Dependability Analysis with Markov Chains: How Symmetries Improve Symbolic Computations *

Michael G. McQuinn,[1] Peter Kemper,[2] and William H. Sanders[1]

[1]Coordinated Science Laboratory,
Information Trust Institute, and
Electrical and Computer Engineering Dept.
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{mmcquinn,whs}@crhc.uiuc.edu

[2]Department of Computer Science
College of William and Mary
Williamsburg, VA 23187
kemper@cs.wm.edu

Abstract

We propose a novel approach that combines two general and complementary methods for dependability analysis based on the steady state or transient analysis of Markov chains. The first method allows us to automatically detect all symmetries in a compositional Markovian model with state-sharing composition. Symmetries are detected with the help of an automorphism group of the model composition graph, which yields a reduction of the associated Markov chain due to lumpability. The second method allows us to represent and numerically solve the lumped Markov chain, even in the case of very large state spaces, with the help of symbolic data structures, in particular matrix diagrams. The overall approach has been implemented and is able to compute stationary and transient measures for large Markovian models of dependable systems.

1 Introduction

Stochastic models, in particular Markovian models, have been frequently used in the assessment of technical systems for dependability and other measures, including performability, reliability, availability, and survivability. The size of the state spaces for realistic models is often prohibitively large, such that simulation must be applied. However, much research has gone into pushing the limits of the numerical analysis of continuous time Markov chains (CTMCs). The current limits are the space used to represent the generator matrix $\mathbf{Q}$, the iteration vectors $\mathbf{p}$, and the time to compute a transient or steady-state distribution, denoted by $\pi(t)$ or π. Corresponding techniques can be broadly classified as either largeness-avoidance or largeness-tolerance techniques. The latter provide data structures and algorithms to handle large systems, where techniques include on-the-fly generation of entries of a generator matrix, symbolic and Kronecker representations, and disk-based and distributed techniques; see [11, 12, 13] for recent overviews. Symbolic techniques are the most prominent example of ways to handle $\mathbf{Q} \in \mathbb{R}^{n \times n}$ with n in the order of 10^6 and more. These techniques have been successfully applied for a space-efficient representation of matrix $\mathbf{Q}$; however, approaches that extend those techniques towards iteration vectors were not as successful. Instead, hybrid techniques with explicit representations of iteration vectors seem to be most profitable at this point. As a result, the current bottleneck for exact analysis is in the space used for iteration vectors.[1]

Largeness-avoidance techniques aim at a reduction of $\mathbf{Q}$ to a much smaller matrix $\tilde{\mathbf{Q}}$ that still allows one to compute reward values of interest. The most prominent result in the field is based on a performance bisimulation known as lumpability, and various techniques are applicable at different levels. For example, techniques exist at the level of a Markovian model, e.g., a stochastic Petri net; a term of a Markovian process algebra; the level of a compositional model for which lumping is either applied on individual submodels or on data structures representing such models, like matrix diagrams (MxDs); or, finally, at the level of the overall global state space of the CTMC of dimension n. Model-level techniques have been heavily considered for formalisms that have a composition operation based on action sharing, for example stochastic automata networks [1],

*This material is based upon work supported in part by Pioneer Hi-Bred International and France Telecom.

[1]The situation is different for approximative techniques, for which structured vector representations give more promising results.

but fewer results have been derived for a composition operation based on sharing of state variables, as for instance in [16, 15]. We will focus on the latter. An interesting issue is how avoidance and tolerance techniques can be combined to observe significant synergy effects, e.g., MxDs allow us to identify certain lumpable states independent of the modeling formalism [6]. For state sharing, [5] discusses how model-level symmetries based on replicated nodes that impose lumpability work well with MxDs and multi-valued decision diagrams (MDDs) as symbolic data structures to analyze large CTMCs with state-sharing composition.

In this paper, we investigate a combination of symbolic representations and lumping techniques for state sharing composed models. The lumping property is based on symmetries in the composition of multiple instances of models from a set of reference models. The lumping property and corresponding algorithms have been proposed in [15] for generation of an explicit, sparse matrix representation of $\mathbf{Q}$. However, a transfer of that approach to symbolic data structures is non-trivial and it is the topic of this paper. We make use of a MxD and an MDD to make entries of $\tilde{\mathbf{Q}}$ accessible to numerical analysis procedures for the transient and steady-state analysis of CTMCs.

The rest of the paper is organized as follows. In Section 2, we define basic terminology for CTMCs and symbolic data structures and recall the results on lumpability of graph composed models. In Section 3, we recall the results on building symbolic representations of the unlumped CTMC. In Section 4, we present new results on generating symbolic representations of a lumped CTMC of a graph-composed model that allow us to perform a virtual matrix-vector multiplication with $\tilde{\mathbf{Q}}$ as the fundamental step of a transient or iterative steady-state analysis of a CTMC. In Section 5 we present experiences and analysis of three example models in the Möbius modeling environment [8]. We conclude in Section 6.

2 Background and Definitions

In the following, we consider a CTMC with a state space $\mathcal{S}$ of cardinality $|\mathcal{S}| = n$, a generator matrix $\mathbf{Q} \in \mathbb{R}^{n \times n}$, and a reward vector $\mathbf{r} \in \mathbb{R}^n$. We want to compute the steady state distribution π that solves $\pi\mathbf{Q} = 0$ or the transient probability distribution $\pi(t)$ for a point in time $t > 0$ and some initial distribution $\pi(0)$. For large values of n, π is usually computed by an iterative fixed-point computation based on matrix-vector multiplications that involve $\mathbf{Q}$ and iteration vectors $\mathbf{p}(x)$, where x indicates the x-th iteration step. It is common to consider $\mathbf{Q} = \mathbf{R} - \mathbf{D}$, with $\mathbf{R}$ being a rate matrix of off-diagonal values, and $\mathbf{D}$ a diagonal matrix of row sums of $\mathbf{R}$. A multitude of numerical procedures are known [17], for which the Power method, the Jacobi method, and the Gauss-Seidel method may serve as examples of particularly simple procedures for steady-state analysis. A transient distribution is usually computed with uniformization, which also requires a matrix vector multiplication that involves $\mathbf{Q}$ and iteration vectors as a basic step.

We consider an example model to help to illustrate the concepts.

Example 1. *Let A be a Markovian model with 3 state variables and 4 states. We consider two instances A_1, A_2 of A. An instance A_i has variables v_{i1}, v_{i2}, and v_{i3} of domain $\{0,1\}$ and states $(0,0,1), (0,1,1), (1,0,0)$, and $(1,1,0)$. For a given instance A_i, the values of state variables v_{i1} and v_{i3} sum to 1 (i.e., $v_{i1} + v_{i3} = 1$), while the state variable v_{i2} will be affected by another model instance. The possible state transitions are $(0, v_{i2}, 1) \to (1, v_{i2}, 0)$ with rate $(1 + v_{i2})\mu$ and $(1, *, 0) \to (0, *, 1)$ with rate λ. A_1 and A_2 are composed by sharing variables in the following way: $v_1 = v_{11} = v_{22}$ and $v_2 = v_{12} = v_{21}$, where v_1 and v_2 are simply placeholder variables to help the discussion. The overall effect is that a submodel i being in a state where $v_{i1} = 1$ (e.g., being in a failure state) increases the rate at which the other submodels reaches the same state. Let $(v_1, v_2, v_{13}, v_{23})$ be the order of variables for a state of the composed model. Then the state space $S = \{(0,0,1,1), (0,1,1,0), (1,0,0,1), (1,1,0,0)\}$, and states of S will be denoted by $s_1, \ldots, s_4$ in that order. The generator matrix is:*

$$Q = \begin{pmatrix} -2\mu & \mu & \mu & 0 \\ \lambda & -\lambda - 2\mu & 0 & 2\mu \\ \lambda & -\lambda - 2\mu & 0 & 2\mu \\ 0 & \lambda & \lambda & -2\lambda \end{pmatrix}$$

Figure 1(a) shows the corresponding CTMC.

Since CTMCs of interest are often very large, much research has gone into finding ways to reduce $\mathbf{Q}$ to a smaller matrix $\tilde{\mathbf{Q}}$ with a state space $\tilde{S}$ of cardinality $\tilde{n}$. A key concept is that of lumpability (ordinary, exact), which can be formalized by defining an equivalence relation R on S that implies $\tilde{n}$ equivalence classes and a surjective mapping function $rep : \mathcal{S} \to \tilde{S}$ such that sRs' implies $rep(s) = rep(s')$ for all $s, s' \in S$. The generator matrix of the lumped CTMC is then defined as $\tilde{\mathbf{Q}} = \mathbf{WQV}$ with a collector matrix $\mathbf{V} \in \{0,1\}^{n \times \tilde{n}}$, such that $\mathbf{V}(i,j) = 1$ if $s_j = rep(s_i)$ and 0 otherwise. Note that we assume an arbitrary but fixed indexing of S and $\tilde{S}$ such that we can use i and s_i interchangeably. Matrix $\mathbf{W} \in \mathbb{R}^{\tilde{n} \times n}$ is called a distributor matrix and is defined as a non-negative matrix where $\mathbf{W}(i,j) \geq 0$ if $s_i = rep(s_j)$ and 0 otherwise, where each row sums to 1, and where for any $s_i \in \tilde{\mathcal{S}}$ there exists at least one positive entry $\mathbf{W}(i,j) > 0$. We define $\tilde{r} = \mathbf{r}\mathbf{W}^\top$ and $\tilde{\mathbf{p}}(0) = \mathbf{p}(0)\mathbf{V}$. If R fulfills the requirements for lumpability, then it is known that a solution $\tilde{\pi}(t)$, resp. $\tilde{\pi}$ is sufficient to compute rewards of the original, unlumped CTMC. For instance, the conditions for ordinary lumpability are that $s, s' \in R$ implies $\forall s'' \in S$, $\sum_{rep(s'')=rep(c)} \mathbf{Q}(s,c) = \sum_{rep(s'')=rep(c)} \mathbf{Q}(s',c)$ and $\mathbf{r}(s) = \mathbf{r}(s')$; see [1] for more details.

Example 2. *Continuing Example 1, states s_2 and s_3 can be lumped according to ordinary lumpability, which yields*

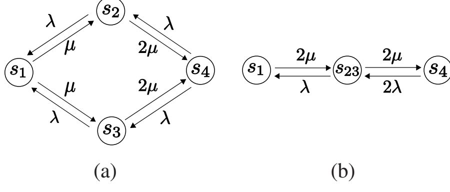

Figure 1. (a) Unlumped and (b) Lumped CTMC

the lumped CTMC shown in Figure 1(b). Also, the relation rep(s) gives the following mapping: $rep(s_1) = s_1$, $rep(s_2) = s_3$, $rep(s_3) = s_3$, *and* $rep(s_4) = s_4$. *We select* s_3 *as the representative of the equivalence class* $\{s_2, s_3\}$, *which is renamed index* s_{23} *in Figure 1(b). The corresponding matrices are:*

$$\mathbf{V} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \mathbf{W} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The definition of the distributor matrix $\mathbf{W}$ *allows us to select values* $\mathbf{W}(2,2) \geq 0$, $\mathbf{W}(2,3) \geq 0$, *such that* $\mathbf{W}(2,2)+\mathbf{W}(2,3) = 1$ *for lumpability. We select one state as a representative, namely* s_3, *and assign* $\mathbf{W}(2,3) = 1$.

In the following, we always choose $\mathbf{W}$ to have exactly 1 non-zero entry per row, namely $\mathbf{W}(i,j) = 1$ iff $s_i = rep(s_j)$. This is one way to define $\mathbf{W}$ for ordinary lumpability and relates to a projection of $\mathbf{Q}$ to those rows that correspond to representative states.

The generic framework and matrix notation for lumping is known [1, 14], so the following points need to be addressed:
1. How to define and compute rep for a model that is specified in a particular modeling formalism,
2. How to define and represent matrix $\tilde{\mathbf{Q}}$, and
3. How to perform a matrix vector multiplication based on the representation of $\tilde{\mathbf{Q}}$.

To begin to answer these questions, we start with a discussion on lumpability based on symmetries in a composed model.

Graph composed models

We consider models that are composed of other models by sharing state variables. Our approach applies to a number of modeling formalisms, as it is a common practice in Möbius[8], so no specific formalism is presented here. Any formalism that has the following properties is sufficient (see [15] for a formal definition of models, states and state transitions):
1. Composition through arbitrary graph connections
2. Composition through state sharing
3. Global actions are of the lowest priority (i.e., timed actions)
4. Each model must have a type property indicating which reference model it is derived from.

We can represent the composition of models with a particular undirected graph $G = (V, E)$ in such a way that symmetries in the graph correspond to lumpability in the associated CTMC. A *model composition graph* (MCG) of the composed model is a undirected graph that represents the separation of state variables that are shared and those that are privat. For a complete discussion, see [15], but the following example is sufficient for this discussion. Each model is decomposed into private and shared state fragments, and edges connecting two fragments correspond to a relationship imposed from the graph structure. See Figure 2 which is the MCG for the running example. Here, each of model A_1 and A_2 has a single private state fragment represented by a square and two shared state variable fragments represented by circles. Labels A, ν_1, ν_2, and ϵ are positioned next to their corresponding nodes. The partition $\Xi = \{\xi_1 = \{v_{13}, v_{23}\}, \xi_2 = \{v_{11}, v_{21}\}, \xi_3 = \{v_{12}, v_{22}\}, \xi_4 = \{v_1, v_2\}\}$ defines which nodes have equal labels.

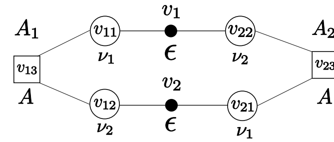

Figure 2. Model composition graph

Let Γ be the set of all permutations on G that permute the nodes of the graph such that the connectivity and labeling remain invariant.[2] From [15], it is known that symmetries in the MCG induce lumpability of the associated CTMC. The MCG carries information on which submodels are equal. In the running example, A_1 and A_2 are both instances of A, and the shared variables of each submodel are the same. In the example, v_{11} and v_{21} are instances of the same variable, as are v_{12} and v_{22}. In addition, the MCG represents the binding/merging of shared variables. In the example, v_{11} is merged with v_{22}, and v_{12} is merged with v_{21}. The graph of the MCG in Figure 2 has two symmetries (plus the identity), of which only one is relevant. The first symmetry mirrors nodes across a vertical line through the connection nodes, mapping A_1 onto A_2, v_{11} onto v_{22}, and v_{12} onto v_{21}. However, this is not consistent with the partitioning (labeling) of nodes, since v_{11} does not match with v_{22}. This symmetry does not impose lumpability and is not considered, since it violates the partitioning Ξ. The second symmetry rotates A_1 onto A_2, maps v_{11} onto v_{21}, and maps v_{12} onto v_{22}, which conforms with Ξ, i.e., nodes that are mapped have

[2] Labels are also called colors in graph theory.

equal labels, so this symmetry does induce lumpability and is correctly taken into account.

The corresponding permutation for this symmetry:

$$\rho_1 = \begin{pmatrix} v_1 & v_2 & v_{13} & v_{23} & v_{11} & v_{12} & v_{21} & v_{22} \\ v_2 & v_1 & v_{23} & v_{13} & v_{21} & v_{22} & v_{11} & v_{12} \end{pmatrix}$$

where we ordered the variables by V_3, V_1, V_2 such that the first four variables actually represent a state of the overall model, and the values of the final four variables are redundant for a state description.

The key observation for lumpability is that state variable sharing in the absence of global priority schemes allows each individual submodel to perform given that the local and shared state variables are set. Consequently, permuting local state information among equal submodels does not change the overall behavior of the stochastic process (if state information of connected submodels is also permuted in a consistent manner). Those constraints are taken care of by the permutations of the MCG. We briefly recall the main definitions and results of [15] to establish lumpability.

Definition 1. *L is a relation such that for two composed model states μ_1 and μ_2, $\mu_1 L \mu_2$ if there exists a $\gamma \in \Gamma$ such that $\mu_2 = \gamma(\mu_1)$.*

L is an equivalence relation. Let $\tilde{S}$ with $|\tilde{S}| = \tilde{n}$ denote the set of equivalence classes that L imposes on state space S of a model and $rep(s) \in \tilde{S}$ indicate a state of S that we select as a representative to identify a particular equivalence class of S. Also, it is assumed that permutations map a state μ_1 to μ_2 only if the reward values of μ_1 and μ_2 are equal. In practice, this is a minor restriction since most often rewards can be written in such a way that they are unaffected by permutations. For example, if a reward is defined on a state μ_1 that is lumped with state μ_2, a new reward could be defined on both of the states μ_1 and μ_2. Then, the original reward could be derived from the modified reward (by perhaps dividing by the size of the equivalence class).

Theorem 1. *For a given model and relation L, the matrix $\tilde{\mathbf{Q}} \in \mathbb{R}^{\tilde{n}\times\tilde{n}}$ with $\tilde{\mathbf{Q}}(i,j) = \sum_{s_j = rep(s_k)} \mathbf{Q}(i,k)$ for $s_i, s_j \in \tilde{S}$ results from an ordinary lumping of $\mathbf{Q}$. See [15] for the proof.*

Note that L does not necessarily yield the coarsest lumping that exists for $\mathbf{Q}$, but it does yield the coarsest one based on symmetries present in the MCG. L and function rep can be automatically computed for a MCG. The procedure is based on computational group theory as devised in [15], and will be discussed in some detail in Section 4.

Symbolic data structures for CMTCs

The equation $\tilde{\mathbf{Q}} = \mathbf{WQV}$ suggests two possibilities for numerical analysis. One can compute $\tilde{\mathbf{Q}}$, store it in an appropriate data structure, and perform an iterative solution method as for any CTMC, which is the approach in [15]. Alternatively, one may represent $\tilde{\mathbf{Q}}$ as $\mathbf{WQV} = \mathbf{WRV} - \tilde{\mathbf{D}}$ with individual data structures and perform a virtual iterative solution for $\tilde{\mathbf{Q}}$ based on the four structures $\mathbf{W}$, $\mathbf{R}$, $\mathbf{V}$, and $\tilde{\mathbf{D}}$. The latter is not immediate, but if we take into account that $\mathbf{Q}$ often has a very regular structure such that it is very amenable to a symbolic representation while $\tilde{\mathbf{Q}}$ is much more irregular, then this guides us to develop a representation of $\mathbf{WQV}$ with symbolic data structures, namely an MxD for $\mathbf{Q}$, resp. $\mathbf{R}$, MDDs for $\mathbf{V}$ and $\mathbf{W}$, and a vector for diagonal entries of $\tilde{\mathbf{D}}$.

MDDs and MxDs are members of a family of data structures named decision diagrams that are rich in variation, theory, and terminology. Within the given space, we can only briefly recall essential aspects that we use here, so for further information we refer to two recent overviews in the context of Markov chain analysis with symbolic and structured representations [13, 11]. MDDs and MxDs are both rooted directed acyclic graphs. Both are associated with K variables $v_1, \ldots, v_K$. An *MDD* encodes a function $f : \times_{k=1}^{K} S_k \rightarrow M$ where $S_k = \{0, 1, \ldots, |S_k| - 1\}$ is the finite set of values that variable v_k can have[3] and M is the finite set of values of f. An *MxD* is similar and encodes function $g : \times_{k=1}^{K} (S_k \times S'_k) \rightarrow M$. We will focus on the special case of MDDs and MxDs that are quasi-reduced, are ordered, and have boolean outputs. Such diagrams have non-terminal nodes associated with a variable and terminal leaf nodes that carry the function value.

For the boolean case, it is common to remove all arcs and entries that end up in the terminal node for "false" such that after that reduction, the remaining terminal node for "true" with its incoming arcs can safely be omitted as well. The resulting MDD and MxD graphs represent only those input parameter combinations of f, resp. g, that yield "true" as a path in the graph from its root node to its nodes at level K (the new leaf nodes after removal of the non-terminal nodes). Let $mdd[]$ denote the root node of the graph, and $mdd[](s_1)$, for some $s_1 \in S_1$ that is present in $mdd[]$, denote the edge starting at value s_1 that leads to a node at level 2. Let $mdd[s_1, \ldots, s_{i-1}]$ denote the node at level i that is reached along corresponding edges in the graph starting at the root node. Analogously, we define $mxd[]$, $mxd[](s_1, s'_1)$, and $mxd[(s_1, s'_1), \ldots, (s_{i-1}, s'_{i-1})]$. For simplicity of notation, we assume that the MxD does not have multiple entries at entry (s_i, s'_i) at any node at level $0 < i \leq K$. Of course, it is possible to consider multiple entries using formal sums, and we do so in our implementation, but it clutters the formal representation without offering much additional insight.

MDDs and MxDs shall both carry a numerical value per

[3]To allow for arbitrary variables of finite domain, one can consider S_k to an index set for that domain.

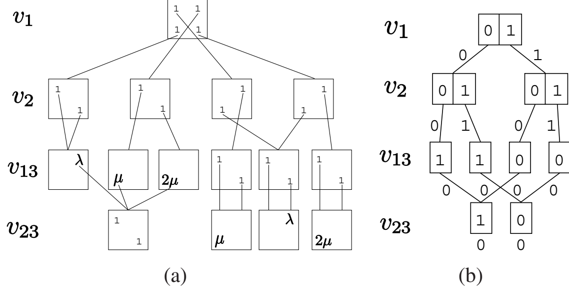

Figure 3. (a) MxD representation of unlumped CTMC (b) Mapping MDD

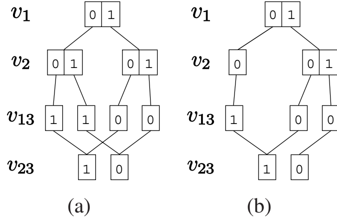

Figure 4. (a) Unlumped and (b) Lumped State Space

edge that makes the MDD a so-called edge-valued MDD (EV^+MDD). Let $w(mdd[s_1,\ldots,s_{i-1}](s_i))$ denote the numerical value for the corresponding edge. Leaf nodes at level K shall carry a weight as well (to the omitted terminal node "true"). For an MDD, the weight of a path $(s_1,\ldots,s_K)$ is defined as:

$$\sum_{k=1}^{K} w(mdd[s_1,\ldots,s_{k-1}](s_k))$$

while for an MxD, we define the weight of a path $((s_1,s'_1),\ldots,(s_K,s'_K)$ as:

$$\prod_{k=1}^{K} w(mxd[(s_1,s'_1),\ldots,(s_{i-1},s'_{i-1})](s_i,s'_i))$$

Decision diagrams gain their efficiency from representing isomorphic (equal) subgraphs only once, i.e., the function can be seen as a graph that starts as a tree and is reduced by sharing equal subtrees (sub-DAGs).

Example 3. *We can represent Q and $\mathcal{S}$ of the unlumped CTMC by a MxD and an MDD. Figures 3(a) and 4(a) shows the MxD and MDD representations of the running example. Note that $K = |V_1| + |V_3|$.*

The state space of the lumped CTMC can be obtained from the MDD of the unlumped CTMC through removal of all paths that correspond to states that are not representative of its equivalence class. Removal of path $(0,1,1,0)$ yields the MDD of the lumped CTMC as shown in Figure 4(b).

3 Generation of unlumped state space and generator matrix

Based on the graph composed modeling technique discussed in the previous section, we now present methods to generate an MDD representation the unlumped state space $\mathcal{S}$ and a MxD representation of the generator matrix $\mathbf{Q}$. In section 4, we then present methods to generate representations of the lumped state space $\tilde{\mathcal{S}}$ and generator matrix $\tilde{\mathbf{Q}}$.

Recall that submodels in the graph composed model are connected through sharing of state variables. We define an action as *global* if it depends on a shared state variable; otherwise, it is defined as *local*. An action depends on a state variable if its enabling condition or affecting conditions contain this state variable. A local action depends only on a single level in the MDD, and that level is the only one affected by its firing. A global action may depend on and affect at most $k+1$ levels in the MDD structure, where k is the number of shared variables of the model that performs the global action. For that reason, we differentiate local and global actions when generating the symbolic unlumped state space [5]. Based on that distinction, we create a symbolic representation of the state-space $\mathcal{S}$ and generator matrix $\mathbf{Q}$ by using known techniques from symbolic state space exploration for action-sharing formalisms [10], and state-sharing formalisms [5].

3.1 Unlumped state space

Given the distinction between local and global actions, we can extend the state-space generation routine for rep-join composed models of [5] to operate on graph composed models. Due to space constraints, we omit the details and instead focus on the differences between the two algorithms.

For rep-join composed models, the composition naturally forms a tree structure that can be mirrored in the MDD and MxD representations. It suggests a certain order of variables for symbolic data structures. For graph-composed models, this natural representation no longer exists because of potential cycles in the composition. Instead, we must determine an ordering of submodels to become the ordering of variables of the MDD and MxD structures. One way to proceed is to perform a DFS on the graph-composed model, marking each node as it is found. The order in which nodes are visited then forms an ordering for the symbolic data structures. It is known that the order of variables has an impact on MDDs and MxDs, however at this point we do not investigate the effect of different variable orderings on the size of MDDs and MxDs. Also, due to the symmetries in the MCG, we can easily determine when two instances of a particular model are equivalent in the composed model. The equivalence could be used to share structures across different levels of the MDD or MxD, but this potential for optimization is also not addressed in this paper.

3.2 Rate matrix

The procedure to build a MxD representation of the rate matrix is similar to that in [5] and mimics what has been done to generate the symbolic representation of the state space. Instead of operating on vectors at each level of the MDD that represent portions of the state space, we operate on matrices at each level of the MxD that represent portions of the transition rate matrix. The entry (s_i, s'_i) of a matrix at level i of an MxD, say at node $mxd[(s_1, s'_1), \ldots, (s_{i-1}, s'_{i-1})]$ in the MxD, points to a matrix, a node, at level $i+1$ if there is a transition in the model from a state s_i to a state s'_i and $i < K$. At the bottom level K, non-zero matrix entries indicate a transition. Local matrices are kept for each level of the MxD, which can easily be computed during state-space generation. When a state s is found to reach state s' with rate λ, we simply update the entries of the corresponding MxD levels. Note that the differentiation between local and global actions still applies. When a local action fires, only MxD nodes in one level will be affected, whereas a global action could affect many MxD levels. See [5] for a complete discussion.

4 Projection on lumped state space

At this point, a symbolic state-space generator could be built using an MDD representation of $\mathcal{S}$ and a MxD representation of $\mathbf{Q}$, and known MD-based numerical analysis techniques could be applied [3]. But, because of the high memory costs associated with iteration vectors and the high performance cost of working with the larger $\mathbf{Q}$ matrix, we investigate a much smaller representation of the Markov chain.

In Section 2, we recalled results that showed that when an equivalence class is replaced by a single representative state, the resulting Markov chain is equivalent to the original Markov chain in that it is possible to compute the same rewards. Essentially, we showed that two composed model states are lumpable if they belong to the same equivalence class. In graph-composed models, equivalence is defined through Γ, the set of automorphisms of the model composition graph. An equivalence class is a set of states such that each is pairwise symmetric to every other member of the set.

To produce the smaller Markov chain representation, we need to be able to perform three functions: 1) find a representative for an equivalence class, i.e., determine $rep(s)$, 2) obtain an MDD representation of $\tilde{\mathcal{S}}$ as a subset of $\mathcal{S}$, and 3) provide a mapping from any state $s \in \mathcal{S}$ to the state $rep(s)$. Our goal is to represent the matrix $\tilde{\mathbf{Q}}$ as $\mathbf{WRV} - \tilde{\mathbf{D}}$, where $\mathbf{W}$ and $\mathbf{V}$ are the distributor and collector matrices and $\mathbf{R}$ is the rate matrix computed in the previous section. A solution to these three steps enables us to perform a virtual matrix vector multiplication for $\tilde{\mathbf{Q}}$ based on $\mathbf{WRV}$ and $\tilde{\mathbf{D}}$.

4.1 Obtaining an MDD for all representative states

The first step is to find a representative state for each equivalence class, which can be used to generate matrix $\mathbf{W}$. A state may only be in one equivalence class, so finding a representative for each class is equivalent to finding $rep(s)$, the representative state for each $s \in \mathcal{S}$. In many composed modeling formalisms, that is done by minimizing (or maximizing) the lexicographical order of state variables. The situation is different, however, for graph-composed models, since the permutations are restricted to those in the MCG. For instance, in Example 1, finding the maximum of $\{v_{11}, v_{21}\}$ restricts the remaining permutations to the set $\{$identity, $\rho_1\}$ rather than the full $2^3 = 8$ mappings, so a more complex procedure is necessary.

To generate the *rep*(s) procedure, we must first find the *automorphism group* Γ of the MCG, or the set of symmetries available in the composed model. A *symmetry* (or *permutation*) maps each variable fragment (public or private) to another fragment that is in the same partition of Ξ. Two states s_2 and s_3 are in the same equivalence class if s_2 can be mapped to s_3 through a series of one or more permutations. In Example 1, states s_2 and s_3 are in the same equivalence class, since permutation ρ_1 maps s_2 to s_3. The following approach is based on algorithms for permutation groups, and we recommend [2] for a comprehensive textbook that describes the fundamental algorithms that we use. In order to find the set of symmetries in our MCG, we make use of *saucy* [4], a well-known and efficient implementation for finding the automorphism group of a graph.

Given an automorphism group, we compute a stabilizer chain represented as a base and strong generating set. A *strong generating set* is a subset of the automorphism group such that it can be expanded to generate the entire group by repeated application of its permutations. Each automorphism in the strong generating set "stabilizes" a particular node in the MCG, meaning subsequent elements in the set will never move it again. The series of "stabilized" nodes forms a set called the *base*. We use an implementation of the Schreier-Sims algorithm to find a base and strong generating set of the automorphism group returned from saucy.

The base and strong generating set give us the required means to "sort" states to find a representative state for each state $s \in \mathcal{S}$. For each element in the base b_i, we find the permutation that moves the largest state variable fragment to b_i. By the definitions of base and strong generating set, we know that the state variable fragment at b_i will never be permuted again. By applying this procedure for each element in the base, we have defined a method to "sort" permutations to find a representative state that is maximal wrt a lexicographic ordering of state variables.

The remaining problem is to find the set of permutations

REP$(s_1, \ldots, s_K)$
1 $B \leftarrow$ Base for stabilizer chain
2 **for** each $b_i \in B$
3 **do** $O^{(i)} \leftarrow$ Orbit of base element b_i
4 $j \leftarrow$ index of the vertex in $O^{(i)}$ with the largest state s_j (ties to smallest index)
5 Apply Permutation to move s_j from $O^{(i)}(j)$ to b_i

Figure 5. Pseudocode for finding a $rep(s)$

that move a state variable fragment to b_i. This can easily be done using a factorized inverse transversal (or Schreier Tree) of the strong generating set. A *Schreier Tree* is a tree rooted in the base element b_i with nodes representing the orbit of b_i. An edge from n_1 to n_2 in the tree represents a permutation that moves a node in the n_1-th position to the n_2-th position. All paths from a node to root correspond to a set of permutations that move an element in the orbit of b_i to b_i, which is needed in the sorting procedure.

See Figure 5 for a procedure that finds the representative state $rep(s)$ of a state $s \in \mathcal{S}$ represented symbolically in an MDD. Unlike the previous description, in this procedure we sort state indices rather than state variable fragments. Since the mapping from a composed state to a set of indices into local state spaces is one-to-one, we can sort the indices instead of the state fragments without any loss of generality. We do this as an optimization, since the size of the indices is most often much smaller than the size of a state. In Figure 5, line 2 "stabilizes" a state index for each element in the base, while lines 3-5 apply the permutation that maximizes the state index to be stabilized.

Once we have developed a procedure to evaluate $rep(s)$ for $s \in \mathcal{S}$, we see at least two ways to generate an MDD representation of the states representing $\tilde{S}$, the lumped state space. The first one enumerates all paths (state indices) in an MDD of $\mathcal{S}$, evaluates $\tilde{s} = rep(s)$, and inserts $\tilde{s}$ into a new MDD for $\tilde{S}$. See Figure 6 for pseudocode. The method GenPath generates an m-level MDD representing the representative state, which is added to $\tilde{S}$.

An alternative method prunes an MDD representation of $\mathcal{S}$ to obtain a representation of $\tilde{S}$, the lumped state space. Using a DFS procedure, one can prune a sub-MDD if the path to the sub-MDD can not be part of a representative state. Once an MDD of $\tilde{S}$ has been achieved, we need to add edge-values for the weight function. This procedure is known and is performed without changing the structure of the MDD [13].

DFSLUMP$(\mathcal{S}_{MDD})$
1 $\tilde{S}_{MDD} \leftarrow$ empty MDD
2 **for** all $(s_1, \ldots, s_K)$ in $\mathcal{S}_{MDD}$
3 **do** $(\tilde{s_1}, \ldots, \tilde{s_K}) = $ REP$(s_1, \ldots, s_K)$
4 $\tilde{S}_{MDD} \leftarrow \tilde{S}_{MDD} \cup$ GENPATH$(\tilde{s_1}, \ldots, \tilde{s_K})$

Figure 6. Pseudocode for finding $\tilde{S}$

4.2 Obtaining a mapping from all states to IDs of representative states

If we examine the $(\tilde{S} \times \mathcal{S})$ matrix $\mathbf{WQ}$ and take into account how we defined $\mathbf{W}$, we see that it represents the projection of the rows of $\mathbf{Q}$ to the lumped state space $\tilde{S}$. We are able to mimic this algorithmically with the help of an MDD of $\tilde{S}$ that guides us to select corresponding row entries from an MxD of $\mathbf{Q}$. The problem is that the entries of $\mathbf{WQ}$ correspond to rates from representative states to reachable but (possibly not) representative states.

There are two orthogonal methods to generate a matrix $\mathbf{V}$ that maps a state to its representative state. We could either perform this computation on-the-fly, or we precompute the values and simply look them up. While both perform the same function, the two methods' characteristics vary greatly. For one thing, the two approaches are very different in their space and time complexity. The on-the-fly procedure uses minimal memory (a vector of state variables and a set of permutations) and more computations, while the lookup technique uses fewer computations but additional space for a data structure to represent the mapping. However, this trade-off need not result in higher computation times for the on-the-fly approach, since that approach promises a better locality and thus may benefit from faster memory access times present in the memory hierarchy of modern CPUs. We now describe, in more detail, the space and time tradeoff of the two approaches.

On-the-fly calculation

The on-the-fly calculation is performed much like the $rep(s)$ procedure described in Figure 5. Whenever a solver requests a particular matrix entry $\mathbf{WR}(i, k)$ with s_k mapped to $rep(s_k)$, then we need to call procedure $rep(s_k)$ to find the representative state, whose weight value is returned with the appropriate rate.

The obvious downside to this procedure is that $rep(s)$ will be called for each nonzero entry of $\mathbf{R}$ in each iteration step of an iterative solution method. This cost may not be prohibitively large, however, since the structures may fit into the lowest-level cache of a modern CPU, which reduces the memory access costs. Another benefit of this

```
mult(p(x), p(x+1), mdd, mxd)
1  for all (s_1, ..., s_K) in mdd
2      do for all ((s_1, s'_1), (s_2, s'_2), ..., (s_K, s'_K)) in mxd
3          do i ← weight(mdd, s_1, ..., s_K)
4             R_ij ← weight(mxd, (s_1, s'_1), ..., (s_K, s'_K))
5             (a1) s̃ ← rep(s'_1, ..., s'_K)
6             (a2) j ← weight(mdd, s̃_1, ..., s̃_K)
7             (b1) j ← weight(mdd', s'_1, ..., s_K)
8             p_j(x+1) ← p_j(x+1) + R_ij · p_i(x)
```

Figure 7. Pseudocode for matrix-vector mult.

approach is that it uses very little memory, leaving most of the system's resources for costly iteration vectors. With space-efficient symbolic data structures, the limiting factor is usually memory for iteration vectors, so this approach is expected to scale better for larger state spaces.

Mapping MDD

An alternative method is to precompute and store the $weight(mdd, rep(s))$ for all $s \in \mathcal{S}$ to avoid the on-the-fly evaluation of $rep(s)$. For the given context, a symbolic representation by a particular $EV^{+}MDD$, similar to the mapping MDD described in [5] is a promising data structure. The key idea is to encode a mapping $weight(mdd, rep(s))$ as a single, additional $EV^{+}MDD$ mdd′ with edge values appropriately set. Whenever a solver requests a particular matrix entry $\mathbf{WR}(i,k)$ with s_k mapped to $rep(s_k)$, we could compute the column index as $j = weight(mdd', s_k)$ for any $s_k \in S$ with $s_j = rep(s_k)$. See Figure 3(b) for an illustration of the mapping MDD for the running example.

4.3 Matrix vector multiplication

The remaining problem is to perform a matrix-vector multiplication such that existing numerical techniques can be applied to find the rewards of interest. Consider an algorithm for matrix vector multiplication $\mathbf{p}(x+1) = \tilde{\mathbf{Q}} \cdot \mathbf{p}(x) = \mathbf{W} \cdot (\mathbf{R} - \mathbf{D}) \cdot \mathbf{V} \cdot \mathbf{p}(x) = \mathbf{W} \cdot \mathbf{R} \cdot \mathbf{V} \cdot \mathbf{p}(x) - \tilde{\mathbf{D}} \cdot \mathbf{p}(x)$. Given our MxD representation of $\mathbf{R}$ and $rep(s)$, it is straightforward to compute $\tilde{\mathbf{D}}$. Since $\tilde{\mathbf{D}}$ can be represented by a vector of dimension $\tilde{n}$ such that a multiplication with $\mathbf{p}(x)$ is trivial,[4] we focus on $\mathbf{p}(x+1) = \tilde{\mathbf{R}} \cdot \mathbf{p}(x)$.

In the algorithm presented in Figure 7, Line 1 enumerates all states $(s_1, \ldots, s_K) \in \tilde{S}$, and corresponds to an enumeration of all nonzero entries of a particular distributor matrix $\mathbf{W}$ with $\mathbf{W}(i,j) = 1$ iff $s_i = rep(s_i)$ and $rep(s_i)$ has index j in $\tilde{S}$. Line 2 takes the corresponding matrix entries of $\mathbf{R}$ into account, where rows correspond to representative states $s \in \tilde{S}$, i.e., those selected in line 1. Line 3 computes the index position of state $(s_1, \ldots, s_K)$ in $\mathbf{p}(x)$ from the weight function encoded in the MDD. Line 4 computes the matrix entry for a state transition (s,s') in the unlumped matrix $\mathbf{R}$. Lines 5 and 6 compute the index position j in $\mathbf{p}(x+1)$ by first computing the representative state $\tilde{s}$ for $(s'_1, \ldots, s'_K)$ and then computing the weight of $\tilde{s}$ from the MDD, in the same manner as in line 3. Alternatively, one can employ a mapping MDD mdd′ that encodes the weight of the representative state $\tilde{s}$ for a state s in a new $EV^{+}MDD$ mdd′ which is described in line 7. The latter variant obviously saves the evaluation of $rep(s')$ that the former must do in line 5. Finally, the multiplication takes place in line 8. Note that line 8 also performs the on-the-fly aggregation of edge values by summation, which is encoded in matrix V, i.e. we perform $\mathbf{p}_j(x+1) = \mathbf{p}_j(x+1) + \tilde{\mathbf{R}}(i,j)\mathbf{p}_i(x+1)$ as $\mathbf{p}_j(x+1) = \mathbf{p}_j(x+1) + \sum_{j=rep(k)} \mathbf{R}(i,k)\mathbf{p}_i(x+1)$ in line 8. The pseudocode sketches what needs to be done; an actual implementation typically has $2 \cdot K$ nested for-loops: one for each level in the MDD and one for each level in the MxD. The implementation mimics a DFS and implies that partial sums and products for the weight functions are reused, and look-ups for paths are reduced.

5 Example and results

In this section, we present results that provide insight into the details of the symmetry detection and symbolic state-space generation algorithms through the use of three example models. Our algorithms implement the Möbius model-level and state-level AFIs [8, 7], which provides us with a rich set of modeling formalisms and solution techniques. Specifically, we wish to examine the details of the symmetry detection routines, including the size of the automorphism group and associated Schreier Trees. In addition, we want to examine the costs of the MxD and MDD implementations, especially the memory requirements for large models. Because this technique is designed to minimize memory usage, only the on-the-fly mapping of states to representatives is considered, but future implementations may find that mapping MDDs are useful for models with smaller memory constraints. We first study a fault-tolerant multi-processor system and then study the dependability of a Lower Earth Orbitting (LEO) satellite network and a fault-tolerant parallel computer system.

Figure 8 gives a diagram of our first example system and its associated MCG; for a complete description, see [15]. The graph-composed model contains fully connected computing clusters, where each cluster comprises three distinct components: a router, a CPU, and an I/O unit. We can easily produce variations of this model by changing the number of clusters or the number of components in an individual clus-

[4] A more sophisticated, space-efficient technique is to use an MTMDD to represent $\tilde{D}$.

ter. A configuration of the model is specified by the number of routers, CPUs, and I/O units.

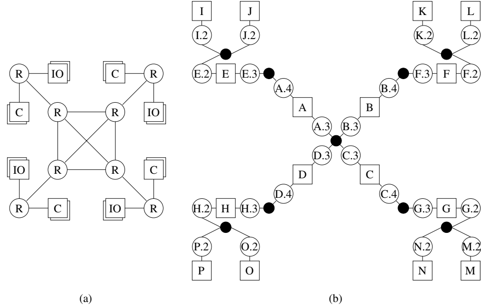

Figure 8. (a) Network with fully connected core and (b) Model composition graph

Table 1 summarizes the reduction in state size for different variants of the model. For example, the smallest configuration $(4, 2, 2)$ corresponds to the configuration of 4 clusters, 2 CPUs, and 2 I/O units. The second column gives the cardinality of the unlumped state space $\mathcal{S}$, while the third gives that of the lumped state space $\tilde{S}$. The fourth column gives the size of the automorphism group, an upper bound on the state space reduction. As the number of components per cluster grows, the size of the automorphism group grows as well, because new symmetries exist between components in a single cluster. Similarly, as the number of clusters increases, the size of the automorphism group grows, because of increased symmetry between different clusters. Due to lumping from model-level symmetry induced by the model composition graph, for some of the variants we are able to reduce the size of the state space to nearly 2%. In [15], we were limited to studying the (8,4,4) configuration because of the high space costs associated with the explicit sparse matrix representation, but by using symbolic data structures we are able to study configurations with hundreds of millions more states.

Table 1. Reduction due to Lumping. Configuration (routers, CPUs, I/O)

#	Config.	Unlumped State Space	Lumped State Space	Size of Γ
1	(4, 2, 2)	3,600	1,830	2
2	(6, 3, 3)	216,000	37,820	6
3	(6, 6, 3)	2,985,984	224,841	48
4	(6, 6, 6)	37,933,056	835,983	384
5	(8, 4, 4)	12,960,000	1,406,294	24
6	(8, 6, 4)	74,649,600	8,652,240	16
7	(8, 6, 6)	406,425,600	22,668,210	64
8	(8, 8, 4)	429,981,696	7,473,433	384

Table 2 presents the results from the computational group theory routines discussed in Section 4. The important observation to make is that the size of the structures remains relatively small even as the state space increases to many hundreds of millions of states. The routines needed to compute the automorphism group have theoretic exponential running times, but for our class of problems $|V| + |E|$ is rather small, and we frequently observed running times of less than one second. Note that even if the model composition graph contains a thousand nodes, computations are still feasible, and the execution times of those routines are still negligible compared to the total running time. In addition, we can see that the number of nodes in a Schreier Tree is also small. That number dominates the running time of the $rep(s)$ routine, which in turn is key to the on-the-fly mapping of indices in a matrix-vector multiplication. So, even when the state space of a model becomes large, the structures required to perform the lumping remain relatively small, leaving most of the memory available for iteration vectors.

Table 2. Symmetry detection results

#	# Aut. in SGS	# Nodes in ST	# Nodes in MDD	Final (KB)	Peak (KB)
1	1	2	30	2.3	4.6
2	2	6	67	5.1	14.2
3	5	24	81	6.3	21.4
4	8	42	95	7.5	30.4
5	3	12	140	10.6	37.5
6	4	12	146	11.2	43.0
7	6	20	152	11.7	48.1
8	7	44	170	13.2	57.4

Given that the structures used to store the permutation information are small, the remaining important factor is the size of the MDD and MxD structures used to represent the lumped state space $\tilde{\mathcal{S}}$ and the rate matrix $\mathbf{R}$. Note that the MDD of $\tilde{\mathcal{S}}$ and the on-the-fly computation of $rep(s)$ are used to mimic the effect of matrices $\mathbf{W}$ and $\mathbf{V}$ such that those matrices do not need any other data structures.

Tables 2 and 3 summarize the memory needed for the MDDs and MxDs to perform a matrix-vector multiplication as described in Section 4.3. We observe an effect that is commonly reported for symbolic state-space exploration. During the beginning of the iterative SSG process, the unlumped MDD grows in size as new states are found and reaches a peak. Then, the size of the unlumped MDD begins to shrink as the state space becomes more structured, which induces more sharing between nodes. Corresponding space requirements in Kilobytes are given in Table 2 for final (column 5) and peak (column 6) memory consumption for the MDD of $\mathcal{S}$.

The performance cost of using symbolic techniques to represent $\mathbf{R}$ and $\tilde{\mathcal{S}}$ is an increase in execution time. We have a complete implementation of all of the discussed al-

Table 3. Memory costs of lumped CTMC.

#	Lumped SS (MDD)			Lumped MxD		
	# Nodes	Final (KB)	Peak (KB)	# Nodes	Peak/Final (KB)	Time (sec) / Iter.
1	72	5.5	5.5	180	5.6	0.015
2	256	19.6	20.2	890	27.6	1.03
3	691	54.4	70.8	2579	81.9	21.7
4	1279	102.2	99.5	5070	159.2	172.1
5	2628	202.0	479.7	10565	328.3	109.0
6	783	59.8	65.8	3599	111.3	727.9
7	949	73.4	78.5	4132	127.9	3101
8	9116	720.3	999.9	39060	1221.5	2235

gorithms in the Möbius modeling tool using the model-level and state-level AFIs. The model-level AFI enables us to analyze Möbius models, whereas the state-level AFI enables us to use numerical solvers to compute reward values for transient and steady-state distributions of such models. See Table 3 column 7 for a summary of the times to execute a single iteration of a numerical solver.

A more recent study is that of a network of LEO satellites [9], where both dependability and performance were studied. In that work, an analytic solution was not possible due to the state space explosion problem, so simulation was used. With our technique, however, we were able to solve the dependability metrics analytically using symmetry detection and symbolic data structures. To accomodate our techniques, the model in [9] was refactored to expose structural symmetry, but no individual submodel was changed. With this modification, our techniques were able to solve the dependability related metrics, even though the unlumped state space has nearly 234 million states. The resulting lumped CTMC has about 20 million states, uses 817 KB of main memory, and has a time/iteration of about 3100 seconds per iteration.

The third example is a well-studied fault-tolerant parallel computer system [5], where a number of different models can be studied by varying the number of computers in the system. This model is formed using rep-join composition, which is a subset of graph composition. Although this model is not perfectly suited for these techniques, it serves to demonstrate the slowdown of the symmetry detection routines with respect to the state of the art rep-join symbolic techniques. We tested the techniques with 1,2, and 3 computers, which had unlumped state spaces of 414; 256,932; and 124,075,800 states, respectively. The symmetry detection routines found lumped state spaces of size 116; 10,114; and 463,268, respectively. Due to the added complexity of the symmetry detection routines, the time to execute a single iteration of a solution technique is slower than that of the results presented in [5]. In this model, we observe slowdowns of 1.3 to 5 times for different variants.

6 Conclusion

This paper proposes a method to combine state-space reduction using lumpability from symmetries in the model composition graph of state-sharing composed models with symbolic data structures to represent the generator matrices of extremely large CTMCs. Both approaches are complementary and allow us to achieve impressive synergy effects for the transient and steady-state solution of CTMCs with millions of states. The overall technique is implemented and integrated within Möbius and will be made available in the near future.

References

[1] P. Buchholz. Exact and ordinary lumpability in finite Markov chains. *J'rnl of Applied Probability*, 1994.

[2] G. Butler. *Fundamental Algorithms for Permutation Groups*. LNCS 559. Springer, 1991.

[3] G. Ciardo and A. S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In *PNPM*, 1999.

[4] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry generation for CNF. In *Proc. of the 41st Design Automation Conference*, 2004.

[5] S. Derisavi, P. Kemper, and W. H. Sanders. Symbolic state-space exploration and numerical analysis of state-sharing composed models. *LAA*, 2004.

[6] S. Derisavi, P. Kemper, and W. H. Sanders. Lumping matrix diagram representations of Markov models. In *Dependable Systems and Networks*, pages 742–751, 2005.

[7] S. Derisavi, P. Kemper, W. H. Sanders, and T. Courtney. The Möbius state-level abstract functional interface. *Performance Evaluation*, 54(2):105–128, 2003.

[8] D. D. D. et al. The Möbius framework and its implementation. *IEEE Trans. Softw. Eng.*, 2002.

[9] E. A. et al. Evaluating the dependability of a LEO satellite network for scientific applications. In *QEST '05*, Washington, DC, USA, 2005. IEEE Computer Society.

[10] G. C. et al. Saturation: An efficient iteration strategy for symbolic state-space generation. *LNCS*, 2001.

[11] P. B. et al. Kronecker based matrix representations for large Markov models. In *Validation of Stochastic Systems*, 2004.

[12] R. Mehmood. Serial disk-based analysis of large stochastic models. In *Validation of Stochastic Systems*, pages 230–255, 2004.

[13] A. S. Miner and D. Parker. Symbolic representations and analysis of large probabilistic systems. In *Validation of Stochastic Systems*, pages 296–338, 2004.

[14] V. F. Nicola. Lumping in Markov reward processes. Technical report, IBM Res. Rep., 1989.

[15] W. D. Obal II, M. G. McQuinn, and W. H. Sanders. Detecting and exploiting symmetry in discrete-state Markov models. *Pacific Rim Dependable Computing*, pages 26–38, 2006.

[16] W. H. Sanders and J. F. Meyer. Reduced base model construction methods for stochastic activity networks. *IEEE J'rnl on Selected Areas in Communications*, 1991.

[17] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton, 1994.