-fertilization

3.1 Using Kronecker for symbolic model checking

A Kronecker encoding of \mathcal{N} (using boolean-valued, instead of real-valued, elements for $\mathbf{R}_{e,k}$) has been used to improve *explicit* reachability set generation [22]. For symbolic reachability set generation, a Kronecker encoding of \mathcal{N} was first used, instead of a $2K$ -level decision diagram, in [26], where

the authors also exploited the presence of *local events* for a more efficient reachability set generation.

This was later extended and improved in [10], which defines $Top(e) = \max\{k : \mathbf{R}_{e,k} \neq \mathbf{I}\}$, the highest level on which each event e depends, and partitions set \mathcal{E} into (up to) K classes,

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

The *saturation* algorithm [10] is then based restricting \mathcal{N} to events that depend only on levels k or below:

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

The algorithm exploits the fact that $\mathcal{N}_{\leq k}$ can be directly applied to any level- k node, since these events are unaffected by nodes above level k . By repeatedly applying $\mathcal{N}_{\leq k}$ to a level- k node as soon as it is created, nodes are *saturated* before they are connected to the MDD. Starting from the bottom node(s) at level 1, nodes are then saturated recursively, so that all descendants of the node being saturated are themselves guaranteed to be already saturated. The reachability set is then encoded by the root node of the MDD encoding S^{init} , once this node has been saturated. This strategy often achieves savings of several orders of magnitude in both time and storage, since only saturated nodes are added to unique tables and operation caches, and not-yet-saturated nodes are updated “in-place”.

The saturation algorithm has been applied to other symbolic computations. In [13], it is used to improve the efficiency of computing the CTL operators EF (essentially, this is state-space generation using \mathcal{N}^{-1} instead of \mathcal{N} , where the Kronecker encoding of \mathcal{N}^{-1} is the same as that of \mathcal{N} , except that it uses the transpose matrices $\mathbf{R}_{e,k}^T$) and EU (in addition, this requires us to distinguish between “safe” and “unsafe” events, and interleave safe saturation with unsafe breadth-first iterations). In [12], it is applied to *edge-valued MDDs* to generate and store the *distance function* $\delta(\mathbf{j}) = \min\{n : \mathbf{j} \in \mathcal{N}^n(S^{init})\}$, for all $\mathbf{j} \in \widehat{\mathcal{S}}$, as the fixpoint of $\delta(\mathbf{j}) = \min\{\delta(\mathbf{j}), \min\{1 + \delta(\mathbf{i}) \mid \mathbf{j} \in \mathcal{N}(\mathbf{i})\}\}$. In turn, δ is then used to efficiently generate shortest-length *witnesses* to CTL EF queries.

As described above, saturation uses a Kronecker representation of \mathcal{N} , which means that each \mathcal{N}_e must be Kronecker-consistent. If this is not the case for an event e , either e can be split into sub-events e_1, \dots, e_l , each of them Kronecker-consistent, or state variables can be combined until e becomes Kronecker-consistent. However, either way to achieve Kronecker consistency can be expensive in terms of memory, time, or both (for example, if we combine many state variables, we approach an explicit enumeration of the states). To overcome this problem, (boolean) *matrix diagrams* (MxDs) were developed [24]. These combine MDDs and boolean Kronecker matrices: a level- k MxD node contains a matrix of $|\mathcal{S}_k| \times |\mathcal{S}_k|$ downward pointers; more notable, though, is that, instead of eliminating redundant nodes, *identity nodes* (corresponding to identity matrices) are eliminated. An MxD encoding of \mathcal{N} allows it to be automatically partitioned into sets $\mathcal{N}_{\leq k}$, even when the relation \mathcal{N} is not Kronecker-consistent [25], thus the saturation algorithm can always be used.

3.2 Using matrix diagrams for Markov analysis

Just as boolean MxDs were developed to improve the Kronecker representation of \mathcal{N} , *edge-valued* MxDs were developed to improve the Kronecker representation of \mathbf{Q} , specifically, of the numerical solution [11]. Edge-valued MxDs assign a real-valued label to each downward pointer from a level- k node, which is analogous to the value of an entry in a matrix $\mathbf{R}_{e,k}$; then, the value of an element in the matrix encoded by an edge-valued MxD is obtained by traversing the MxD and multiplying the edge values along the path. Since a Kronecker product is a special case of a MxD where all downward edges from level k point to the same node at level $k - 1$, building an edge-valued MxD from a Kronecker representation of $\widehat{\mathbf{Q}}$ is straightforward. However, using appropriate MxD operations, the “unreachable” rows and columns of $\widehat{\mathbf{Q}}$ can be filled with zeroes, eliminating one source of overhead with Kronecker-based numerical solution. Furthermore, a more efficient vector-column multiplication algorithm is possible with edge-valued MxDs, which *caches* previously-built matrix columns (an idea borrowed from MDD manipulation algorithms). Finally, an *edge-valued* MDD-based structure [11, 12] can be used not only to store \mathcal{S} , but also to resolve the indexing difference between $\widehat{\pi}$ and π , by efficiently computing the lexicographic index in \mathcal{S} of a state \mathbf{i} .

3.3 CSL

Continuous stochastic logic (CSL) [1, 2] is analogous to CTL except it includes (real) timing and probability information, and is applied to a CTMC instead of a state transition graph. For instance, given a state formula ϕ , a probability $\alpha \in [0, 1]$, and a comparison operator $\Delta \in \{<, \leq, \geq, >\}$, the state formula $\mathbf{S}_{\Delta\alpha}(\phi)$ is true for a state s if, given that the CTMC begins in state s , the steady-state probability of being in a state where ϕ holds is $\Delta\alpha$. The state formula $\mathbf{P}_{\Delta\alpha}(\psi)$ is true for state s if, when the CTMC begins in state s , the probability of *path formula* ψ holding is $\Delta\alpha$. Intuitively, path quantifiers \mathbf{E} and \mathbf{A} are equivalent to $\mathbf{P}_{>0}$ and $\mathbf{P}_{\geq 1}$, respectively, assuming appropriate fairness constraints to avoid the complication of non-empty sets of paths having probability measure 0.

Similarly, the temporal operators \mathbf{X} and \mathbf{U} are given bounds, which are nonempty time intervals $I \subseteq \mathbb{R}$: path formula $\mathbf{X}_I\phi$ holds along paths where the second state satisfies ϕ and the transition from the first state to the second state occurs in time $t \in I$, and path formula $\phi_1\mathbf{U}_I\phi_2$ holds along paths where ϕ_2 is satisfied at some time instant $t \in I$ along a path where ϕ_1 holds before time t . Again, note that “ordinary” \mathbf{X} and \mathbf{U} are equivalent to $\mathbf{X}_{(0,\infty)}$ and $\mathbf{U}_{(0,\infty)}$, respectively.

Algorithms to check CSL specifications [2] combine concepts from untimed logic model checking algorithms and numerical analysis of CTMCs. For instance, we can compute the probability of $\phi_1\mathbf{U}_{[0,t]}\phi_2$ by recursively determining states that satisfy ϕ_1 and ϕ_2 , then merging the states that satisfy ϕ_2 into an “absorbing good macrostate” and states that satisfy neither ϕ_1 nor ϕ_2 into an “absorbing bad macrostate”, then computing π_t for the resulting CTMC, and extracting the probability of the good macrostate.

4 Challenges

The previous section introduced several exciting advances that greatly improve the efficiency of symbolic model checking (saturation), reduce the memory requirements to store a CTMC (matrix diagrams), or generalize our ability to discuss CTMC measures (CSL). However, several advancements are still needed to fully exploit the promise of symbolic methods.

4.1 Completely symbolic CTMC solution

Kronecker or, even better, MxDs can compactly encode enormous CTMCs. However, the best current numerical solution algorithms that use these symbolic encodings for \mathbf{Q} still employ explicit storage for the required vectors, including π . This severely limits the state-space size of practically solvable models to perhaps 10^8 or so, even on the largest workstations.

To overcome this limitation, a fully symbolic approach must be devised, where not only \mathbf{Q} but also π is stored using some form of decision diagrams. Both *multi-terminal binary decision diagrams* (MTBDDs) and *probabilistic decision diagrams* (PDGs) have been proposed for this purpose, but with limited success: an effective fully symbolic approach still eludes us. MTBDDs are a simple extension of BDDs where, instead of just 0 and 1, an arbitrary number of real-valued terminal nodes can be present. MTBDDs have been used to store \mathbf{Q} with reasonable success [23] when many entries in the matrix have the same value; but, unfortunately, in practice most entries of π are distinct, producing an MTBDD that is nearly a full tree and thus even more expensive to store than the full vector we are trying to avoid in the first place. The extension to multivalued instead of binary nodes is possible, but suffers from the same problem. PDGs are instead a form of edge-valued decision diagrams where the sum of the edge values leaving a node is exactly 1.0 [3] (they can be seen as a special case of MxDs where the nodes are 1×2 matrices instead of $|\mathcal{S}_k| \times |\mathcal{S}_k|$). While perhaps more promising than MTBDDs, PDGs also seem to suffer from a lack of structure when storing π in practice, and, again, switching to multivalued nodes (i.e., using $1 \times |\mathcal{S}_k|$ matrices) does not seem to help.

4.2 Symbolic methods for approximate CTMC solution

An approach to determine an *approximation* to π was investigated in [27], based on the MDD representation of \mathcal{S} : each downward pointer in the MDD has an associated real-valued edge label, and the value of $\pi[\mathbf{i}]$ is estimated as the product of the edge labels along the path through the MDD corresponding to \mathbf{i} . Based on the exact MDD structure of \mathcal{S} and Kronecker structure of \mathbf{Q} , an aggregated CTMC is built for each MDD level, whose states correspond to the downward MDD pointers; the edge labels corresponding to these downward pointers are obtained from solving this aggregated CTMC. Since the aggregated CTMC for each level can depend on the edge labels at all other levels, fixed-point iterations are required to obtain the edge labels. A similar but more adaptive approach is used in [5].

Extending this technique to arbitrary symbolic encodings such as MxDs should be possible, but it remains to be seen how this would affect the quality of the approximation. This raises interesting questions about the relation between the logical structure of the state space and the nature of the CTMC solution. For example, it is known that the algorithm of [27] provides exact results if the model being studied has a product-form solution. Finally, we note that established methods such as lumping and aggregation are only starting to be investigated in a symbolic context [17, 20].

4.3 Symbolic CSL algorithms

To date, all CSL implementations we are aware of are based on either an explicit or symbolic storage of the transition rate matrix and an explicit storage of the solution vector [2, 7, 23], thus the limitations on the size of the probability vector (hence of the state space) discussed in Sect. 4.1 remain. Indeed, this is even more so in the case of CSL, since several CTMCs might have to be solved in order to evaluate a CSL formula.

If approximations such as those discussed in Sect. 4.2 are employed, limitations on the size of the state space are lifted, but at the expense of having to cope with numerical uncertainties. For example, if, during the evaluation of a CSL formula, we seek to build the set of states satisfying a certain condition ϕ with probability $\geq \alpha$, how can we proceed if all we have is an approximation of the true probability vector π ? It is likely that the answer to this question holds the key to the wide acceptance of CSL as *the* language to discuss measures for CTMC models in practice.

References

[1] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model-checking continuous-time markov chains. *ACM Trans. Comput. Logic*, 1(1):162–170, 2000.

[2] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model checking algorithms for continuous-time Markov chains. *IEEE TSE*, 29(6):524–541, June 2003.

[3] M. Bozga and O. Maler. On the representation of probabilities over structured domains. In *Proc. CAV*, LNCS 1633, pages 261–273, July 1999. Springer-Verlag.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE TC*, 35(8):677–691, Aug. 1986.

[5] P. Buchholz. An adaptive decomposition approach for the analysis of stochastic Petri nets. In *Proc. DSN*, pages 647–656, June 2002.

[6] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. Comp.*, 12(3):203–222, 2000.

[7] P. Buchholz, J. P. Katoen, P. Kemper, and C. Tepper. Model-checking large structured Markov chains. *JLAP*, 56(1/2):69–97, 2003.

[8] P. Buchholz and P. Kemper. Numerical analysis of stochastic marked graphs. In *Proc. PNPM*, pages 32–41, Oct. 1995. IEEE CS Press.

[9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and

L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[10] G. Ciardo, G. Luetzgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In *Proc. TACAS*, LNCS 2031, pages 328–342, Apr. 2001. Springer-Verlag.

[11] G. Ciardo and A. S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In *Proc. PNPM*, pages 22–31, Sept. 1999. IEEE CS Press.

[12] G. Ciardo and R. Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In *Proc. FM-CAD*, LNCS 2517, pages 256–273, Nov. 2002. Springer-Verlag.

[13] G. Ciardo and R. Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. In *Proc. CAV*, LNCS 2725, pages 40–53, July 2003. Springer-Verlag.

[14] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

[15] M. Davio. Kronecker products and shuffle algebra. *IEEE TC*, C-30:116–125, Feb. 1981.

[16] D. D. Deavours and W. H. Sanders. “On-the-fly” solution techniques for stochastic Petri nets and extensions. In *Proc. PNPM*, pages 132–141, June 1997. IEEE CS Press.

[17] S. Derisavi, P. Kemper, and W. H. Sanders. Symbolic state-space exploration and numerical analysis of state-sharing composed models. *Linear Algebra and Its Applications*, 386C:137–166, 2004.

[18] S. Donatelli. Superposed generalized stochastic Petri nets: definition and efficient solution. In *Proc. ICATPN*, LNCS 815, pages 258–277, June 1994. Springer-Verlag.

[19] P. Fernandes, B. Plateau, and W. J. Stewart. Efficient descriptor-vector multiplication in stochastic automata networks. *JACM*, 45(3):381–414, 1998.

[20] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous-time Markov chains. In *Proc. NSMC*, pages 188–207, Sept. 1999. Prentice Hall, Zaragoza, Spain.

[21] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[22] P. Kemper. Numerical analysis of superposed GSPNs. *IEEE TSE*, 22(4):615–628, Sept. 1996.

[23] M. Z. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: a hybrid approach. *Software Tools for Technology Transfer*, 6(2):128–142, 2004.

[24] A. S. Miner. Efficient state space generation of GSPNs using decision diagrams. In *Proc. DSN*, pages 637–646, June 2002.

[25] A. S. Miner. Saturation for a general class of models. In *Proc. QEST*, pages 282–291, Sept. 2004.

[26] A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Proc. ICATPN*, LNCS 1639, pages 6–25, June 1999. Springer-Verlag.

[27] A. S. Miner, G. Ciardo, and S. Donatelli. Using the exact state space of a Markov model to compute approximate stationary measures. In *Proc. SIGMETRICS*, pages 207–216, June 2000.

[28] B. Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. SIGMETRICS*, pages 147–153, May 1985.

[29] A. Pnueli. The temporal logic of programs. In *Proc. FOCS*, pages 46–57. IEEE CS Press, Nov. 1977.