NUMERICAL SOLUTION of MARKOV CHAINS, p. 167–189

Symbolic State-space Exploration and Numerical Analysis of State-sharing Composed Models

* Salem Derisavi[1], Peter Kemper[2], William H. Sanders[1]

[1] *University of Illinois at Urbana-Champaign, Coordinated Science Laboratory, Urbana, IL 61801, USA, {derisavi,whs}@crhc.uiuc.edu*
[2] *Informatik IV, Universität Dortmund, D-44221 Dortmund, Germany, peter.kemper@udo.edu*

KEY WORDS: Multi-valued Decision Diagrams, Matrix Diagrams, Numerical Analysis, Symbolic State-space Exploration

ABSTRACT

The complexity of stochastic models of real-world systems is usually managed by abstracting details and structuring models in a hierarchical manner. Systems are often built by replicating and joining subsystems, making possible the creation of a model structure that yields lumpable state spaces. This fact has been exploited to facilitate model-based numerical analysis. Likewise, recent results on model construction suggest that decision diagrams can be used to compactly represent large Continuous Time Markov Chains (CTMCs). In this paper, we present an approach that combines and extends these two approaches. In particular, we propose methods that apply to hierarchically structured models with hierarchies based on sharing state variables. The hierarchy is constructed in a way that exposes structural symmetries in the constructed model, thus facilitating lumping. In addition, the methods allow one to derive a symbolic representation of the associated CTMC directly from the given model without the need to compute and store the overall state space or CTMC explicitly. The resulting representation of a generator matrix allows the analysis of large CTMCs in lumped form. The efficiency of the approach is demonstrated with the help of two example models.

1. Introduction

Model-based evaluation of computer and communication systems often takes place by simulation. In many cases, the desired results could, in principle, also be derived through

*Correspondence to: [1] University of Illinois at Urbana-Champaign, Coordinated Science Laboratory, Urbana, IL 61801, USA, derisavi@crhc.uiuc.edu

Contract/grant sponsor: This material is based upon work supported by the National Science Foundation under Grant Nos. CCR-00-86096 and INT-0233490 and by DFG, SFB 559, and the DAAD/NSF exchange project (PPP USA), No. D/0247256. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

analysis of generated CTMCs; however, in practice, the size of the systems of equations that would need to be solved is prohibitive. This "largeness problem" has motivated much research in the construction and numerical solution of CTMCs.

Many construction and solution techniques that have been developed for large CTMCs can be classified as "largeness avoidance" techniques in which certain properties of some representation of the model (ranging from the high-level description of the model to the underlying CTMC itself) are exploited to reduce the size (in number of states and transitions) of the underlying CTMC that needs to be solved to obtain a solution of the model. For example, state lumping is an approach that reduces the size of a CTMC by considering the quotient of the CTMC with respect to an equivalence relation that preserves the Markov property and many performance measures defined on the CTMC. Since the computation of that equivalence relation for a large CTMC is costly in space and time, most practical lumping approaches identify appropriate lumpings by operating on a higher-level formalism, rather than by constructing the unlumped CTMC and then operating on it. For some modeling formalisms, the equivalence that is used for lumping is established by the modeling formalism itself; for instance, this is the case for stochastic well-formed nets (SWNs) [5] and stochastic activity network-based composed models (SANs) [20]. It can also be shown that lumping has the property of a congruence that is preserved by parallel composition in a number of process algebra formalisms and stochastic automata, so approaches that make use of a compositional structure in stochastic process algebras can also be used to generate lumped overall state spaces (e.g., [2]).

Nevertheless, even a lumped state space can be extremely large, and further work on "largeness tolerance" techniques is needed to practically support such lumped state spaces. For example, binary and multi-valued decision diagrams (BDDs and MDDs) have been successfully applied to explore and represent large unlumped state spaces. The key idea is to encode states as paths in a directed acyclic graph. Techniques that generate state spaces using decision diagrams are referred to as *symbolic* state-space exploration and representation techniques (e.g., [10]), and in some cases, they allow one to verify logical properties of systems with "10^{20} states and beyond" [4].

For the numerical analysis of CTMCs, it is also necessary to represent the generator matrix $\mathbf{Q}$ in a space-efficient manner. Different approaches exist, and one possibility is to follow a divide-and-conquer strategy and represent the overall matrix $\mathbf{Q}$ by a set of relatively small component matrices that are appropriately combined. Such so-called Kronecker representations are built upon a specific matrix algebra whose operators (Kronecker product and sum) serve as composition operators to build $\mathbf{Q}$ from component matrices (e.g., [3, 18, 22]). In those approaches, the composition of a system from subsystems is built upon synchronization of actions, under certain assumptions about the structure of a model. Alternatively, component matrices can be combined using a suitable variant of a decision diagram, namely a matrix diagram (MD) (e.g., [7, 8, 9, 17]). MDs have been proposed for systems that again are built in a compositional manner upon synchronization of actions. Another approach is to employ multi-terminal binary decision diagrams (MTBDDs) that store $\mathbf{Q}(s, s')$ at the end of a path through a BDD, where the path itself encodes the transition (s, s') (e.g., [13]). MTBDDs do not rely on a given structure of a model, but they only perform well if there are not too many different entries in $\mathbf{Q}(s, s')$. Preliminary work on combining state-sharing and action synchronization between models for MTBDD-based analysis is reported in [15].

An important result of this paper is that it extends previous work on MDDs and MDs to composed models that share state variables and that support next-state and weight functions that are state-dependent in general[†]. Furthermore, it combines lumping techniques, which have been applied to state-sharing composed models, with largeness tolerance techniques that use MDDs and MDs. In particular, our efforts have resulted in a new algorithm that symbolically generates the state space of a hierarchical model (which is built using *join* and *replicate* operators [20]) in the form of an MDD data structure. The replicate operator imposes symmetries that create regular structures in the state space, and therefore make symbolic exploration of the state space efficient with MDDs.

We also designed an algorithm to obtain an MDD representation of the lumped state space from the MDD generated by the state-space generation algorithm. The lumping algorithm, which reduces the size of the state space, also reduces the regularity of the MDD, whose representation becomes larger as a result. However, that increase is negligible compared to the space used for an iteration vector in the subsequent numerical analysis of the lumped CTMC. We obtain an MD representation of the lumped CTMC as a projection of the MD of the unlumped CTMC on the lumped state space. In performing a numerical analysis on that MD, one must use extra care in matching states with their corresponding lumped states in the lumped CTMC.

The remainder of the paper is organized as follows. First, we begin in Section 2 with some definitions and notations we need to specify the modeling formalism we use and to describe MDDs and MDs. Then, in Section 3, we present a symbolic state-space exploration algorithm. Section 4 discusses how to obtain an MD for the generator matrix of the lumped CTMC and how to operate on that structure for numerical analysis. The proposed approach has been implemented and used for the numerical state-space analysis of a highly redundant fault-tolerant parallel computer system [16, 19]. We also consider a well-known performance model of a communication protocol [23]. Results for these models are presented in Section 5. We conclude in Section 6.

2. Background

2.1. Hierarchical Model Specification

In this paper, we develop a representation of the CTMC of a hierarchical composed model that is built on shared state variables (SVs) among submodels. This composition operation is the same as the one used in SAN-based reward models [20], but is different from action-synchronization composition, which has been used in superposed generalized stochastic Petri nets, (stochastic) process algebras, and stochastic automata networks. In order to describe precisely how hierarchical composed models of discrete event systems are constructed, we start with the definition of a model and the composition operators that we use to build those models. Note that the actual formalism used to describe the models we compose together can take many forms, including stochastic extensions to Petri nets, stochastic process algebras, and "buckets and balls" models, among others. Our intent is not to create yet another formalism, but simply to specify a simple model type that allows us to describe our technique. In reality,

[†]Approaches based on action synchronization typically impose restrictions on actions that are synchronized.

it will work with any discrete-event formalism that has the characteristics described below, including composed models with constituent models expressed in different formalisms.

Definition 1. *A model M is an 8-tuple $(V, V_{\tilde{s}}, V_s, A, s^0, \delta, w, p)$ where V is a finite, non-empty set of SVs and $V_s \subseteq V_{\tilde{s}} \subseteq V$. $V_{\tilde{s}}$ is a set of* shared *SVs; V_s is a set of exported* shared *SVs. D_v is the set of possible values $v \in V$ can take. A is a finite, non-empty set of actions. A state s is an element of $\times_{v\in V} D_v$, and s^0 is the initial state. The next state function $\delta : A \times (\times_{v\in V} D_v) \to (\times_{v\in V} D_v)$ is only partially defined and describes a successor state for a given action and state. Function $w : A \times (\times_{v\in V} D_v) \to \mathbb{R}^+$ defines a non-negative weight for an action and $p : A \to \{0, 1, \ldots, n\}$ defines a priority for an action using a finite subset of $\mathbb{N}$. For ease of notation, $\delta(a, s)$ and $w(a, s)$ are denoted by $\delta_a(s)$ and $w_a(s)$, respectively.*

Note that we do not impose restrictions on δ and w, as is typically done for formalisms using action synchronization. For instance, action synchronization requires that enabling conditions and state changes of synchronized actions be conjunctions of local conditions and local effects, e.g., the requirement called "product-form" decomposition in [9]. Since we compose models by sharing variables, we can allow $\delta_a(s)$ to be defined in a non-decomposable, rather arbitrary manner; e.g., $\delta_a(s)$ may be defined only for states where $\sum_{v\in V} s_v \geq c$, for some constant c. We compose models by sharing SVs. There are two ways to do so. If M itself consists of submodels and results from some composition of those submodels (composition operators will be defined below) then $V_{\tilde{s}}$ contains SVs that are shared among those submodels. In addition, if M is subject to composition itself, then certain SVs of M may be shared with other models; set V_s identifies those externally shared SVs within $V_{\tilde{s}}$. The usefulness of subsets $V_{\tilde{s}}$ and V_s of $V_{\tilde{s}}$ will become more clear after composition operators are defined below.

We will limit ourselves to consideration of models whose behaviors are Markov processes by enforcing the following two restrictions. 1) For any state s and action a with $p(a) < n$, $\delta_a(s)$ can only be defined if there is no action a' such that $\delta_{a'}(s)$ is defined and $p(a') > p(a)$. If $\delta_a(s)$ is defined, we say that a is *enabled* in s. 2) If $p(a) > 0$, then a is called *immediate* and action a takes place (fires) with probability $w_a(s)/\sum_{a'\in E(s)} w_{a'}(s)$, where $E(s) \subseteq A$ denotes the set of actions that are enabled in s. If $p(a) = 0$, then a is *timed* and action a takes place after a delay that is exponentially distributed with rate $w_a(s)$. Note that δ induces a reachability relation among states and that the reflexive, transitive closure of δ results in the state space of M. We restrict ourselves to models with a finite state space and those whose structure allows us to perform an on-the-fly elimination of vanishing states. The resulting set of tangible states is denoted by S. The generator matrix of the associated CTMC is $\mathbf{Q} = \mathbf{R} - \mathbf{D}$ where $\mathbf{R}(s, s')$ gives the sum of rates of all timed actions whose firing leads from s to s' (possibly including a subsequent sequence of immediate actions whose probabilities are multiplied with the rate of the initial timed action). $\mathbf{D} = diag(rowsum(\mathbf{R}))$ provides diagonal entries of $\mathbf{Q}$ as the sum of row entries of $\mathbf{R}$. In representing $\mathbf{Q}$, we will focus mainly on construction of $\mathbf{R}$, since for any given $\mathbf{R}$, derivation of $\mathbf{D}$ is straightforward.

In order to build models of complete systems from smaller and simpler models, we define two composition operators, "join" and "replicate," which are based on sharing SVs of the models on which they are defined [20]. The *join* operator combines a number of (possibly non-identical) models by sharing a subset of their SVs, while the *replicate* operator combines a number of copies of the same model by sharing the same subset of each of the models' SVs. The definition of join uses the notion of *substate* s_W, the projection of s on a set of state variables $W \subseteq V$.

Definition 2. *The* join *operator* $\mathcal{J}(V_J, M_1, \dots, M_n)$ *over models* $M_i = (V_i, V_{\tilde{s}i}, V_{si}, A_i, s^{0,i}, \delta_i, w_i, p_i)$, $i \in \{1, \dots, n\}$ *with* $V_J \subseteq \cup_{i=1}^n V_{si}$ *yields a new model* $M = (V, V_{\tilde{s}}, V_s, A, s_0, \delta, w, p)$ *with state variables* $V = \cup_{i=1}^n V_{si} \cup \uplus_{i=1}^n V_i \backslash V_{si}$*, where an appropriate renaming of SVs in* $V_i \backslash V_{si}$ *ensures unique names such that the union is over disjoint sets, and where* $\cup_{i=1}^n V_{si}$ *means that SVs with the same names are indeed joined.* $V_{\tilde{s}} = \cup_{i=1}^n V_{si}$ *and* $V_s = V_J$. $A = \uplus_{i=1}^n A_i$ *where an appropriate renaming of actions in* $A_1, \dots, A_n$ *ensures that the union is over disjoint sets.* $s^0(v) = s^{0,i}(v)$ *if* $v \in V_i - V_{si}$ *and* $s^0(v) = max_i s^{0,i}(v)$ *if* $v \in V_{\tilde{s}}$. *Functions* δ, w, *and* p *are defined such that* $\delta_a(s) = s'$, $w_a(s) = \lambda$, *and* $p(a) = p_i(a)$ *if there exists* $i \in \{1, \dots, n\}$ *such that* $a \in A_i$, $\delta_{i,a}(s_{V_i}) = s'_{V_i}$, $w_{i,a}(s_{V_i}) = \lambda$, *and* $s_{V-V_i} = s'_{V-V_i}$. *We call* $M_1, \dots, M_n$ *the* children *of model* M.

We now more precisely identify the role of $V_{\tilde{s}}$ and V_s in M. Elements of $V_{\tilde{s}}$ are SVs shared among the children of M, i.e., if M_i and M_j both have an SV x, and if $x \in V_{\tilde{s}}$, then M contains a single SV x shared by M_i and M_j. On the other hand, if $x \notin V_{\tilde{s}}$ then x is renamed in M_i and M_j (e.g., as x_i and x_j) such that M contains two different SVs. Furthermore, if M itself is used as a child in a subsequent join operator, only the SVs in V_s are visible and can be shared with other children of that join operator.

By convention, we use the maximum initial value of the shared SVs as the value of the resulting shared SV. Note that the join operator is a commutative operator.

Definition 3. *The* replicate *operator* $\mathcal{R}_n(V_J, M)$ *yields a new model* $M' = \mathcal{J}(V_J, M_1, \dots, M_n)$ *with* $M_i = M$ *for all* $i \in \{1, \dots, n\}$ *and* $V_J \subseteq V_{sM}$. *We call* M *the* child *of model* M', *and* n *the* cardinality *of the operator.*

The replicate operator is a special case of the join operator, in which all composed models are identical; for that reason, it exhibits desirable properties with respect to the lumpability of the CTMC its resulting model generates.

Note that the set of models is closed under the join and replicate operators, meaning that the result of each of the operators is a model itself, and therefore can be a child of another join or replicate operator. This property enables us to build composed models that are hierarchical. Such composed models require a starting set of "atomic" models that act as building blocks. Atomic models are built without use of replicate or join operators and have $V_{\tilde{s}} = V_s$ since there is no reason to have shared SVs that are not externally visible. For analysis of a single atomic model as such, classical CTMC analysis applies. Hence, in the following, we are interested only in composed models that contain at least one join or replicate operator.

For a composed model that is given in terms of possibly nested join and/or replicate operators, we call each occurrence of an atomic model or the result of each occurrence of an operator a *component*. Note that every component is a model. For a model that contains m components we can define an *index* $1, 2, \dots, m$ over the components of a term from left to right after expanding replicate operators into join operators. For example,

$$M = \mathcal{R}_2(V_J{}', \mathcal{J}(V_J, M', M'')) = \mathcal{J}(V_J', \mathcal{J}(V_J, M', M''), \mathcal{J}(V_J, M', M''))$$

obtains indices as in

$$= \mathcal{J}_1(V_J', \mathcal{J}_2(V_J, M_3, M_4), \mathcal{J}_5(V_J, M_6, M_7))$$

where $m = 7$, the leftmost join operator corresponds to the component with index 1, the second leftmost join operator has index 2, and the last component is M'' to the right with index 7.

Obviously, V_J and V'_J do not receive indices, because they are not components. In the rest of the paper, the set of SVs of component c, the set of actions of component c, and the set of SVs of model M are respectively denoted by V_c $(V_{\tilde{s}c}, V_{sc})$, A_c, and V. In the following, we consider only the non-trivial case $m > 1$. The motivation for this indexing scheme is partitioning of the set of SVs V into m disjoint subsets as follows. If component c corresponds to a join operator, then $\mathcal{V}_c = V_{\tilde{s}c} \backslash V_{Jc}$. If component c is an atomic model, then $\mathcal{V}_c = V_c \backslash V_{sc}$. The partition is denoted by $\mathcal{P} = \{\mathcal{V}_1, \ldots, \mathcal{V}_m\}$, and we call $\mathcal{V}_i$'s $(1 \leq i \leq m)$ the *blocks* of $\mathcal{P}$. [‡]

For any component c, we can define an injective mapping $g_c : \times_{v \in \mathcal{V}_c} D_v \rightarrow \mathbb{N}_0$ ($\mathbb{N}_0$ is the set of non-negative integers) that gives an index number to any setting of SVs in $\mathcal{V}_c$. Since we consider only models with finite state spaces, the domain of g_c is finite. Clearly, many such mappings exist. At this point the only condition on the mapping is that it be injective such that any state $s = (s_1, \ldots, s_m) \in \times_{c=1}^{m} (\times_{v \in \mathcal{V}_c} D_v)$ of a model, where $s_i = s_{\mathcal{V}_i}$, has a unique representation as a vector $v = (g_1(s_1), \ldots, g_m(s_m))$ in $\mathbb{N}_0^m$. The ith component of the vector v, which is denoted by v_i $(1 \leq i \leq m)$, is in fact the index of substate $s_{\mathcal{V}_i}$. We will use v and s interchangeably to represent states. In that way, we will obtain a uniform representation of a state as a vector of natural numbers.

An important property of the replicate operator is that it generates a behavior that enables lumping on the associated CTMC of a model [20]. We can define the lumped state space of a model with full state space S through the help of equivalence relations. In particular, for $\mathcal{R}_{n_c}(V_J, M)$ with index c and cardinality n_c, let l_c be the number of indices used for a single replica of M. Then, by construction, all indices in the range $c, c+1, \ldots, c+n_c l_c$ are associated with that replicate component. Let $v(c,i) = (v_{c+il_c+1}, \ldots, v_{c+il_c+l_c})$ be a subvector of $v \in \mathbb{N}_0^m$ for $i \in \{0, \ldots, n_c - 1\}$. If the child of a replicate component c is an atomic component, then $l_c = 1$, and $v(c,i)$ consists of a single element. We define an equivalence relation R_c on $v \in \mathbb{N}_0^m$ as follows. A pair $(v, v') \in R_c$ with vectors $v, v' \in \mathbb{N}_0^m$ if and only if

1. $v_i = v'_i$ for all $i \in \{1, \ldots, c, c+n_c l_c + 1, \ldots, m\}$ and
2. there exists a permutation (a bijective function) $q : \{0, \ldots, n_c - 1\} \rightarrow \{0, \ldots, n_c - 1\}$, such that $v(c,i) = v'(c, q(i))$ for all $i = 0, \ldots, n_c - 1$.

When a hierarchical model contains a number of replicate components, we define the overall equivalence relation R to be the union of equivalence relations over all replicate components, i.e., $R = \cup_{c \text{ is replicate}} R_c$. In the next section, we describe how we lump the state space S by building the quotient S/R; we identify each equivalence class of R by a specific representative state. The set of these representative states constitutes the lumped state space, S_{lumped}.

2.2. Review of MDD and MD data structures

In order to compute performance measures of a composed model, we need to construct a CTMC representing the behavior of the model. Our main goal is to extend the size of composed models that can be handled on a typical computer system by using the structural properties of a model both to reduce the number of states that need to be considered and to compactly represent

[‡]Depending on the composed model, some of the blocks of $\mathcal{P}$ may be empty. For the sake of simplicity of the presentation, we assume that all blocks are non-empty. However, the degenerate cases are addressed in our implementation.

the states that need to be considered. With that aim, we chose to use MDD and MD data structures, respectively, to represent the set of states and the set of transitions of the CTMC associated with a composed model, and use the structural characteristics of the model to lump equivalent states. In the following, we give a brief description of the two data structures.

Multi-valued Decision Diagrams Review. MDDs [21] generalize binary decision diagrams (BDDs) [1]. They are useful for encoding a set of vectors $S \subseteq \times_{i=1}^{m} S_i$ since they can represent functions of the form $f : \times_{i=1}^{m} S_i \rightarrow \{0, 1\}$ for finite sets $S_i = \{0, \ldots, |S_i| - 1\}$, $i = 1, \ldots, m$. Hence, $(s_1, \ldots, s_m)$ is an element of S if and only if $f(s_1, \ldots, s_m) = 1$. MDDs are ordered, i.e., the order of S_i's is fixed; we consider the order $S_1, \ldots, S_m$. They are rooted, directed, acyclic graphs ("folded trees") with *terminal* and *non-terminal* nodes. A terminal node is either 0 or 1; a non-terminal node has a variable $x_i \in S_i$ assigned to it and contains $|S_i|$ pointers to a node with a variable x_j, $j > i$ or to a terminal node. A pointer corresponds to a co-factor of f that is defined as $f_{x_i=c} = f(s_1, \ldots, s_{i-1}, c, s_{i+1}, \ldots, s_m)$ for variable x_i and a constant $c \in S_i$. Since the order is fixed, a node with variable x_i is referred to as a *level-i node*, and the function that a level-i node represents is denoted by $(x_i, f_{x_i=0}, \ldots, f_{x_i=|S_i|-1})$. In the algorithms we develop, we use $u[k]$ to denote the node to which the kth pointer of a non-terminal node u points, and the set of all level-i nodes is denoted by N_i. $u[k]$ is also called a *child* of node u. As described in [9], typical set operations like union, intersection, and difference can be performed on MDDs efficiently. MDDs are often enhanced by a so-called offset function $\rho : \mathcal{S} \rightarrow \{0, 1, \ldots, |\mathcal{S}| - 1\}$, where the ith element of S with respect to lexicographical order obtains value $i-1$. ρ is encoded through assignment of an additional weight $\rho_i(s_1, \ldots, s_k)$ to each pointer of a level-i node u with value k, and the offset of s is the sum of weights along the corresponding path in the MDD, i.e., $\rho(s) = \sum_{i=1}^{m} \rho_i(s_1, \ldots, s_i)$.

The main advantage of MDDs is that a reduction operation is used to represent isomorphic subgraphs only once. This reduction is based on a notion of equality for nodes; two nodes are *equal* if they are terminal nodes of the same value or if they have equal tuples $(x_i, f_{x_i=0}, \ldots, f_{x_i=|S_i|-1})$. A non-terminal node is *redundant* if all of its pointers point to the same node. We follow [8] and consider ordered MDDs, where equal nodes have been merged and redundant nodes are retained only to ensure that a pointer of a level-i node can lead only to a level-$(i+1)$ node or to the terminal node with value 0. The value of function f is derived by following a path in an MDD graph starting at the root node and ending, after at most m nodes, at a leaf node that represents the resulting value of $\{0, 1\}$. At each intermediate level-i node, a successor node is selected according to s_i. MDDs are particularly space-efficient for representation of S when there are a significant number of common subvectors in S. In the rest of the paper, the MDD representation of a set (e.g., S) is denoted by the calligraphic letter (e.g., $\mathcal{S}$) corresponding to that set.

Matrix Diagram Review. An MD is a directed acyclic graph like an MDD, but its nodes are matrices; MD provides one matrix per node, whereas an MDD provides one vector per node. However, there are further differences. A non-terminal $S_i \times S_i'$ matrix at level $i \in \{1, \ldots, m\}$ of an MD contains elements that are sets of pairs. Each pair (r, p) consists of a real value r and a pointer to a level-$(i+1)$ node or a terminal node. Terminal nodes are 1×1 matrices with entries $\{0, 1\}$. Clearly, only paths that finally lead to the entry 1 are relevant, so terminal nodes are necessary only so that the theoretical framework will be coherent; they are not explicitly considered in an implementation.

As in MDDs, the order of levels is fixed, and we can define two level-i nodes to be equal if their matrices are equal. Again, we consider a reduced structure in which equal nodes have been merged. In a reduced MD, any two pairs (r, p), (r', p') in a matrix entry (s_i, s'_i) of some level-i node are replaced by a pair $(r + r', p)$ if $p = p'$. Let Π denote the set of all paths with elements (s_i, s'_i, r_i, p_i) that start at the root node and follow a sequence $(s_1, s'_1), \ldots, (s_m, s'_m)(1)$ of matrix elements $(r_1, p_1), \ldots, (r_m, p_m)$ that ends at 1. A matrix diagram encodes a function $f : \times_{i=1}^{m}(S_i \times S'_i) \to \mathbb{R}$ where $f((s_1, s'_1), \ldots, (s_m, s'_m)) = \sum_{\pi \in \Pi} \prod_{i=1,(s_i,s'_i,r_i,p_i)\in\pi}^{m} r_i$, i.e., real values are multiplied along a path and summed over all paths. This definition allows us to use MDs to encode a matrix like $\mathbf{R}$. Algorithms for manipulating MDs are described in detail in [8]. In order to make the MDs that we generate compatible with the MDD of the state space, we encode the SVs in $\mathcal{V}_i$ in level i of the MD, as we did for the MDD.

3. Symbolic Generation of Lumped State Space

In this section, we will give a detailed description of our new algorithm for symbolic generation of the lumped state space of a composed model. In doing so, we first describe the state-space generation (SSG) algorithm that does not take lumping properties into account, and therefore generates the unlumped state space. Then we give an algorithm that exploits the structural properties of the replicate operator to lump the state space computed by the previous algorithm.

3.1. Symbolic Generation of Unlumped State Space for Composed Models

A symbolic SSG algorithm is similar to a traditional one in the sense that both algorithms start with the initial state of the model and keep firing actions until all reachable states have been explored. The difference is that in a traditional algorithm, each time an action is fired only one state is visited, while in a symbolic algorithm, a (potentially large) *set* of states is visited. In our SSG algorithm, we use MDDs to represent sets of states. In order to design an efficient symbolic algorithm for composed models, we identify key structural properties of a model, and based on those properties we determine the "meaning," with respect to the composed model structure, of each level of the MDD. In particular, when we use an MDD to represent the set of states of a model, we represent each state of the model by a vector. This vector representation is determined by partition $\mathcal{P}$ of the set of SVs. More formally, for each component $1 \leq c \leq m$, level c of the MDD represents substates of the form s_c. In other words, we define S_c, the set of possible values of a level-c node, such that $|S_c| = |\{s_c | s \in S\}|$.

An action a is called *independent* of a set of SVs W (in the context of a model M) if a's next state function δ and weight w are evaluated independently from the value settings for SVs in W; otherwise, a is *dependent* on W. To support our SSG algorithm, we partition the set of actions A_c of an atomic component c into $A_{c,l}$ and $A_{c,g}$, which are the sets of local and global actions of component c, respectively. a is *global* if it is dependent on any shared SV, and it is *local* otherwise. More formally, $a \in A_{c,l}$ if and only if a is independent of $V_c \backslash \mathcal{V}_c$.

In order to design an efficient state-space generation algorithm, we consider a restricted class of composed models in which all global actions are of the lowest priority, i.e., they are timed actions. There are no other restrictions on how actions are enabled or change state, i.e.,

$\delta_a(s)$ can be an arbitrary function on its atomic model's SVs. This generality implies that a distinction into acylic and cyclic dependencies as discussed in [22] does not apply. There is no restriction on local actions. The slight restriction on global actions we do have has two important implications that enable us to design an efficient SSG algorithm: 1) the elimination of vanishing states can take place locally, i.e., in each atomic component, and on the fly, i.e., without storing intermediate vanishing states, and 2) atomic components that share SVs cannot stop one another from proceeding locally. The latter property gives us the ability to use an approach similar to saturation [6] (in firing local actions) and generate a subset of the state space of an atomic component independently from other atomic components as long as the fired actions are independent from the shared SVs of that component, i.e., the actions are local.

3.1.1. The Overall Algorithm We first describe SSSE (shown in Figure 1(a)), the algorithm we employ to generate the unlumped state space of a composed model. SSSE calls two major procedures: LocalSSE, which explores the state space by firing local actions, and GlobalSSE, which does the same by firing global actions.

```
SSSE($s^0$)
1  $\mathcal{S} \leftarrow \{s^0\}$
2  $\mathcal{U} \leftarrow \{s^0\}$
3  while $\mathcal{U} \neq \emptyset$
4      LocalSSE($\mathcal{U}$)
5      GlobalSSE($\mathcal{U}$)
6      $\mathcal{U} \leftarrow \mathcal{U} - \mathcal{S}$
7      $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{U}$
8  return $\mathcal{S}$
```

(a) Overall symbolic state-space exploration algorithm

```
LocalSSE($\mathcal{U}$)
1  for all non-root nodes $u_c \in N_c$
       visited in a dfs order
       starting from $r$, root of $\mathcal{U}$
2      if $c \in I_{AM}$
3          $u_{\text{sat}} \leftarrow$ copy of $u_c$
4          Saturate($u_{\text{sat}}$)
5          if $u_c \neq u_{\text{sat}}$
6              replace $u_c$ by $u_{\text{sat}}$
```

(b) Local state-space exploration procedure

```
Saturate($u$)
1   $c \leftarrow$ level of $u$
2   $Y \leftarrow \{0, \ldots, |S_c| - 1\}$
3   while $Y \neq \emptyset$
4       pick $i$ from $Y$
5       for all $i'$ s.t. $\mathbf{B}_c^*(i, i') = 1$
6           $u' \leftarrow u[i] \cup u[i']$
7           if $u' \neq u[i']$
8               $u[i'] \leftarrow u'$
9               $Y \leftarrow Y \cup \{i'\}$
10      $Y \leftarrow Y - \{i\}$
```

(c) Saturation procedure

Figure 1. Pseudocodes of the overall and local state-space exploration procedures

We keep an MDD representation of two subsets of S: $\mathcal{S}$ and $\mathcal{U}$. SSSE starts by initializing $\mathcal{S}$ and $\mathcal{U}$ to $\{s^0\}$, the starting state of the system, in lines 1-2. At the beginning of each iteration of the while loop (line 3), two invariants hold true: $\mathcal{S}$ is the set of states that have been reached so far and $\mathcal{U}$ is the set of reached but unexplored states. Both sets contain only tangible states as we eliminate vanishing states on the fly. Thus, $\mathcal{U} \subseteq \mathcal{S}$. In lines 3-7, actions are repeatedly fired on states in $\mathcal{U}$, and $\mathcal{U}$ and $\mathcal{S}$ are updated accordingly. Each iteration of the while loop holds the invariants true. Therefore, the algorithm is over when $\mathcal{U} = \emptyset$, i.e., the firing of actions no longer generates any new states. At that point (line 8), $\mathcal{S}$ is the set of reachable states of the composed model.

The important point about this algorithm is the way it efficiently fires actions on states in $\mathcal{U}$. As we will describe below in detail, we handle the firing of local and global actions separately because we exploit the unique way each type of action modifies the MDD of the state space. LocalSSE($\mathcal{U}$) adds to $\mathcal{U}$ the set of states that can be reached from any state in $\mathcal{U}$ by a (finite) sequence of local action firings. Note that the local actions are timed; immediate actions are taken care of by on-the-fly elimination of vanishing states. GlobalSSE($\mathcal{U}$) adds to $\mathcal{U}$ the set of states that can be reached from any state in $\mathcal{U}$ by a *single* global action firing followed by a (finite) sequence of immediate (local) action firings.

LocalSSE and GlobalSSE do not take into account the lumping properties of replicate operators. Instead, they treat replicate operators as join operators with identical children. Moreover, they consider firing actions of atomic components only, because join and replicate operators do not introduce new actions of their own.

3.1.2. Firing Local Actions By definition, a local action $a \in A_{c,l}$ is independent of $V_c \backslash \mathcal{V}_c$, and therefore $\delta_a(s)$ depends only on $s_{\mathcal{V}_c}$. Furthermore, by the restriction we introduced earlier, all immediate transitions are local. Finally, note that all SVs in $\mathcal{V}_c$ are encoded in level c of the MDD. These properties imply that in order to generate a set of states that are visited by completion of action a, we only need to manipulate the nodes in level c of the MDD.

Suppose that a (tangible) substate s_c can lead to a tangible substate s'_c by a sequence of actions in $A_{c,l}$. Hence, if state $(v_1, \ldots, v_{c-1}, i, v_{c+1}, \ldots, v_m)$ is reachable, then state $(v_1, \ldots, v_{c-1}, i', v_{c+1}, \ldots, v_m)$ is also reachable where $i = g_c(s_c)$ and $i' = g_c(s'_c)$. To implement this local state exploration on the MDD, we perform the "saturation" operation on all nodes u in level c: $u[i'] \leftarrow u[i] \cup u[i']$ for all possible values of i and i'. In that operation, values of v_j's are implicit; all state paths that go through u constitute all states of the form $(v_1, \ldots, v_{c-1}, i, v_{c+1}, \ldots, v_m)$.

Figure 1(b) shows LocalSSE, which explores the state space using only local actions $A_{c,l}$ of every atomic component c. Therefore, LocalSSE iterates through all nodes u_c of levels that correspond to atomic components (denoted by I_{AM}) in depth-first search (dfs) order. For each node u_c, which encodes a set of substates of the form s_c, it saturates u_{sat}, the node that is to be the saturated version of u_c, by calling Saturate(u_{sat}) in line 4. Finally, in lines 5-6, u_c is replaced by its saturated version u_{sat}.[§] The reason for iterating through all nodes in dfs order is that (due to implementation issues) we need to ensure that a node is saturated after all its children have been saturated.

Saturate(u) (shown in Figure 1(c)) fires local actions until no further local action firing can add any substate to the set. Lines 3-10 perform the abovementioned saturation operation on u in a "symbolic" manner, i.e., for each i', lines 6-8 add all states of the form $(v_1, \ldots, v_{c-1}, i', v_{c+1}, \ldots, v_m)$ to $\mathcal{U}$. Notice that during the saturation operation, we may need to increase the size of u (i.e., the number of its pointers), since we do not know the size of the state space of M_c in advance. The important point is that due to the locality of the actions, we can expand the set of reachable states of the system only by (local) changes to u.

Repetitive computations related to local state exploration might occur, since the same substate may be explored many times for different nodes throughout the execution of SSSE. In order to prevent these extra computations, we need an efficient data structure for each atomic component c that stores the reachability relation among substate indices of that component. More formally, we need to know, for every i, the set of all substate indices i' where substate with index i' can be reached from substate with index i' by a (finite) sequence of local action firings. We can determine that by computing the reflexive and transitive closure of a square Boolean-valued matrix denoted by $\mathbf{B}_c$. $\mathbf{B}_c$ is defined on the tangible state space of M_c, which

[§]In the actual implementation, u_c is not replaced by u_{sat} in one step. Instead, u_c is replaced by u_{sat} for each of the pointers coming from the upper level. Hence, eventually no node will point to u_c, u_c will be garbage-collected, and therefore u_c will essentially be replaced by u_{sat}.

```
GlobalSSE($\mathcal{U}$)
1  for all $c \in I_{AM}$
2      allocate arrays $u_{0..|L_c|}, i_{0..|L_c|}$
3      $u_0[i_0] \leftarrow$ root of $\mathcal{U}$
4      $\mathcal{U}' \leftarrow$ FireAllGlobals$(c, 1, u, i)$
5      $\mathcal{U} \leftarrow \mathcal{U} \cup \mathcal{U}'$
```

(a)

```
FireAllGlobals($c, k, u_{0..|L_c|}, i_{0..|L_c|}$)
1  $\mathcal{U}' \leftarrow \emptyset$
2  for all nodes $w$ of level $L_c(k)$
3      $u_k \leftarrow w$
4      if MDDConnector$(u_{k-1}[i_{k-1}], u_k) \neq \emptyset$
5          if $(k < |L_c|)$  $\mathcal{T}' \leftarrow \emptyset$
6          for $j \leftarrow 0$ to $|S_{L_c(k)}| - 1$
7              $i_k \leftarrow j$
8              if $(k < |L_c|)$
9                  $\mathcal{T}' \leftarrow \mathcal{T}' \cup$ FireAllGlobals$(c, k+1, u, i)$
10             else $\mathcal{U}' \leftarrow \mathcal{U}' \cup$ FireOneGlobal$(c, u, i)$
11         $\mathcal{U}' \leftarrow \mathcal{U}' \cup \mathcal{T}'$
12 return $\mathcal{U}'$
```

(b)

Figure 2. Pseudocode of the global state-space exploration procedure

means that $\mathbf{B}_c$ has $|S_c|$ rows and columns. $\mathbf{B}_c(i, i') = 1$ if and only if, starting from substate i, there is a sequence of local action firings (in which the first is timed and the others, if any, are immediate) that leads to substate i'. Otherwise, $\mathbf{B}_c(i, i') = 0$. Let $\mathbf{B}_c^*$ be the reflexive and transitive closure of $\mathbf{B}_c$. That means $\mathbf{B}_c^*(i, i') = 1$ if and only if there is a (possibly empty) sequence of local action firings that takes component c from substate i to i'. Since entries of $\mathbf{B}_c$ are updated as we explore the atomic component state space, and since computing the transitive closure from scratch is expensive, we use a simple but rather efficient online algorithm given by Ibaraki and Katoh [14] to maintain $\mathbf{B}_c^*$ as we update $\mathbf{B}_c$. In the actual implementation, the size of $\mathbf{B}_c^*$ increases as the set of possible substates s_c grows.

To be concise, we have not shown the pseudocode for computing $\mathbf{B}_c^*$. One way to understand how its elements are computed is to assume that accessing $\mathbf{B}_c^*(i, i')$ in line 5 of Saturate causes a function call if substate i has not already been explored. The function explores substate i by firing actions in $A_{c,l}$, computes one row of $\mathbf{B}_c$ as defined above, and computes elements of $\mathbf{B}_c^*$ using Ibaraki and Katoh's algorithm.

3.1.3. Firing Global Actions Figure 2(a) shows GlobalSSE, which explores the state space using only global actions $A_{c,g}$ of every atomic component c. GlobalSSE iterates through all atomic components $c \in I_{AM}$. For each one, the recursive procedure FireAllGlobals in line 4 generates the set of tangible states that are reachable from states in $\mathcal{U}$ by firing one action in $A_{c,g}$. Finally, in line 5, the states are added to $\mathcal{U}$. The roles of arrays of nodes u and array of substate indices i are described below. They are allocated in GlobalSSE but initialized and used in the recursive calls of FireAllGlobals.

Consider a global action $a \in A_{c,g}$ of an atomic component c. By definition, a depends not only on SVs in $\mathcal{V}_c$ but also on shared SVs in other $\mathcal{V}$ sets. The partition $\mathcal{P} = \{\mathcal{V}_1, \ldots, \mathcal{V}_m\}$ determines I_c, the set of indices of MDD levels in which the SVs in V_c are encoded. In particular, $I_c = \{c' | \mathcal{V}_{c'} \cap V_c \neq \emptyset\}$. Therefore, $V_c \subseteq \cup_{c' \in I_c} \mathcal{V}_{c'}$, and, due to the order we chose on $\mathcal{P}$, $c' \leq c$ for all $c' \in I_c$. In order to impose an order on the members of I_c, we use L_c to denote the sequence of members of I_c sorted in ascending order. In other words, if the jth member of a sequence L is denoted by $L(j)$, we have $|L_c| = |I_c|$, $L_c(j) \in I_c$ for all $1 \leq j \leq |L_c|$, and $L_c(j) < L_c(j+1)$ for all $1 \leq j < |L_c|$.

It is important to note that, in terms of changes that need to be applied on the MDD, firing a global action in state-sharing composed models is inherently more difficult than firing a synchronizing action in an action-synchronization model, as discussed in [6]. The reason is that in the latter case, the sets of SVs of atomic submodels are disjoint, and due to the product-form behavior [6], the changes that need to be applied on a node v (in the level corresponding to an atomic model) during saturation depend only on the information present in v and the action a to be fired, regardless of whether a is local or synchronizing. However, in the former case, some SVs are shared among atomic models, so that firing a global action a on a node v requires not only the information in v but also the information in other levels of the MDD. This makes the saturation approach inapplicable in the case of firing global actions in state-sharing composed models.

Now that we know what levels of the MDD are affected by action a, we discuss how they are affected. To fire action a, we need to add the state $(s'_{V_c}, s_{V\setminus V_c})$ to $\mathcal{U}$ for each state $s \in \mathcal{U}$ where $\delta_a(s) = s'$. To realize this state addition operation on the MDD, we have to consider the paths corresponding to all such states s. Then, for each path, we have to update nodes in appropriate levels. However, considering the paths one by one is not the best way to do so. To describe the better method we have developed, we first need to define the concept of an "MDD connector." An *MDD connector between two nodes w and w'* is a subgraph of the MDD that includes only sub-paths of the MDD that start from w and end with w'. MDD connectors connect the nodes of levels in I_c if they differ by more than one level.

To illustrate this, consider the example in Figure 3, in which $|L_c| = 3$. u_0 is an imaginary node such that $u_0[i_0]$ is equal to the root of $\mathcal{U}$[¶] (line 3 of GlobalSSE). It is used to avoid case-by-case analysis, and thus to simplify the presentation. u_k is a node in level $L_c(k)$ of the MDD for $k \in \{1, \ldots, |L_c|\}$. The left-hand side of the figure shows all paths of the MDD (before firing action a) that pass through all u_k's. Let $\hat{\mathcal{U}}$ be the set of all states that these paths represent. Let U_k be the MDD connector between $u_{k-1}[i_{k-1}]$ and u_k where $i_k = g_c(s_{L_c(k)})$. FireOneGlobal, which is called by FireAllGlobals (Figure 2(b)) generates another MDD that represents the set of states reachable from $\hat{\mathcal{U}}$ by firing of all actions $a \in A_{c,g}$, that is, $\hat{\mathcal{U}}' = \{(s'_{V_c}, s_{V\setminus V_c}) | s \in \hat{\mathcal{U}}, \delta_a(s) = s', a \in A_{c,g}\}$. Notice that in order to generate $\hat{\mathcal{U}}'$, we do not need to change the nodes in any of the U_k's ($k \in \{1, \ldots, |L_c|\}$), because action a is independent of the SVs encoded in the levels corresponding to U_k's. Instead, FireOneGlobal 1) makes a copy of each U_k, 2) computes $i'_k = g_c(s'_{L_c(k)})$ and creates new nodes u'_k for each enabled action in $A_{c,g}$, and 3) connects all the new nodes as shown on the right-hand side of Figure 3 in order to build $\hat{\mathcal{U}}'$. Because of limited space, the pseudocode of FireOneGlobal is not given.

To generate states reached by firing actions in $A_{c,g}$ from all states in $\mathcal{U}$, we have to consider all distinct sets of nodes $\{u_1, \ldots, u_{|L_c|}\}$ (i.e., nodes with index levels in L_c) and their corresponding indices: $i_1, \ldots, i_{|L_c|}$. For each of the distinct sets of nodes and indices we have to consider the corresponding $\hat{\mathcal{U}}$ and generate the corresponding $\hat{\mathcal{U}}'$ as described above. Generation of all such $\hat{\mathcal{U}}'$'s is the role of FireAllGlobals$(c, k, u_{0..|L_c|}, i_{0..|L_c|})$, which recursively iterates through all nodes in the levels $L_c(k), \ldots, L_c(|L_c|)$ (line 2) and all substate indices (line 6) of those nodes. In each recursive call, MDDConnector in line 4 checks whether there is an MDD connector between two neighboring nodes, i.e., between $u_{k-1}[i_{k-1}]$ and u_k. If there is one, the procedure goes deeper down in the MDD by a recursive call; otherwise, it tries the next substate index

[¶]Strictly speaking, no such u_0 exists, because there is no node that points to any level-1 node, including the root of $\mathcal{U}$.

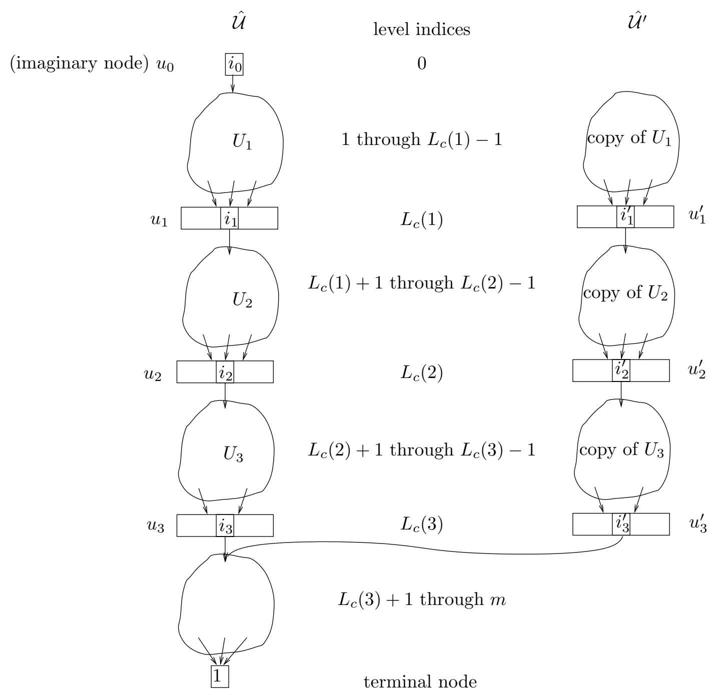

Figure 3. Computing the set of next states for global actions using MDD connectors

in u_k or the next node in level $L_c(k)$. When $k = |L_c|$ the algorithm is at level $L_c(|L_c|)$, which means that all the substate indices necessary to rebuild the state of an atomic model are known and stored in array i. Moreover, the nodes in each of the levels $L_c(1), \ldots, L_c(|L_c|)$ are stored in array u. In that situation, FireOneGlobal in line 10 fires all actions in $A_{c,g}$ that are enabled in the substate given by array i, builds $\hat{\mathcal{U}}'$, and adds the states of $\hat{\mathcal{U}}'$ to $\mathcal{U}'$. Finally, we compute the union of all the resulting $\mathcal{U}'$ sets and add it to $\mathcal{U}$.

Given $\mathcal{S}$, we could obtain an MD of the unlumped CTMC, and apply known approaches for the MD-based numerical analysis of CTMCs [8]. Part of our goal, however, is to reduce the number of states in the resulting CTMC, so, as discussed in the next subsection, we compute $\mathcal{S}_{lumped}$.

3.2. MDD Conversion to Lumped State Space

In the SSG algorithm described above, we treated the replicate operator as a join operator with identical children, without considering lumping properties. The final step is to lump the states according to the lumping induced by the structure of the composed model or, in other words, to compute S/R. In order to do that, we use a specific element of each equivalence class as an identifier (or representative) of the class. In particular, for a replicate component c, we define on state vector v function $min_c(v) : \mathbb{N}_0^m \to \mathbb{N}_0^m$ to be the state vector v' that satisfies $(v', v) \in R_c$ and $v'(c,i) \leq_{lex} v'(c,i+1)$ for all $i \in \{0, \ldots, n_c - 2\}$ where $\leq_{lex}$ is the lexicographical order on vectors. We also define function $min(v) : \mathbb{N}_0^m \to \mathbb{N}_0^m$ to be the vector v' that satisfies $min_c(v) = v'$ for all replicate components c. Obviously, $min_c(v) = min_c(v')$

for all $(v, v') \in R_c$, and similarly $min(v) = min(v')$ for all $(v, v') \in R$. Given any vector v, the corresponding class identifier $min(v)$ can be computed by appropriate sorting operations on v. Notice that for every equivalence class $\mathcal{C}$ of S/R there exists only one v such that $v \in \mathcal{C}$ and $min(v) = v$. To compute S/R we eliminate from $\mathcal{S}$ all paths (states) that do not satisfy $min(v) = v$. That computation is translated, in terms of MDDs, to the computation of $\mathcal{S} \cap \mathcal{R}$, where $\mathcal{R}$ is the MDD representation of the set of all states $v \in \mathbb{N}_0^m$ that satisfy $min(v) = v$. Hence, the problem of computing S/R is reduced to building $\mathcal{R}$ based on the definition of min.

However, the definition of $min(v)$ imposes a strict relationship among the various components of vector v, which implies a tight coupling among levels of $\mathcal{R}$ in terms of MDDs. That implies that the number of nodes of $\mathcal{R}$ grows very quickly in terms of the number of levels involved in the definition of $min(v)$,[‖] and this makes the direct computation of $\mathcal{S} \cap \mathcal{R}$ problematic.

To avoid this large memory consumption, we can express the large MDD of $\mathcal{R}$ in terms of a small number of considerably smaller MDDs, and instead of computing $\mathcal{S} \cap \mathcal{R}$ directly, we compute the intersection of $\mathcal{S}$ with a large set expression that is equal to $\mathcal{R}$. As the first step, we observe that $\mathcal{R} = \cap_{c \text{ is replicate}} \mathcal{R}_c$ where $\mathcal{R}_c$ is the set of all states $v \in \mathbb{N}_0^m$ that satisfy $min_c(v) = v$. That implies $\mathcal{S} \cap \mathcal{R} = (\cdots(\mathcal{S} \cap \mathcal{R}_{c_1}) \cap \cdots \cap \mathcal{R}_{c_j})$, where $c_1, \ldots, c_j$ are the indices of all replicate components of a composed model. Since in general, each of the $\mathcal{R}_c$'s involves tight coupling among far fewer levels than $\mathcal{R}$ does, each $\mathcal{R}_c$ is significantly smaller than $\mathcal{R}$. Hence, computing $(\cdots(\mathcal{S} \cap \mathcal{R}_{c_1}) \cap \cdots \cap \mathcal{R}_{c_j})$ is much faster than computing $\mathcal{S} \cap \mathcal{R}$ directly, because the efficiency we gain by using smaller-sized $\mathcal{R}_c$'s outweighs the extra time we have to spend to compute j intersection operations rather than one.

Still, we can do better. The next phase is to divide each $\mathcal{R}_c$ into many MDDs, each of which has tight coupling between only two levels. As an example, suppose $l_c = 1$ for a replicate component c. Then $\mathcal{R}_c$ is the MDD representation of the set of vectors $v \in \mathbb{N}_0^m$ that satisfy $v_{c+1} \leq \ldots \leq v_{c+n_c}$, and therefore, $\mathcal{R}_c$ involves coupling among n_c levels. However, we have $\mathcal{R}_c = \mathcal{R}_{c,1} \cap \cdots \cap \mathcal{R}_{c,n_c-1}$ where $\mathcal{R}_{c,i}$ $(1 \leq i < n_c)$ is the set of vectors that satisfy $v_{c+i} \leq v_{c+i+1}$. Now, instead of computing $\mathcal{S} \cap \mathcal{R}_c$ directly, we compute $(\cdots(\mathcal{S} \cap \mathcal{R}_{c,1}) \cap \cdots \cap \mathcal{R}_{c,n_c-1})$. For cases in which $l_c > 1$, the same technique is still applicable, and generally, it can be shown that indirect computation of $\mathcal{S} \cap \mathcal{R}_c$ involves creating $\mathcal{O}(n_c l_c)$ small MDDs and performing $\mathcal{O}(n_c l_c)$ MDD set operations (i.e., union and intersection), where $\mathcal{O}$ is the big O notation.

4. State Transition Rate Matrix Generation and Numerical Analysis

In this section, we describe how to perform an iterative numerical analysis based on an MD representation of $\mathbf{R}$ for the lumped CTMC. Its basic step is a matrix-vector multiplication, which requires consideration of several issues if it is performed with an MD. We start with the generation of an MD from the local transition rate matrices generated during state-space exploration. In Section 3 only Boolean matrices $\mathbf{B}_c$ of state transitions are mentioned; however, it is straightforward to have corresponding rates (possibly scaled by probabilities of paths

[‖]In the case of nested replicate operators, we claim that this number can be exponential in terms of the cardinality of the inner replicate operators. Proving this claim is not difficult, but to be concise, we do not give a proof here.

of subsequent immediate transitions) as matrix entries yielding matrices $\mathbf{R}_c$. With the MD representation of the unlumped CTMC at hand, we need to focus on $\mathcal{S}_{lumped}$ as the set of rows. Matrix entries in those rows will refer to columns s' whose correspondence to $min(s')$ must be established. Finally, there are cases in which the MD will generate multiple elements that must be added for a single matrix entry in $\mathbf{R}_{lumped}$. These issues are resolved in the remaining section.

4.1. State Transition Rate Matrix Generation using MDs

Formally, we first derive a generalized Kronecker representation of the rate matrix $\mathbf{R}$ that gives us an oversized MD in a straightforward manner, and then use the MDD to obtain a projection to the lumped state space. Conceptually, MD generation with the help of a Kronecker representation and MDD projection follows the line of arguments in [8]. However, it differs in important aspects. In particular, the Kronecker representation we derive contains functional transitions [22] that are subsequently resolved to constant values in the MD. The MD that finally results requires additional, specific algorithms to describe the rate matrix of the lumped CTMC. Note that an implementation directly generates an MD based on the local transition rate matrices obtained during state-space exploration.

A Kronecker representation for R. A Kronecker structure makes use of the matrix operator Kronecker product $\otimes$ to combine small component matrices into a large matrix. The building blocks of the Kronecker representation are matrices $\mathbf{R}_{a,c}$ that represent the effect of timed action a on atomic component c.[**] Let $\gamma_c : S \to \times_{v \in V_c} D_v$ be a mapping that provides the state in terms of its SVs for an atomic model M_c with component index c. Also let $n_c = |codomain(\gamma_c)|$. In fact, $\mathbf{R}_{a,c} \in \mathbb{R}^{n_c \times n_c}$, and $\mathbf{R}_{a,c}(s, s')$ is the weight of a at s multiplied by the probability of reaching s' via some sequences of immediate actions in M_c, where s and s' are states of atomic component c.

Note that the difficulty in the derivation of $\mathbf{R}_{a,c}$ is not in calculating entries, which is done by using the definition of M_c. The difficulty is in finding the set of reachable states of M_c, since sharing s with other models causes other models to generate new states as well[††]. This difficulty is overcome by using S as described below. Specifically, for each timed action a and atomic component c, we define m matrices $\mathbf{R}^i_{a,c}, i \in \{1, \ldots, m\}$ where $\mathbf{R}^i_{a,c}$ denotes the projection of $\mathbf{R}_{a,c}$ on $S_i \times S_i$ and $S_i = \{s_i | s \in S\}$. In fact, $\mathbf{R}^i_{a,c}$ denotes the "effect" of $\mathbf{R}_{a,c}$ on level i of the MD representation of $\mathbf{R}$. More formally, $\mathbf{R}^i_{a,c} \in \mathbb{R}^{S_i \times S_i}$ and

$$\mathbf{R}^i_{a,c}(s_i, s'_i) = \begin{cases} 1 & \text{if } i \neq c \text{ and } \exists s = (s_1, \ldots, s_m), s' = (s'_1, \ldots, s'_m) \in S \\ & \text{such that } \mathbf{R}_{a,c}(\gamma_c(s), \gamma_c(s')) \neq 0 \\ f_a & \text{if } i = c \text{ and } \exists s = (s_1, \ldots, s_m), s' = (s'_1, \ldots, s'_m) \in S \\ & \text{such that } \mathbf{R}_{a,c}(\gamma_c(s), \gamma_c(s')) \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

where $f_a : S \times S \to \mathbb{R}$ is a functional transition that evaluates to $\mathbf{R}_{a,c}(\gamma_c(s), \gamma_c(s'))$ for given

[**]Immediate actions are used only during the on-the-fly eliminations of vanishing states.
[††]Formally, one may consider those new states as a set of initial states S_0 that may grow as a result of firing global actions of other models.

states s, s'; see [22] for the definition and treatment of Kronecker representations that are generalized with respect to functions as matrix entries. Notice that $\mathbf{R}^i_{a,c}$ is simply an identity matrix if $i \notin I_c$, where, as defined before, I_c is the set of indices of MDD levels in which the SVs in V_c are encoded.

With those matrices, we obtain a Kronecker representation to describe a state transition rate matrix $\widehat{\mathbf{R}}$. Its basic operation, the Kronecker product $C = A \otimes B$, is defined for matrices $A \in \mathbb{R}^{n\times m}, B \in \mathbb{R}^{k\times l}$, and $C \in \mathbb{R}^{nk\times ml}$ as $C(a_1 \cdot k + b_1, a_2 \cdot l + b_2) = A(a_1, a_2) \cdot B(b_1, b_2)$. We define $\widehat{\mathbf{R}}$ as

$$\widehat{\mathbf{R}} = \sum_{M_c \in AM} \sum_{a \in A_c} \otimes_{i=1}^m \mathbf{R}^i_{a,c}, \tag{1}$$

where AM stands for atomic models. We briefly argue why $\mathbf{R}$ is a submatrix of $\widehat{\mathbf{R}}$. We consider an entry $\mathbf{R}((s_1, \ldots, s_m), (s'_1, \ldots, s'_m)) = \lambda$. Since several actions may contribute to λ we have $\lambda = \sum_{a \in E(s)} \lambda_a(s, s')$ where $\lambda_a(s, s')$ is $w_a(s)$ possibly multiplied by the probability of a subsequent sequence of immediate actions yielding s'. For any term $\lambda_a(s, s') > 0$, we defined $\mathbf{R}^i_{a,c}(s_i, s'_i) = \lambda_i > 0$ for $i = 1, \ldots, m$. Since only $\lambda_c \neq 1$ we have $\prod_{i=1}^m \lambda_i = \lambda_c = f_a = \mathbf{R}_{a,c}(\gamma_c(s), \gamma_c(s'))$. By the definition of Kronecker product, $\otimes_{i=1}^m \mathbf{R}^i_{a,c}$ contributes $\lambda_c = \prod_{i=1}^m \mathbf{R}^i_{a,c}(s_i, s'_i)$ to $\widehat{\mathbf{R}}((s_1, \ldots, s_m), (s'_1, \ldots, s'_m))$. $\mathbf{R}$ is a submatrix of $\widehat{\mathbf{R}}$ since $S \subseteq \times_{i=1}^m S_i$.

MD construction for lumped state-transition rate matrix. Transformation of a Kronecker representation into an MD is immediate. For each term $\otimes_{i=1}^m \mathbf{R}^i_{a,c}$, we define an MD with 1 node per level, the node at level i contains matrix $\mathbf{R}^i_{a,c}$, and its nonzero entries point to node $i+1$. In the case of $i = m$, the nonzero entries formally point to terminal node 1. Since addition is defined for MD, we can sum the resulting MDs of all terms in the two sums in Eq. 1. Note that the functional transitions that appear in $\mathbf{R}^i_{a,c}$ can be resolved to constant values in the MD, because sets $\mathcal{V}_i$ are ordered such that the sets that contain shared SVs of an atomic model M_c all have lower indices than c, and thus those sets appear at a higher level of MD. Hence, if a path through the MD reaches level c, the values of all shared SVs are known. Resolving functional transitions into constant values may require the splitting of matrices that were otherwise shared in the MD.

The advantage of an MD over a Kronecker representation is that we can restrict the MD to the $\mathcal{S} \times \mathcal{S}$ submatrix contained in a Kronecker representation. In order to restrict this representation to S, we refine the definition of matrices as $\mathbf{R}^i_{a,c}[(s_1, \ldots, s_{i-1}), (s'_1, \ldots, s'_{i-1})] \in \mathbb{R}^{S_i \times S_i}$ for an atomic model at component c and action a to depend on the subset of states $(s_1, \ldots, s_{i-1}), (s'_1, \ldots, s'_{i-1})$, namely:
$\mathbf{R}^i_{a,c}[(s_1, \ldots, s_{i-1}), (s'_1, \ldots, s'_{i-1})](s_i, s'_i)$

$$= \begin{cases} 1 \text{ if } i \neq c \text{ and } \exists s = (s_1, \ldots, s_m), s' = (s'_1, \ldots, s'_m) \in S \text{ s.t. } \mathbf{R}_{a,c}(\gamma_c(s), \gamma_c(s')) \neq 0 \\ \mathbf{R}_{a,c}(\gamma'_c(s_1, \ldots, s_c), \gamma'_c(s'_1, \ldots, s'_c)) \text{ if } i = c \\ 0 \text{ otherwise} \end{cases}$$

where γ'_c is the same as γ_c but defined on components $1, \ldots, c$, which is possible since all (shared) SVs of atomic model M_c appear at components $i \leq c$ and a is independent of SVs at components $c+1, \ldots, m$. That means that for any $(s_1, \ldots, s_m) \in S$ the equality $\gamma'_c(s_1, \ldots, s_c) = \gamma_c(s_1, \ldots, s_m)$ holds. To build a matrix diagram out of these matrices, we let an entry in $\mathbf{R}^i_{a,c}[(s_1, \ldots, s_{i-1}), (s'_1, \ldots, s'_{i-1})](s_i, s'_i)$ point to matrix $\mathbf{R}^{i+1}_{a,c}[(s_1, \ldots, s_i), (s'_1, \ldots, s'_i)]$

if $i < m$. Pointers from nonzero entries at level m point to terminal node 1. To keep the definition of those matrices readable, we oversized their dimension as $S_i \times S_i$; hence, some rows and columns in the matrices of the MD contain only zero entries and can safely be removed.

By construction, it is fairly clear that any path in the MD corresponds to a tuple $((s_1, \ldots, s_m), (s'_1, \ldots, s'_m))$ that describes the effect of action a in M_c, and that its value results from the product of values along the path. Since all numerical values except $\mathbf{R}_{a,c}(\gamma'_c(s), \gamma'_c(s'))$ are 1, it is clear that the resulting value gives the appropriate entry corresponding to a (possibly followed by some local immediate actions in M_c).

The MD of the overall model is then obtained by addition of the MDs for each timed action of all atomic models. Local actions of a model have room for optimization; for instance, their matrices can be summed up to a single local action to reduce the number of actions to be considered. So far, our presentation has followed a top-down approach to generate an MD; that gives us a natural way to verify the correctness of the MD construction. Clearly, during the construction of the MD, the reduction operator for matrix diagrams is applied to minimize space requirements of the overall structure.

The final step for the MD construction is to use the approach of [7, 8, 17], which is to project the rows and columns of the MD to $\mathcal{S}_{lumped}$ and $\mathcal{S}$, respectively. The resulting MD provides only rates of state transitions from $s \in \mathcal{S}_{lumped}$ to $s' \in \mathcal{S}$. However, note that s' may belong to $\mathcal{S} \backslash \mathcal{S}_{lumped}$, i.e., $s' \neq min(s')$. A recursive depth-first-search procedure enumerates all matrix entries encoded in the MD as triples (s, s', λ), where λ results from the product of values found on a path from the root node to a leaf node in the MD. The state information $s = (s_1, \ldots, s_m)$ must be mapped to the corresponding index value in $\{0, \ldots, |\mathcal{S} - 1|\}$ to support a matrix-vector multiplication. Other MD approaches perform that mapping by an offset function ρ encoded in an MDD [7, 8, 17]. In our case, we can look up $\rho(s)$ from the MDD of $\mathcal{S}_{lumped}$ only for $s \in \mathcal{S}_{lumped}$ by the help of the offset computation known for MDDs. If $s' \notin \mathcal{S}_{lumped}$, a straightforward option is to sort entries of s' to obtain the representative $min(s')$ of its equivalence class.

Sorting is avoided if we construct a new "sorting" MDD whose offset function ρ' is modified to fulfill $\rho'(s') = \rho(min(s'))$. This means paths of elements of the same equivalence class will evaluate to the same offset value. To generate the sorting MDD, we can start from an unreduced MDD in the form of a tree for set $\mathcal{S}$. A valid initial encoding of the mapping is to assign $\rho'_m(s_1, \ldots, s_m) = \rho(min(s_1, \ldots, s_m))$ and 0 to all internal values $\rho'_i(s_1, \ldots, s_i), i < m$. In order to allow for sharing, we perform a bottom-up procedure. Let $min(s_1, \ldots, s_i) = min_{s_i}\{\rho'_i(s_1, \ldots, s_i)\}$, then new offset values are $\rho'_{i-1}(s_1, \ldots, s_{i-1}) = min(s_1, \ldots, s_i)$ and $\rho'_i(s_1, \ldots, s_i) = \rho'_i(s_1, \ldots, s_i) - min(s_1, \ldots, s_i)$. The changes leave $\rho'(s) = \sum_{i=1}^{m} \rho'_i(s_1, \ldots, s_i)$ invariant, but reduce the ranges of numerical values at lower levels of the sorting MDD to allow for sharing. The space used for the sorting MDD depends on the degree of sharing; however, the offset computation for $s' \notin \mathcal{S}_{lumped}$ can take place at the same cost as for $s \in \mathcal{S}_{lumped}$. Both alternatives are investigated in Section 5.

Accumulation of multiple entries If a model has replicated components, the MD and MDDs have one level for each replica. Therefore, if k out of m replicas are in the same local state, any action performed by one of the k replicas will be performed by all of them, resulting in k triples (s, s', λ) that need to be summed for the matrix entry of $\mathbf{R}_{lumped}$. In the case of a single replicate operator, if k is known, we can scale λ, the rate of a, by a factor k and consider it only once. In the case of nested replicate operators, the procedure is more complicated, as we need to consider products of state-dependent scaling factors that result in a function $scale(s)$ for state s. Then, $scale(s) \cdot \lambda$ gives the corresponding entry for the lumped system.

In the current implementation, we solved the problem of scaling in a straightforward manner. We insert matrix entries that belong to the same row of $\mathbf{R}_{lumped}$ during their generation into a binary tree whose entries are ordered by column index; entries with the same column index are summed. From that tree, matrix entries are accessed by numerical analysis procedures to perform a matrix-vector multiplication by rows. The tree acts as a buffer and holds elements of a single row only temporarily.

Numerical Analysis So far, we described how to enumerate all matrix entries of $\mathbf{R}_{lumped}$ as triples $(\rho(s), \rho(min(s')), \lambda)$ by rows. Following [12], that is sufficient to allow implementation of matrix-vector multiplication $\mathbf{x} \cdot \mathbf{R}_{lumped}$, which in turn is essentially what is needed to perform iterative solution methods like the Power method or Jacobi's method for steady-state analysis and uniformization for transient analysis. However, some iterative methods require access by columns (e.g., Gauss-Seidel and SOR) or by submatrices (e.g., IAD and Takahashi's method). That suggests an open research problem on how efficiently we can enumerate the entries of $\mathbf{R}_{lumped}$ in columns or submatrices. As a side remark, we note that we can also use the current enumeration of entries by rows to create an additional, canonical MD [17] and use existing MD multiplication schemes for that canonical MD.

5. Performance Results

As stated in the introduction, the goal of this work was to create CTMC generation algorithms that simultaneously exploit the symmetries in models to reduce the number of states that need to be considered and make use of MDD and MD data structures to compactly represent the states and transitions. While the previous sections show that our approach is indeed possible from a theoretical point of view, the concrete evidence of their utility comes from their implementation and use on example models. In this section, we briefly describe the implementation we have made, and illustrate its use. The results show that symbolic generation and representation of the lumped CTMC of composed models with shared state variables are indeed practical, and enable us to solve much larger composed models than would be possible using lumping or symbolic representation techniques alone.

Implementation in Möbius. In order to test the efficiency of the developed algorithms, we implemented them within Möbius [11]. We have completed the implementation of the MDD-based state space (SS) generation, lumping algorithms, and the MD-based generation of the lumped CTMC for composed models that consist of an arbitrary number of replicate and join operators. We also implemented iterators to support numerical analysis using the Möbius state-level AFI. The SSG implementation interacts with the component models using the Möbius model-level AFI [11], thus supporting any atomic model type that Möbius supports, including stochastic activity networks, PEPA (Performance Evaluation Process Algebra), and Buckets and Balls, and accepts composed models generated by the Möbius Replicate-Join composed model editor. Since the MD-based state-transition rate matrix implementation supports the

Möbius state-level AFI [12], all the numerical solvers in Möbius that support this AFI can be used. All experiments were conducted using an Athlon XP2400 machine with 1.5 GB of main memory.

In order to develop algorithms that are efficient, there are many enhancements that are small from a conceptual point of view, but can have a large practical impact. One obvious and effective technique we used was to automatically remove levels of the MD/MDD data structures whose corresponding $\mathcal{V}_c$ sets are empty. In the second example model we describe below, this technique reduced the number of levels by about 50% and made the MD row access code, which is the most time-consuming part of the CTMC analysis, about 1.7 times faster. The other technique we used to speed up row access operation is the caching technique described in [8]. This made the row access operation 1.1 to 1.2 times faster for the second example model. This technique requires generation of an MDD representation of the (lumped) SS in which the order of the levels is the opposite of the order of the levels in the original MDD.

We now present the results from two models to illustrate the time and space characteristics of our implementation.

Courier protocol. We first consider a GSPN model of a parallel communication software system [23]. The model is parameterized by the transport window size *TWS*, which limits the number of packets that are simultaneously communicated between the sender and receiver. In order to retain a significant number of actions, we considered a model in which all actions are timed. To form a composed model, we have broken the original model into 4 atomic models, one for each of the following parts of the model: 1) the receiver's session layer, 2) the receiver's transport layer, 3) the sender's session layer, and 4) the sender's transport layer. In each of the models built by the join operator, the child models M_1 and M_2 interact with each other by sharing a subset of their SVs (i.e., places in the GSPN model), as shown in Table I. Since the replicate operator is not used in the model, lumpability induced by structure is not present, and therefore the lumping algorithm is not applied to the SS produced by the SSG algorithm for this model.

Table II shows the size of the state space and the state-space generation time for different values of *TWS*. Since the atomic models each have a nonempty set of local actions, they have some local behavior that makes use of the saturation technique.

M_1	M_2	Shared SVs
sender's sess layer	sender's trans layer	{p8, p9}
sender's trans layer	receiver's trans layer	{p23, p24, p25}
receiver's trans layer	receiver's sess layer	{p36, p37}

Table I. State variables shared among atomic models

Fault-tolerant parallel computer system. As a second test, we consider a model of a highly redundant fault-tolerant parallel computer system [16]. This model uses both replicate and join operators, and hence provides a more complete test of our algorithms and implementation. Space does not permit us to describe the model here, but a full description can be found in [19].

TWS	SS				SSG
	#	final	mem (KB)		time
	states	# nodes	final	peak	(sec)
3	$2.38\,e6$	23	4.7	13.3	0.92
4	$9.71\,e6$	29	10.7	31.0	4.36
5	$3.24\,e7$	35	23.0	67.6	20.7
6	$9.33\,e7$	41	45.3	135	83.8
7	$2.40\,e8$	47	85.4	254	347
8	$5.62\,e8$	53	151	453	1040

Table II. State-space sizes and generation times for the Courier protocol model

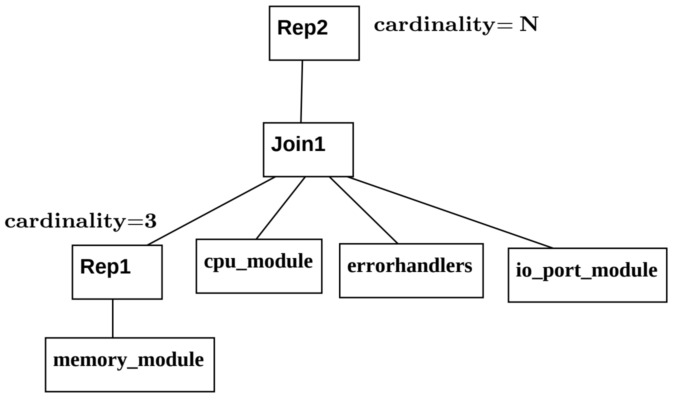

Figure 4. The composed model of the parallel computer system

We built a composed model for the entire system by first defining atomic models using the SAN formalism [20] to represent the failure of various components in the system. We then used the replicate and join operators to construct the complete composed model shown in Figure 4. The leaf nodes of the tree, which are labeled "memory_module," "cpu_module," "io_port_module," and "errorhandlers," correspond to the atomic models of the reliability of the computer's memory module, its 3 CPU units, its 2 I/O ports, and its error-handling mechanism, respectively. The memory module is replicated 3 times, which equals the number of memory modules in one computer. The replicate component is then joined with the I/O ports model, CPUs failure model, and error-handler model to create a join component that models a computer. Finally, the model of one computer is replicated N times to generate the complete composed model of the multiprocessor system.

Table III shows the sizes of the unlumped and lumped state spaces and the lumped CTMC and the total time it takes to generate the MDD representation of the state spaces and the MD representation of the lumped CTMC. The number of states, the number of MDD nodes used to represent the SS, and the amount of memory taken by the nodes in kilobytes (KB) are given for each form (i.e., lumped and unlumped) of the SS. The peak memory usage of the MDD nodes is also given. For the MD representation, the number of nodes and the memory usage of the data structure are shown under the column labelled "MD (final/peak)" since the peak values are equal to the final values for the MD representation. The last column of the

N	unlumped SS (MDD)			lumped CTMC						Total gen. time (sec)
					MDD			MD (final/peak)		
	# states	# nodes	mem (KB)	# states	final # nodes	mem (KB) final	mem (KB) peak	# nodes	mem (KB)	
1	$4.14\,e2$	14	1.46	$1.16\,e2$	18	2.06	15.0	$4.9\,e1$	3.11	0.022
2	$2.57\,e5$	43	4.54	$1.01\,e2$	340	44.9	118	$1.26\,e3$	$9.43\,e1$	0.77
3	$1.24\,e8$	99	10.3	$4.63\,e5$	1494	194	441	$5.64\,e3$	$4.12\,e2$	20.1
4	$5.50\,e10$	167	17.3	$1.48\,e7$	3395	424	1050	$1.35\,e4$	$9.97\,e2$	187
5	$2.35\,e13$	247	25.5	$3.67\,e8$	5724	705	2050	$2.46\,e4$	$1.82\,e3$	1200
6	$9.9\,e15$	339	34.8	$7.53\,e9$	8481	1040	3560	$3.89\,e4$	$2.89\,e3$	11700

Table III. Unlumped state-space, lumped state-space, and lumped CTMC sizes and generation times

table shows the total time for generating the unlumped SS, the lumped SS, and the lumped CTMC from the composed model representation. Due to the technique we use to compute lumped SS from unlumped SS, the lumping operation takes less than 0.1% of the total time for the example model. That means that the time to generate the lumped SS is essentially equal to the time to generate the unlumped SS. Note that this example is a “worst case” input for our state-space exploration algorithm. By “worst case” we mean that none of the atomic components of the model have any local action. This lack of local behavior is caused by the tight coupling that exists among the atomic models of a computer module; in terms of modeling, that coupling is realized by sharing all the SVs in the join operator of the model. Not having local actions means that techniques described in Section 3 can not be used to generate any new state based on local behavior of the atomic models. Nevertheless, the generation times reported are reasonable, and show that the memory and time required to generate the lumped CTMC are small, even for state spaces of extremely large size.

Note that the amount of memory that a lumped SS takes is larger than the amount of memory that the corresponding unlumped SS takes. That happens because from each of the equivalence classes of the state space, we eliminate all except one representative state. That causes the set of states after lumping to be less “structured” than before lumping, and hence the size of the MDD grows after the lumping operation. However, even after lumping, the size of the final MDD is still very small (< 1.1 MB) for all considered values of N. Since our goal is the numerical solution of the resulting CTMC, in addition to considering the time and space constraints on the CTMC generation, we also have to consider the limitation we have on the size of the solution vector, which grows linearly with the number of states. Therefore, reducing the number of states of the CTMC is crucial, and is a significant advantage of our technique over symbolic techniques that do not support lumping. In that respect, it is important to observe that the lumped state space does not grow as fast as the unlumped one for increasing values of N.

Finally, we measured the performance of our implementation in computing the elements of an MD-based state-transition rate matrix and compared it to the performance of a traditional sparse-matrix implementation, such as the Möbius sparse solvers. Both solvers analyze the lumped CTMC. As we are measuring the reliability of the parallel computer system, we use the uniformization method, as implemented in Möbius, for transient solution of the model. Table IV shows the sizes of the lumped CTMCs and also the solution times using two different representations: MD representation and sparse-matrix representation of the lumped CTMC.

Remember that in the case of MD representation, for each row, we need to convert each next state index to its corresponding lumped state index and merge the resulting transitions. This means that the number of transitions represented in the MD representation is larger than the number of transitions in the sparse-matrix representation. The binary tree buffer is used to accumulate those entries for the numerical analysis. For this example model, the number of transitions processed in the MD representation, i.e., the number of entries inserted into the binary tree is 39% to 42% more than the number of actual transitions, i.e., the number of transitions in the sparse-matrix representation.

N	# states	# transitions	time/ iteration (sec)	
			MD	sparse matrix
1	$1.16\,e2$	$3.80\,e2$	$6.62\,e{-4}$	$6.33\,e{-6}$
2	$1.01\,e4$	$5.51\,e4$	$1.69\,e{-1}$	$2.83\,e{-3}$
3	$4.63\,e5$	$3.51\,e6$	$1.55\,e1$	$1.75\,e{-1}$
4	$1.48\,e7$	$1.43\,e8$	$8.12\,e2$	–

Table IV. Lumped CTMC sizes and solution times (per iteration)

The available 1.5 GB of main memory in our machine limits the numerical solution. In particular, the sparse matrix solver causes thrashing of virtual memory for $N = 4$ due to the space needed for the sparse $\mathbf{R}_{lumped}$ matrix. The MD-based solver causes thrashing for $N = 5$, this time due to the space needed for the solution vectors. The size of the MD and MDD data structures is insignificant, relative to the size of the solution vectors. We exercised different implementations of the offset computation, as discussed in Section 4. In particular, we employed two methods to compute $\rho(min(s'))$: 1) using the sorting MDD and 2) sorting s' and obtaining $\rho(min(s'))$ from the MDD of $\mathcal{S}_{lumped}$. Although the latter is inferior from a conceptual point of view, it performs better for this example because of a higher locality in accesses to hardware caches. We thus present results for the second option.

6. Conclusion

Möbius provides a model composition that is built upon shared state variables by replicate and join operators. The major advantage of state variable sharing in replicate-join composed models is that the replicate operator imposes symmetries in a way that allows an associated CTMC to be lumped. In this paper, we have proposed a symbolic exploration algorithm that generates symbolic structures, namely a multi-valued decision diagram for the state space and a matrix diagram for the state-transition rate matrix of the lumped CTMC, that correspond to the composed model structure present in Möbius. We described how to obtain the lumped CTMC and how to access elements of its generator matrix $\mathbf{Q}$ without ever explicitly generating $\mathbf{Q}$ in storage. To reach our goal in building an appropriate MD, we resolved a number of technical issues including mapping state descriptors to indices, and scaling entries in the case of multiple transitions. Full integration into Möbius and further, extensive empirical evaluations of the performance and robustness of the overall approach are underway.

Acknowledgments We would like to thank Jenny Applequist for her editorial assistance.

REFERENCES

1. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, Aug. 1986.
2. P. Buchholz. Markovian process algebra: Composition and equivalence. In *Proc. 2nd Workshop on Proc. Algebras and Perf. Modelling*, volume 27 of *Arbeitsberichte des IMMD*, pages 11–30, 1994.
3. 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. on Computing*, 12(3):203, 2000.
4. J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
5. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed colored nets and symmetric modeling applications. *IEEE Trans. on Computers*, 42(11):1343–1360, November 1993.
6. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state-space generation. In *TACAS 2001*, volume 2031 of *LNCS*, pages 328–342. Springer, 2001.
7. G. Ciardo and A. Miner. Storage alternatives for large structured state spaces. In *Proc. 9th Int. Conf. Modelling Techniques and Tools for Computer Performance Evaluation*, volume 1245 of *LNCS*, pages 44–57. Springer, 1997.
8. G. Ciardo and A. Miner. A data structure for the efficient Kronecker solution of GSPNs. In *Proc. 8th Int. Workshop Petri Nets and Performance Models*, pages 22–31, 1999.
9. G. Ciardo and A. Miner. Efficient reachability set generation and storage using decision diagrams. In *Proc. 20th Int. Conf. Application and Theory of Petri Nets*, volume 1639 of *LNCS*, pages 6–25. Springer, 1999.
10. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.
11. D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius framework and its implementation. *IEEE Trans. on Soft. Eng.*, 28(10):956–969, October 2002.
12. S. Derisavi, P. Kemper, W. H. Sanders, and T. Courtney. The Möbius state-level abstract functional interface. In *Proc. of the 12th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS 2002)*, pages 31–50, 2002.
13. H. Hermanns, J. Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In *Proc. of 3rd Meeting on Numerical Solution of Markov Chains(NSMC)*, pages 188–207, 1999.
14. T. Ibaraki and N. Katoh. On-line computation of transitive closures of graphs. *Information Processing Letters*, 16:95–97, 1983.
15. K. Lampka and M. Siegle. Symbolic composition within the Möbius framework. In B. Wolfinger and K. Heidtmann, editors, *Leistungs-, Zuverlässigkeits- und Verlässlichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen, 2. MMB Arbeitsgespräch*, pages 63–74. Universität Hamburg, Fachbereich Informatik, Bericht 242, 2002.
16. D. Lee, J. Abraham, D. Rennels, and G. Gilley. A numerical technique for the evaluation of large, closed fault-tolerant systems. In *Dependable Computing for Critical Applications 2 (DCCA-2)*, pages 95–114, 1992.
17. A. Miner. Efficient solution of GSPNs using canonical matrix diagrams. In *Proc. 9th Int. Workshop Petri Nets and Performance Models*, pages 101–110, 2001.
18. B. Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. 1985 SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 147–153, 1985.
19. W. H. Sanders and L. M. Malhis. Dependability evaluation using composed SAN-based reward models. *Journal of Parallel and Distributed Computing*, 15(3):238–254, July 1992.
20. W. H. Sanders and J. F. Meyer. Reduced base model construction methods for stochastic activity networks. *IEEE Journal on Selected Areas in Communications*, 9(1):25–36, Jan. 1991.
21. A. Srinivasan, T. Kam, S. Malik, and R.E. Brayton. Algorithms for discrete function manipulation. In *Int'l Conf. on CAD*, pages 92–95, 1990.
22. W.J. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton, 1994.
23. C. M. Woodside and Y. Li. Performance Petri net analysis of communications protocol software by delay-equivalent aggregation. In *Proc. of the 4th Int. Workshop on Petri Nets and Performance Models*, pages 64–73, 1991.