Detecting and Exploiting Symmetry in Discrete-state Markov Models

W. Douglas Obal II, Michael G. McQuinn, *Member, IEEE,* and William H. Sanders, *Fellow, IEEE*

***Abstract*— Dependable systems are usually designed with multiple instances of components or logical processes, and often possess symmetries that may be exploited in model-based evaluation. The problem of how best to exploit symmetry in models has received much attention from the modeling community, but no solution has garnered widespread support, primarily because each solution is limited in terms of either the types of symmetry that can be exploited or the difficulty of translating from the system description to the model formalism. We propose a new method for detecting and exploiting model symmetry in which 1) models retain the structure of the system, and 2) all symmetry inherent in the structure of the model can be detected and exploited for the purposes of state-space reduction. Composed models are constructed from models through specification of connections between models that correspond to shared state fragments. The composed model is interpreted as an undirected graph, and results from group and graph theory are used to develop procedures for automatically detecting and exploiting all symmetries in the composed model. We discuss the necessary algorithms to detect and exploit model symmetry and provide a proof that the theory generates an equivalent model. After a thorough analysis of the added complexity, a state-space generator which implements these algorithms within Möbius [1] is then presented.**

***Index Terms*— Markov Processes, Group Theory, Graph Theory, Modeling, State space methods**

NOTATION

S	Set of state variables
E	Set of events
ϵ	Event enabling function
λ	Transition rate function
τ	State transition function
Σ	Set of models
I	Set of instances of models in Σ
κ	Instance type function
C	Set of connections for a composed model
Ξ	Partition of vertices
μ_A	Projection of composed model state μ onto model instance A
μ^γ	Action of automorphism γ on composed model state μ
$[\mu^\gamma]_A$	Projection of μ^γ onto model instance A
Δ_μ	Set of states reachable from composed model state μ
Γ	Automorphism group of the graph
$< S >$	Generating set for a group

I. INTRODUCTION

WE consider state-based modeling methods for evaluating dependable systems. Systems with dynamic structure and/or state-dependent component failures cannot be handled by combinatorial techniques like fault trees. A state-based modeling method is required to model systems with such characteristics. State-based Markov models have been successful in determining the reliability of many different systems [2]. More recently, they have been used to test multichip systems [3], study operation-time managed systems with critical faults [4], and study the interaction between hardware and software failures [5]. The first step in applying a state-based method is usually the generation of the state space. Unfortunately, large models usually produce very large state spaces, which are problematic for numerical evaluation techniques.

The question of how to cope with large state spaces has received much attention from the modeling community over the last two decades. The various solutions that have been invented fall into two categories. Each approach either seeks to tolerate large state spaces, or seeks to reduce large state spaces to smaller ones. Research on the problem of tolerating very large state spaces has focused on efficient algorithms and data structures that seek to maximize the number of states that can be represented and the speed with which they may be manipulated within the memory hierarchy of a workstation. Examples of this approach include the Kronecker product [6] algorithms of Plateau [7], [8], Buchholz [9], [10], Donatelli [11], and Kemper [12], and the methods of Deavours and Sanders [13], [14], Ciardo and Miner [15], and Derisavi [16].

Research on methods for reducing large state spaces has focused on methods for exploiting system characteristics to reduce the number of states that must be considered. Methods for partial exploration of the state-space with error bounds for some measures are discussed in [17] and [18]. Other methods for state-space reduction take advantage of the structure of the model. Examples of this approach are the work of Aupperle and Meyer [19], [20], the hierarchical modeling techniques of Buchholz [21], stochastic well-formed nets [22], performance evaluation process algebra [23], the reduced base model construction methods of Sanders and Meyer [24], and Somani's

This material is based upon work supported by DARPA/ITO under Contract No. DABT63-96-C-0069, by the National Science Foundation under Grant No. CCR-0311616, and by Pioneer Hi-Bred International, Inc. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA/ITO, the National Science Foundation, or Pioneer.

This work was done while Doug Obal was a Research Associate at the Coordinated Science Laboratory at the University of Illinois.

M. McQuinn and W. Sanders are with the Department of ECE, The University of Illinois at Urbana-Champaign, Urbana, IL, 61801 USA (email mmcquinn@crhc.uiuc.edu and whs@uiuc.edu)

symmetry exploitation algorithms [25]. These approaches all use symmetry to reduce the state space by lumping states that correspond to symmetric configurations. An alternative to these exact methods is the decomposition method of Ciardo and Trivedi [26], which treats nearly independent submodels as independent and uses fixed-point iteration to solve the model.

Each symmetry exploitation technique has its advantages and disadvantages. Some techniques are easy to use but limited in the types of symmetry that can be detected. For example, [20] is limited to multiprocessor systems with permanent faults. Other techniques do well at detecting symmetry, but the specification language leads to models that are difficult to read. Ideally, we would like to have a modeling technique that produces models that reflect the structure of the system we are modeling, but is still able to detect and exploit all symmetry in the model. In order to reflect the structure of the system, the method needs to be compositional, with explicit relationships between submodels. To exploit symmetry, the model specification must either directly indicate the symmetry that is present or be amenable to an analysis that detects the symmetry.

In this paper we present a new technique for detecting and exploiting symmetry in discrete-state Markov models. The technique relies on a composition graph that specifies interaction between submodels, and automatically detects all symmetry present in the graph structure. We present the technique in the general context of discrete-state Markov models, since our work is not specific to any existing modeling formalism like the work of Aupperle and Meyer [20]. The theory underlying our approach uses results from group and graph theory, but presents results that are more general than that work. With our technique, models retain the structure of the system, and all symmetry inherent in the model structure is detected and exploited to reduce the state space.

II. Model Description

The model specification language is meant to simplify the exposition by providing the minimum notation needed to discuss the composition of models and the construction of the underlying stochastic process. Many different formalisms for describing discrete event systems can be mapped onto this basic notation, but the ideas presented here are useful regardless of the details of the specification.

Definition 1: A *model* is a five-tuple $(S, E, \varepsilon, \lambda, \tau)$ where

- S is a set of state variables $\{s_1, s_2, \ldots, s_n\}$ that take values in $I\!N$. The state of the model is defined as a mapping $\mu : S \to I\!N$, where for all $s \in S$, $\mu(s)$ is the value of state variable s. Let $M = \{\mu \mid \mu : S \to I\!N\}$ be the set of all such mappings.
- E is the set of events that may occur.
- $\varepsilon : E \times M \to \{0, 1\}$ is the event enabling function. For each $e \in E$ and $\mu \in M$, $\varepsilon(e, \mu) = 1$ if event e may occur when the current state of the model is μ, and zero otherwise.
- $\lambda : E \times M \to (0, \infty)$ is the transition rate function. For each event e and state μ such that $\varepsilon(e, \mu) = 1$, event e occurs with rate $\lambda(e, \mu)$ while in state μ.
- $\tau : E \times M \to M$ is the state transition function. For each $e \in E$ and $\mu \in M$, $\tau(e, \mu) = \mu'$, the new state of the model that is reached when e occurs in μ.

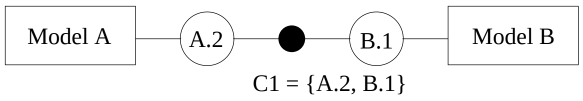

Fig. 1. Models are connected through shared state variables

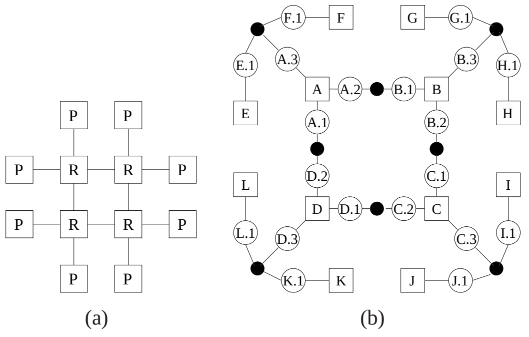

Fig. 2. (a) Ring of dual processors system and (b) model composition graph

The *behavior of a model* is a characterization of possible sequences of events and states. Event occurrence rates are determined by λ. In Definition 1, once an event is chosen, the next state is determined by τ. Given that the current state of the model is μ, the probability of transition to a particular next state, μ', is the probability that the next event to occur is such that $\tau(e, \mu) = \mu'$. This is calculated as

$$\Pr\{\mu \to \mu'\} = \frac{\sum_{\{e \in E \mid \tau(e,\mu) = \mu'\}} \lambda(e, \mu)}{\sum_{\{e \in E \mid \varepsilon(e,\mu) = 1\}} \lambda(e, \mu)}.$$

Models are connected together through shared state variables to form “composed models.” Figure 1 shows an example in which two models are composed through the specification of the sharing of two state variables. Models A and B each have state variable sets containing two state variables $\{A.1, A.2\}$ and $\{B.1, B.2\}$. In this case, the second state variable for instance A is joined to the first state variable for instance B, forming a single composed model state variable named $C1$. In this specification, the values of state variables $A.2$ and $B.1$ will always be equal and can conveniently be labelled with $C1$. The resulting composed model state variable set is $S = \{A.1, C1, B.2\}$, where $C1$ represents the sharing of state variables $A.2$ and $B.1$. As shown in Figure 1, $C1$ is the connection representing the superposition of $A.2$ and $B.1$.

Systems composed of multiple identical subsystems exhibit symmetry. For example, consider Figure 2-a, which shows a ring of dual-processor nodes. Each of the boxes labeled “R” represents a network node, and each box labeled “P” represents a processor. Figure 2 serves well to demonstrate symmetry ideas that will be more completely exploited in Section VI. For the system shown, we make two models, one for the network node and one for the processor, and then build the composed model from “instances” of these models. Before showing how this is done, we give the formal definition of a composed model.

Definition 2: A *composed model* is a four-tuple (Σ, I, κ, C) where

- Σ is a set of models.
- I is a set of instances of models in Σ. Each instance is a complete copy of a model in Σ, and is independent of all other instances, except as explicitly defined through the connection set.
- $\kappa : I \to \Sigma$ is the instance type function.
- C is a set of connections. Each connection $c \in C$ represents a state variable shared among two or more instances. In this way, c represents the superposition of one state variable from each connected instance.

A composed model thus partitions the state variable set of each instance into two subsets: those variables that are shared with other instances, and those that are not. Subsets of a state variable set will be called *state variable fragments*. The subset of an instance state variable set that is not shared will be called the *private state variable fragment* of that instance. Each state variable that is not in the private state variable fragment is shared, and appears as an element in exactly one connection set.

The tuple notation in Definition 2 is useful for formal definition, but the graphical representation illustrated in Figure 1 is better suited for visualization of the structure of a composed model. The following conventions for drawing composed models will be adopted. The private state variable fragment for an instance is depicted by a box with the instance identifier, while shared state variable fragments are represented by circles labeled with a fragment identifier comprising the instance name and state variable identifier. Connection nodes are represented by small solid circles.

We call the graphical representation a "model composition graph." A *model composition graph* is an undirected graph, $G = (V, W)$. Elements of the vertex set, V, are private state variable fragments, shared variables, or connection nodes. Every instance has exactly one private state variable fragment, but this fragment may be empty. It is possible that all state variables in an instance state variable set are shared. In that case, the empty private state fragment serves as an anchor for the shared variables, as will be understood from the requirements on the edge set. There are two rules that must be satisfied by the edge set, W, of the graph. First, every shared state variable for an instance must be adjacent to the private fragment for that instance. Second, each shared variable is adjacent to exactly one connection node.

To explain this approach to modeling, we use the example of Figure 2. The first step in using our approach is to model the two components used in this system. We give detailed examples of component models in Section VI; our main focus here is the composition of models. Given models of a processor and a network node, the question is how to compose them to form the system model. Figure 2-b shows a model composition graph for the system. Instances A, B, C and D are instances of the network node model, while instances E–L are instances of the processor model. In drawing this model composition graph, we have assumed the ring has direction and the processors share an interface. Thus, network node instance A has three shared state fragments. $A.1$ is A's incoming link, $A.2$ is its outgoing link, and $A.3$ is its processor interface. To form the ring, A connects its outgoing link to $B.1$, the incoming link of instance B. Meanwhile, processor instance E has a network interface, $E.1$, which is connected to $A.3$, as is $F.1$, the network interface of processor instance F. This connection indicates symmetric access to the network node. From Figure 2, one can see the symmetry that can be exploited. There is a rotational symmetry around the center of the ring, and the processors at each node are symmetric about the interface.

We can now describe the composed model in terms of the models it comprises. The composed model state variable set may be derived from the vertex set of the model composition graph through deletion of vertices corresponding to shared state variables. The resulting composed model state variable set contains the state variables in the private state variable fragment of each instance, and a state variable for each vertex corresponding to a connection. As it was with models, the *composed model state* is defined as a mapping $\mu : S \to I\!N$ from the composed model state variable set (S) to the nonnegative integers. M again represents the set of all possible states.

The composed model event set is simply the union of the event sets for all instances. The subset of the composed model event set that originates in an instance A is denoted E_A. The event enabling function for the composed model is also a simple union of the functions for each instance. That is, given a composed model state, μ, and some event, e, the composed model event enabling function is $\varepsilon(e, \mu) = \varepsilon_A(e, \mu_A)$, when $e \in E_A$. Likewise, the composed model event rate function is $\lambda(e, \mu) = \lambda_A(e, \mu_A)$ when $e \in E_A$.

The interaction between instances in the composed model is captured in the "composed model transition function," whose definition utilizes the notion of the "local state" of an instance. The *local state of an instance* is the projection of the composed model state onto the state variables of the instance. Note that the private state variable fragment of an instance is represented explicitly in the composed model state. The shared state variables of an instance are assigned the values held by the associated connections in the composed model state. Given a composed model state μ, the local state of instance A will be denoted μ_A.

Definition 3: The *composed model state transition function* is defined as $\tau : E \times M \to M$, where E and M are the set of events and the set of all possible states for the composed model. Let τ_A denote the state transition function for the instance A. Then, for all $e \in E_A$ and composed model states μ,

$$\tau(e, \mu) = (\mu - \mu_A) \cup \tau_A(e, \mu_A).$$

With the composed model functions defined, writing a procedure that will generate the state space for a composed model is straightforward, as shown in Figure 3. However, the detailed state space will be very large for most composed models. Fortunately, in many cases the detailed state space contains much more information than is needed to evaluate the dependability measure of interest. In such cases, the specific identity of a model is not required. Sometimes all that is needed is the quantity of each type of model in each

```
$U$ : Unexplored states, $S$ : Discovered states
$E$ : Event set, $E_i$ : Event set of instance $i \in I$
Initial state $\mu_0$, $U = \{\mu_0\}$, $S = \{\mu_0\}$
while $U \neq \emptyset$
    choose a $\mu \in U$ and let $U = U - \{\mu\}$
    $E(\mu) = \{e \in \bigcup_{i \in I} E_i \mid \varepsilon(e, \mu) = 1\}$
    for each $e \in E(\mu)$
        $\mu' = \tau(e, \mu)$
        if $\mu' \notin S$
            $S = S \cup \mu'$
            $U = U \cup \mu'$
        add arc from $\mu$ to $\mu'$ with rate $\lambda(e, \mu)$
    end for
end while
```

Fig. 3. Procedure for detailed state generation from a composed model

state possible for that type of model. In other cases, it is not enough to know the numbers; one also needs to know something about the configuration of the various model states. An example of such a system is the BIBD network proposed by Aupperle and Meyer [19]. In either case, and in other situations where the precise identity of each component in a redundant set is not required, there are symmetries that may be exploited. Detecting such symmetries in the model is the topic of the following section.

III. Detecting Symmetry

In the composed model formalism of Section II, the structure of the model is exposed in the model composition graph. We now describe a new method for detecting symmetry using the model composition graph. In this new approach, we use the automorphism group of the model composition graph to detect structural symmetry. The main result of this section is a proof that we can construct a Markov process with states that are the partition of the state space induced by the automorphism group of the model composition graph. Readers not interested in the derivation are encouraged to skip to Section IV where an example model illustrates the procedure.

A structural symmetry is present whenever there are multiple instances of the same model present in the composed model, and the names of these instances can be permuted in some way so that the model is structurally indistinguishable from the original. A behavioral symmetry is present if the permutation of instance names can be done without changing the behavior of the model. The main tool for detecting structural symmetry in the model composition graph is the automorphism group of the graph. An automorphism of a graph permutes the names of the vertices in such a way as to maintain the structure of the graph, thus exposing a structural symmetry. We now turn to the technical details. Then we will show that the structural symmetry induces a behavioral symmetry, which can be exploited to obtain a smaller Markov process representation.

The assumption of a homogeneous vertex set is implicit in the standard definition of graph automorphism, so that any two vertices with the same degree may be mapped onto one another. However, the model composition graph may include vertices representing instances of different models, so it will not have a homogeneous vertex set. In this case, automorphisms must be restricted to permutations that map vertices representing state variables of each model type only among themselves.

Let $\Xi = \{\xi_1, \xi_2, \ldots, \xi_n\}$, be a partition of V that satisfies the following requirements:

1) Two vertices are in the same partition element if and only if the vertices correspond to the same state variable fragment of instances of the same model.
2) All vertices corresponding to connection nodes are in the same partition element.

Let Γ be the automorphism group of the graph with respect to Ξ. By this it is meant that Γ is a permutation group on the vertex set of the composition graph, such that for all $\gamma \in \Gamma$, $v \in \xi_i$ if and only if $\gamma(v) \in \xi_i$.

Permutations in Γ map V onto itself. For convenience, the permutation notation $\gamma(\cdot)$ will be overloaded so it can be used with an argument that is either a state variable fragment or a state variable. This overloading is justified by the fact that elements of Γ are restricted by definition to map vertices within their own partition elements, which means that for all $\gamma \in \Gamma$, $\gamma(v)$ is a vertex with the same structure as v. Therefore, $\gamma(s)$ will be used to denote the state variable in the fragment $\gamma(v)$ that is the image of s under γ. An example should help clarify this notion. Suppose $v_1 = \{A.1, A.2\}$ is the private state variable fragment of instance A, and $v_2 = \{B.1, B.2\}$ is the private state variable fragment of instance B. If $\gamma(v_2) = v_1$, then by definition $\gamma(B.1) = A.1$ and $\gamma(B.2) = A.2$. The next step is to demonstrate that such structural symmetries induce behavioral symmetry, and to characterize the nature of the behavioral symmetry.

To demonstrate the behavioral symmetry among symmetrical structural configurations of a composed model, we must investigate the effect of an automorphism on the composed model state. An automorphism is a renaming of instances, and a composed model state is a mapping of instance state variables to numbers. One way to visualize the effect is to imagine the model composition graph with each vertex additionally labeled with the projected composed model state. Now imagine that the vertex names are shuffled according to an automorphism, while the projected composed model states remain in place. Formally, for a given composed model state, μ, and an automorphism, $\gamma \in \Gamma$, the action of γ on μ is defined as

$$\mu^\gamma = \mu \circ \gamma, \tag{1}$$

where $\circ$ denotes composition of functions. For every state variable, s, in the composed model, $\mu^\gamma(s) = \mu(\gamma(s))$.

For the simple example above, Equation 1 indicates that $\mu^\gamma(B.1) = \mu(A.1)$. Having given the mathematical definition, the next step is to consider what it means in terms of the model behavior.

An automorphism of the model composition graph maps instance states among themselves. If, as in the above example, the action of γ maps the state of an instance A onto the instance B, this will be denoted by $B^\gamma = A$. In turn, the new state of instance A is A^γ. Determining the model behavior in the permuted composed model state requires knowledge of the set of events that are possible in the new state. Suppose $e \in E_A$ and $B^\gamma = A$. Then, by the definition of Γ, which ensures $B^\gamma = A$ if and only if A and B are instances of the same model, there must be an event, e', in B that corresponds to e in A. Since e' is to e what μ^γ is to μ, it is natural to call e' *the action of* γ *on* e and use the notation e^γ.

We will now show how Γ detects symmetry in the model. The first step is to show how Γ induces a partition of M, the set of all mappings of composed model state variables to numbers.

Definition 4: 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 $\mu_2 = \mu_1^\gamma$.

Proposition 1: L is an equivalence relation.

Proof: See [27]. ■

By Proposition 1, the automorphism group of the model composition graph partitions the state space of the composed model into equivalence classes defined by L. It will now be shown that elements in the equivalence classes of L are symmetric in the sense that all elements in a class of L have future behavior that is statistically indistinguishable. The main step in the proof is to demonstrate that for two composed model states in the same class of L, the sets of next possible states are equivalent under L.

As with the composed model state, it will be necessary to refer to projections of permuted composed model states onto instance states. The projection of μ^γ onto some instance, A, is denoted by $[\mu^\gamma]_A$. It is also useful to define precisely the idea of the action of an automorphism on the local state of an instance. Given the projection of a composed model state, μ, onto the local state of an instance, A, the action of an automorphism, γ, on μ_A must be defined. Note that this cannot be a straightforward composition of functions, since the codomain of γ is not the same as the domain of μ_A. On the other hand, it is easy to define what is meant.

Definition 5: Let the domain of μ_A, $A \subseteq S$, be denoted $\mathcal{D}(\mu_A)$. The action of γ on μ_A is defined as

$$[\mu_A]^\gamma = \{(s, \mu_A(\gamma(s))) \mid \gamma(s) \in \mathcal{D}(\mu_A)\}.$$

The relationship between the projection of Definition 5 and the action of an automorphism can now be explored. Suppose one delineates the local state of an instance, A, and follows it as it is moved by an automorphism to another instance, B. Now suppose one first applies the same automorphism and then examines the local state of B. In each case one sees the same local state. The formal statement is Proposition 2.

Proposition 2: For all instance pairs (A, B) and for all $\gamma \in \Gamma$ such that $B^\gamma = A$,

$$[\mu_A]^\gamma = [\mu^\gamma]_B.$$

Proof: Automorphisms are one-to-one and onto, so for any instance A and automorphism γ, there exists some other instance B such that $B^\gamma = A$. The set $\{s \mid \gamma(s) \in \mathcal{D}(\mu_A)\}$ is exactly the subset of composed model state variables that is projected onto the set of state variables of instance B to obtain the local state of instance B in a given composed model state. Therefore, $\mathcal{D}([\mu^\gamma]_B) = \mathcal{D}([\mu_A]^\gamma)$. This means that $[\mu_A]^\gamma$ assigns the values of the local state variables of A in composed model state μ to the corresponding local state variables of B, since $\gamma(\cdot)$ must be the same type of state variable by definition of Γ. Finally, the result follows from the definition of μ^γ.

■

Now that the effect of an automorphism on a composed model state has been established, the next point to consider is the state transition function in the new state, and how it relates to the state transition function of the original state. This point is the key to behavioral symmetry, since the state transition function defines what can happen next. After the relationship between state transition functions of states related by automorphism has been established, the behavioral symmetry can be characterized.

Proposition 2 is the main step in proving that states within an equivalence class of L have the same behavior. The next proposition gives the relationship between the transition functions for two states related by automorphism of the model composition graph. Informally, the proposition says that for any two states in an equivalence class of L, their sets of next possible states are related by the same automorphism as the two states themselves.

Proposition 3: For all $\mu \in M$, $e \in E$, and $\gamma \in \Gamma$,

$$\tau(e, \mu)^\gamma = \tau(e^\gamma, \mu^\gamma).$$

Proof: Let e be an event from an arbitrary instance A. Then, for every $\gamma \in \Gamma$ there exists an instance B such that $B^\gamma = A$. First, the automorphism γ is applied to the definition of the state transition function (Definition 3) to get:

$$\tau(e, \mu)^\gamma = [(\mu - \mu_A) \cup \tau_A(e, \mu_A)]^\gamma.$$

Since $(\mu - \mu_A)$ and $\tau_A(e, \mu_A)$ are disjoint sets, the action of γ from Definition 5 can be applied to result in

$$\tau(e, \mu)^\gamma = [\mu_{S-A}]^\gamma \cup [\tau_A(e, \mu_A)]^\gamma.$$

At this point, recalling that $B^\gamma = A$, it follows from Proposition 2 that

$$\tau(e, \mu)^\gamma = [\mu^\gamma]_{S-B} \cup \tau_B(e^\gamma, [\mu^\gamma]_B). \tag{2}$$

Finally, after rewriting (2) as

$$(\mu^\gamma - [\mu^\gamma]_B) \cup \tau_B(e^\gamma, [\mu^\gamma]_B),$$

and applying Definition 3, the result follows from the fact that e and A are arbitrary. ■

The set of states that may be reached from a composed model state, μ, is

$$\Delta_\mu = \bigcup_{\{e \in E \mid \varepsilon(e,\mu)=1\}} \tau(e, \mu).$$

Each state in Δ_μ is also an element of some equivalence class with respect to L. Let H be an equivalence class with respect to L and suppose $H \cap \Delta_\mu \neq \emptyset$. In this case, H will be called a *destination class of* μ. Furthermore, when H is a destination class of μ, the set of events $\{e \in E \mid \tau(e, \mu) \in$

$H\}$ will be denoted by $E_{\mu,H}$. With these definitions, we can precisely define the notion of equivalent behavior.

Proposition 4: For all pairs of composed model states μ_1 and μ_2, if $\mu_1 L \mu_2$ then μ_1 and μ_2 have the same set of destination classes and the same transition rates to those classes.

Proof: $\mu_1 L \mu_2$ implies there exists γ such that $\mu_2 = \mu_1^\gamma$. Therefore, the transition functions $\tau(\cdot, \mu_1)$ and $\tau(\cdot, \mu_2)$ lead to the same destination classes by Proposition 3. By the definition of Γ, there is a one-to-one correspondence, $e \leftrightarrow e^\gamma$, between the set of events that may occur in μ_1 and the set of events that may occur in $\mu_2 = \mu_1^\gamma$. Therefore, for each destination class H, $|E_{\mu_1,H}| = |E_{\mu_2,H}|$, and the total transition rate from μ_1 to H is equal to that from μ_2 to H. ■

Proposition 4 establishes a localized notion of equivalent behavior. If two states are related by an automorphism of the model composition graph, then the things that can happen next in both states are equivalent, in the sense that each state that can be reached from one state is related by automorphism to a state that can be reached from the other state. So Proposition 4 establishes equivalent behavior for one step into the future. To prove that the entire future behaviors of the two states are also symmetric, we use a well-known result from the theory of Markov chains, commonly known as the *Strong Lumping Theorem*.

Proposition 5: The model created by replacing each equivalence class of L with a single representative state satisfies the Markov property.

Proof: Follows directly from Proposition 4 and the Strong Lumping Theorem. ■

In this section we have shown how the automorphism group of the graph may be used to detect symmetry in a model. The next section discusses the practical issues involved in exploiting the detected symmetry for the purposes of reducing the state space.

IV. Exploiting Symmetry

As shown in Section III, the automorphism group of the model composition graph may be used to detect symmetries, which can in turn be used to reduce the state space of the model. This section considers the practical issues of how to compute the automorphism group and how to use that information to directly generate a reduced state space for the composed model. The main result is a procedure for generating a compact state space for composed models.

Also, we provide an example model to illustrate the procedure. Consider a model M containing the three state variables $priv$, $link1$, and $link2$. Each state variable is binary and represents whether a portion of the system is functioning. If $priv$ is 0 (i.e., not working), then $link1$ and $link2$ must also be 0. So, M has the following five states $\{priv, link1, link2\} = \{(1,1,1),(1,0,1),(1,1,0),(1,0,0),(0,0,0)\}$. Three instances of M are joined to form a composed model represented in Figure 5-a. Two instances of model M are connected by sharing the link state variables. For example, the state variable $link1$ of M_1 and the state variable $link2$ of M_2 form a single shared state variable.

A. Generating the Model Composition Graph

The first step in generating a compact state space is to derive the model composition graph. For each model m in the set of instance models I of a composed model, we must determine its possibly empty private state variable fragment and its public state variable fragments. This procedure must be done over all instances of models in I and not just over the set of models Σ since the connection set C may be different for different instances of the model. This may happen when a modeler creates a submodel for a component in a large system, but uses the submodel in slightly different ways. For example, in Figure 10, the router model is used differently depending on where it is used in the composed model. If it is used in the inner ring, both connection links are shared, whereas if it is used in the outer ring, only one connection link is shared.

Figure 4 presents an algorithm to find the model composition graph of a composed model. For each instance of each model, it first finds its private state variable fragment and then proceeds to find all public state variable fragments using the connection information. The methods CreateVertex and AddEdge can easily be implemented using many different data structures and are not presented here. Also, special care must be taken to ensure that only one connection node is created for each set of shared variables. The method ConnectionInGraph returns the connection node associated with the shared variable if it exists, otherwise it returns NULL. This method can be implemented with a map or hash table or other similar structure.

The last step in generating the model composition graph is to partition the vertices into different elements. Once again, special care must be taken since models can be used differently in different instances of the composed model. So, for example, not all private state variable fragments of a model type are necessarily in the same partition element. If two models of a specific type have different sharing behavior, the private state variable fragments will be in different partition elements. Three rules can be used to find a suitable partition. Two vertices are in the same partition element if:

- they both correspond to private state variable fragments of the same model and contain the same private state variables, **OR**
- they both correspond to public state variable fragments of the same model and contain the same shared state variable, **OR**
- they are both connection nodes

The model composition graph for the example model is represented in Figure 5-b. Here, the state variables $link1$ and $link2$ are represented by the numbers 1 and 2 to save space, and the state variable $priv$ is repesented by the private state variable fragment. The composed model is formed by joining the state variables $link1$ and $link2$ in a cycle formation.

B. Finding the Automorphism Group

Once the model composition graph of a composed model is found, the next step is to compute its automorphism group. For

```
for each $m \in I$
   let $V_p$ be the non-shared state variables in $m$
   CREATEVERTEX($V_p$, private)
   for each $c \in C$
      let $V_s$ be the shared variable of $c$ in $m$
      CREATEVERTEX($V_s$, public)
      ADDEDGE($\{V_p$, private$\}$,$\{V_s$, public$\}$)
      let node = CONNECTIONINGRAPH($\{V_s$, public$\}$)
      if node == NULL
         node = CREATEVERTEX($\emptyset$, connection node)
      ADDEDGE(node, $\{V_s$, public)
   end for
end for
PARTITIONVERTICES()
```

Fig. 4. Procedure to find the model composition graph of a composed model

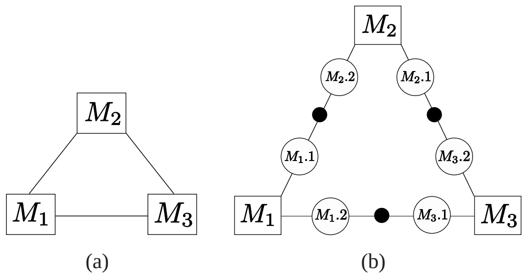

Fig. 5. (a) Example model and (b) model composition graph

models rich in symmetry, the order of the automorphism group can be very large, but a group can be compactly described by a few of its elements, called a *generating set*. A *generating set* of a group Γ is a subset of Γ, which when expanded to the smallest possible set that satisfies the properties of a group, becomes Γ. If a set S generates a group Γ, this is denoted by $< S > = \Gamma$. The next problem is how to find a generating set for the automorphism group of the model composition graph.

Efficient algorithms for computing a generating set for the automorphism group of a graph have been developed in the literature on computational group theory [28]. A brute-force implementation could search all possible labellings and determine which were automorphisms, but this would be computationally infeasible. The software package *nauty* efficiently reduces the search space by using already found automorphisms to prune the search tree of possible labellings [29], [30]. The software package *saucy* improves on the algorithm by using a sparse representation to minimize the memory used to solve the problem [31]. After we create the model composition graph, we output the results to *saucy*, which reads a simple graph description language and a vertex partition and produces a generating set for the automorphism group of the graph.

The example composed model has three automorphisms representing the three possible rotations of the ring of three instances of the model M. The two non-indentity automorphisms are:

$(M_1, M_2)(M_2, M_3)(M_3, M_1)$ and
$(M_1, M_3)(M_2, M_1)(M_3, M_2)$,

where the pair (x, y) represents moving the node x in the model composition graph to the node y. Also, the shared state variables and connection nodes must also be permuted accordingly, but these permutations are ommitted for clarity.

C. Finding a Canonical Label of a Vertex

Given that the automorphism group is known, the next problem is how to exploit this knowledge to reduce the state space. Since detailed state spaces are very large, it is important to find a method for the direct generation of the reduced state space. The reduced state space contains a single state for each equivalence class, so a procedure for choosing a representative state to serve as a canonical label for each equivalence class is needed. The standard method of choosing a canonical representative of a set is to order the set and choose the minimum or maximum of the ordered elements. The model composition graph places constraints on the sorting operation, however. For equivalence classes of composed model states, the only permutations that may be applied to order the set are those in Γ, the automorphism group. In this case, transposing two elements means that other elements may have to shift as well in order to reach a state that is still within the equivalence class. So the problem is that once a state fragment has been moved to the vertex where it belongs in the canonical label, further moves must fix this state fragment at that specific vertex.

We use the concept of a "stabilizer chain" from computational group theory [32] to develop a structure-sensitive sorting procedure that solves the canonical label problem. Recall that the automorphism group, Γ, is a permutation group acting on the vertices $v \in V$. The *stabilizer of* v *in* Γ is the set $\Gamma_v = \{\gamma \in \Gamma \mid \gamma(v) = v\}$. Thus Γ_v is the set of permutations in Γ that map the vertex v to itself. The set Γ_v is a *subgroup* of Γ, which means that the elements of Γ_v are a subset of the elements in Γ, and Γ_v satisfies the properties of a group.

The idea of a stabilizer is easily extended to more than one vertex. A stabilizer $\Gamma_{v_1 v_2}$ is a subgroup of Γ_{v_1} that also fixes v_2. A *stabilizer chain* is a decreasing sequence of subgroups, $\Gamma \supseteq \Gamma_{v_1} \supseteq \Gamma_{v_1 v_2} \supseteq \cdots \supseteq \Gamma_{v_1 v_2 \ldots v_k} = I$, that stabilize a growing number of vertices. As the number of stabilized vertices increases, the size of the stabilizing group shrinks. Eventually, the only stabilizing group remaining is the identity. The sequence $[v_1, v_2, \ldots, v_k]$ is called a *base* when the corresponding stabilizer chain ends in the identity. The subgroup that stabilizes the i^{th} component of the base will be denoted by $\Gamma^{(i+1)}$, with $\Gamma^{(1)} = \Gamma$. A stabilizer chain is typically described by a base and a "strong generating set." A *strong generating set* is a set S of generators for Γ that satisfies the condition $< S \cap \Gamma^{(i)} > = \Gamma^{(i)}$.

The stabilizer chain gives us exactly what we need to find a canonical label, since it identifies the subgroups of permutations in Γ that fix vertices. A composed model state may be translated to its canonical label through maximization of the state fragment at every vertex in the base of the stabilizing chain. If $\mu_1 L \mu_2$, each must each have vertices with the same state in each subgroup $\Gamma^{(i)}$ in the stabilizer chain. Therefore, in each case, the same state will be moved to the base vertex.

So, to find the canonical label of a vertex, we must find a base and strong generating set representation of the stabilizer chain and we must be able to use this representation to generate a particular automorphism.

1) Finding a Base and Strong Generating Set: The Schreier-Sims algorithm is the most efficient known deterministic algorithm for computing a base and strong generating set for a stabilizer chain of a given group [33]. Our implementation of the Schreier-Sims algorithm reads the output from the *saucy* package and produces a base and strong generating set for the stabilizer chain of the automorphism group.

While the size of the automorphism group of the example model is three, a strong generating set can be found that only contains a single automorphism. Using the Schreier-Sims algorithm, the base of the strong generating set is the node M_1, and the single automorphism is:

$(M_1, M_2)(M_2, M_3)(M_3, M_1)$

It is easily seen that all three automorphims can be generated by repeatedly applying the single automorphism. For example, by applying the automorphism twice, the second automorphim of the original group is found.

2) Generating a Particular Automorphism: A stabilizer chain, stored in the form of a base and a strong generating set, provides a compact description of the automorphism group. However, using the stabilizer chain to find a canonical label requires access to permutations in the subgroups that form the chain. The method used in our implementation is based on a list called the *factorized inverse transversal*, or a *Schreier Tree*.

A *Schreier Tree* is a tree rooted at α that represents the orbit of α, or the possible locations in the labelling that α can be permuted to using the automorphism group. Each vertex in the tree represents a possible valid labelling of the graph. An edge from node x to node y in the tree represents the automorphism that moves α from the x-th position in the label to the y-th position in the label. So, a Schreier Tree gives the sequences of permutations to move any vertex in the orbit of α to α.

Figure 6 presents a method to find the Schreier Tree of a node α using a breadth first search of the strong generating set. The method GETP-TH(p, v) returns the v-th location of permutation p and the methods $AddEdge$ and $CreateVertex$ are defined as before. Starting from an element b of the base, we apple permutations from the strong generating set until all elements in the orbit of b are found. The sequence of permutations along the edges from a vertex to the root give the permutations that move the vertex to the location of b in the labelling.

Using the Schreier-Sims algorithm, we found that the base of the strong generating set is the node M_1. M_1 has three elements in its orbit, namely M_1, M_2, and M_3. Figure 7 represents the Schreier Tree associated with the basis element M_1. For example, to move node M_2 to the location of node M_1, the permutation $(M_2, M_1)(M_3, M_2)(M_1, M_3)$ is used, which is the inverse of the single permutation in the strong generating set. The inverse is needed since we want to move a node in the orbit of M_1 to the location of M_1. Also, portions of the tree can be pruned to eliminate identity elements.

By forming a Schreier Tree for each element of the base, we can easily generate the set of permutations that move a vertex

```
Frontier = $\alpha$
NewFrontier = $\emptyset$
CREATEVERTEX($\alpha$)
while Frontier $\neq \emptyset$
  for each vertex $v \in$ Frontier
     for each permutation $p$ in Strong Generating Set
        let $v'$ = GETPERMUTATION($p$, $v$)
        if INTREE($v'$) == false
           CREATEVERTEX($v'$)
           ADDEDGE($v$, $v'$, $p$)
           NewFrontier += $v'$
     end for
  end for
  Frontier = NewFrontier
end while
```

Fig. 6. Procedure to find the Schreier Tree of a vertex α

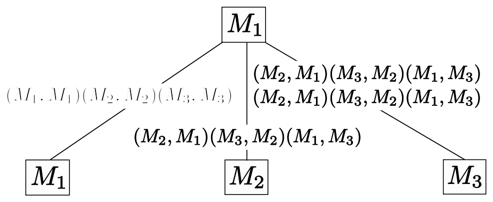
M_1
$(M_1, M_1)(M_2, M_2)(M_3, M_3)$
$(M_2, M_1)(M_3, M_2)(M_1, M_3)$
$(M_2, M_1)(M_3, M_2)(M_1, M_3)$
$(M_2, M_1)(M_3, M_2)(M_1, M_3)$
M_1
M_2
M_3

Fig. 7. Schreier-tree for base element M_1

to the position of a base element, which is exactly what is needed for the canonical labelling procedure. To find the set of permutations that move a vertex v to an element b of the base, we simply find the path in the tree rooted at b that leads to v, which can easily be done in a depth first manner.

Figure 8 shows the procedure for producing a canonical label for a composed model state. The full procedure for exploiting model symmetry to generate a reduced state space is given in Figure 9. An advantage of this approach is that the changes required to existing state space generators are minimal. All that is required is a transformation from a found state to its canonically labelled state.

Consider a composed model state of the example model where $M_1.priv = 0$, while $M_2.priv = 1$ and $M_3.priv = 1$, and all other state does not matter. This state is equivalent to two other composed model states, namely the two states where $M_2.priv = 0$ and the rest are 1 and where $M_3.priv = 0$ and the rest are 1. Using the canonical labelling procedure in Figure 8, we need to only compare the state to two other states. The two other states are defined by the Schreier Tree rooted

```
$B = [b_1, b_2, \ldots, b_n]$ : Base for stabilizer chain
$O^{(i)}$ : Orbit of $i^{th}$ base point
for each base point $b_i \in B$
  let $k$ be the index of the vertex in $O^{(i)}$ with the
      largest state (ties go to the vertex with lower index)
  apply permutation that moves $O^{(i)}(k)$ to $b_i$
end for
```

Fig. 8. Procedure for canonical labelling of a composed model state

```
$U$ : Unexplored states, $S$ : Discovered states
$E$ : Event set, $E_i$ : Event set of instance $i \in I$
Compute automorphism group
Compute stabilizer chain
Convert initial state $\mu_0$ to canonical label
$U = \{\mu_0\}$, $S = \{\mu_0\}$
while $U \neq \emptyset$
    choose a $\mu \in U$ and let $U = U - \{\mu\}$
    $E(\mu) = \{e \in \bigcup_{i \in I} E_i \mid \varepsilon(e, \mu) = 1\}$
    for each $e \in E(\mu)$
        $\mu' = \tau(e, \mu)$
        convert $\mu'$ to canonical label
        if $\mu' \notin S$
            $S = S \cup \mu'$
            $U = U \cup \mu'$
        add arc from $\mu$ to $\mu'$ with rate $\lambda(e, \mu)$
    end for
end while
```

Fig. 9. Procedure for generating compact state-space for a composed model

in M_1 defined in Figure 7. The resulting canonical state is the lexicographic maximum of the three symmetric states, or the state where $M_1.priv = 1$, $M_2.priv = 1$, and $M_3.priv = 0$.

We have built a state generator that works with models described as stochastic Petri nets. The models are described graphically using the Möbius modeling tool. Then, the model composition graph is created, and the structures for canonical labelling are found. At this point, state-space generation begins and is carried out according to the procedure listed in Figure 9.

V. Complexity Analysis

It is well known that the complexity of a state generation procedure is dominated by the number of states that must be generated. In general, the number of states in the model may be an exponential or combinatorial function of the number of components in the system. However, since the new procedure differs from the standard state generation procedure, it is useful to examine the impact of the changes. There are two significant differences. The new procedure computes the automorphism group of the model composition graph and canonically labels each new state that is found.

A. Computing the Automorphism Group

To compute the automorphism group, we must first create the model composition graph. Analyzing Figure 4, the inner loop must be executed for each connection associated with a particular node in the composed model. The number of connections is limited by the number of shared variables, so the inner loop will be executed in the worst case $O(s)$ times, where s is the number of shared variables. The inner loop is executed for each instance in the composed model, or $O(n)$ times, where n is the number of nodes in the composed model. So, the total execution time is $O(ns)$.

The computation time to find the automorphism group of the graph (or the more general isomorphism problem) is not fully understood. No polynomial time algorithm exists in the general case, but no proof that the problem is NP-Complete exists either. It is believed to be somewhere between P and NP-complete, if P ! = NP [34]. However, our experience is that computing the automorphism group of the model composition graph rarely consumes more than 1 CPU second. This time is insignicant relative to the time required to generate the states. Therefore, we assume that it is unlikely to dominate the complexity of the entire procedure except in trivial cases where the state space is very small (a few hundred states, perhaps).

Many different implementations of the Schreier-Sims algorithms exist to find a base and strong generating set of an automorphism group. The fastest implementations have running times on the order of $O(n^2 log^3(g) + tnlog(g))$, where g is the order of the group and t is the number of generators. Our simpler implementation has a slower $O(n^8 logn)$ running time. Like the previous computation, our version of the Schreier-Sims algorithm takes less than 1 CPU second, a minor contribution to the overall running time.

The last algorithm needed is to calculate the Schreier Trees for each element of the base. Analyzing Figure 6, we must apply every permutation to each node in the tree in the worst case. So, this leads to a $O(tn)$ running time, where n is the number of nodes and t is the number of generators. Since we must build a tree for each element in the base, the total time required is $O(btn)$.

B. Finding a Canonical Label

The canonical labelling procedure in Figure 8 is critical in the run time of our current implementation. This procedure is called each time a state is reached. Note that a large fraction of the states in the detailed state space are not visited by our procedure, since newly discovered states that (after canonical labelling) are already in the state tree are not put in the queue of unexplored states. However, some states will be visited multiple times since they are reachable from a (possibly large) number of other states. Therefore, the number of calls to the canonical label procedure depends on the nature of the model.

We analyze the interior of the loop first. Finding the subset of vertices in the orbit of a base vertex that all share the maximal state is linear in the length of the orbit. In the worst case this is $O(v)$, where v is the number of vertices in the model composition graph. Identifying the vertex in the set of maximal vertices that will result in the maximum state when moved to the base vertex by the permutation tabled in the factorized inverse transversal is $O(vs)$, where s is the number of state variables in the composed model. Finally, applying the permutation to create the canonical label for the state is $O(s)$. Therefore, the interior of the loop is $O(vs)$.

Since we iterate over the length of the base, b, the overall asymptotic worst-case complexity of the canonical label procedure is $O(bvs)$.

The factor dominating the overall complexity of the procedure is the number of states. In the worst case, the number of states can be an exponential function of the number of vertices. But, if we can dramatically reduce the number of states, we can dramatically reduce the running time of the total algorithm.

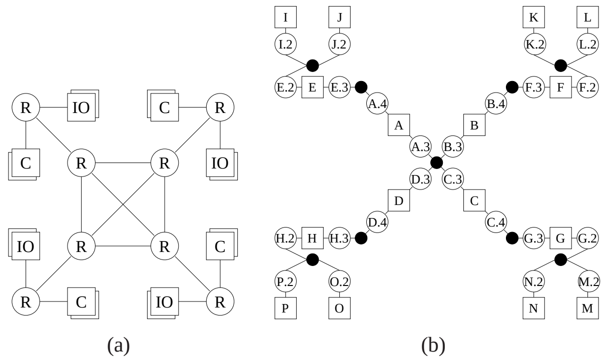

Fig. 10. (a) Network with fully connected core and (b) model composition graph

TABLE I
STATE-SPACE SIZES FOR CONFIGURATIONS (ROUTERS, CPU, I/O) OF THE CONNECTED SYSTEM

Config.	Detailed	Symm.	Compact	Rel. Size
(4, 2, 2)	3,600	2	1,830	51%
(6, 3, 3)	216,000	6	37,820	18%
(6, 6, 3)	2,985,984	48	224,841	8%
(6, 6, 6)	37,933,056	384	835,983	2%
(8, 4, 4)	12,960,000	24	1,406,294	11%
(8, 8, 4)	429,981,696	384	7,473,433	2%

State variables

Identifier	Description	Type
r_1	router state	private
r_2	cluster state	shared
r_3	link one	shared
r_4	link two	shared

Transition function

State	Event	Next State
1,0,0,0	re_1	0,0,0,0
1,0,0,0	re_2	0,1,0,0

Event set

Identifier	Description	Enabling Condition
re_1	fails safe	$\mu(r_1) = 1$
re_2	failure propagation	$\mu(r_1) = 1$

Fig. 11. Router model

VI. Examples and Results

In this section, we present modeling examples that demonstrate some of the symmetries that can be detected and exploited using our techniques. The algorithms presented are implemented as a state-space generator within Möbius. Atomic models are hierarchically created to form composed models. The state-space generator finds all of the symmetry in the composed model to produce a compact state space.

Figure 10-a is a diagram of the first example system. This system consists of four clusters interconnected by a two-level point-to-point network. The core of the system is a fully connected set of four routers. Each cluster contains several processor and I/O devices. Each time a device fails, there is a chance that the failure will propagate and the operation of the whole cluster will be disrupted. For this example we do not model failure propagation between routers. The routers at the clusters can disrupt or be disrupted by a processor or I/O device, but propagation of the failure of an outside router to a core router, or vice versa, has not been included in this example.

The composed model for this system comprises instances of three models: one for the routers, one for the processors and one for the I/O devices. The model for the router is described in Figure 11. The router model has four state variables and two events. If the router is functioning, $r_1 = 1$; otherwise it is zero. Likewise, if the cluster is functioning, $r_2 = 1$; else it is zero. The last two state variables are dummy variables, which will be used to represent connections to other routers. The two events correspond to the two types of failures that are possible. The first is a failure that is successfully handled by the fault-tolerance mechanism of the router. The second is a failure that propagates to connected components. The coverage depends on the state of the system, so the rates for the two failure events are state-dependent, which precludes the use of fault-trees. The rate function was omitted because the specific rates for the events do not affect the size of the state space. The processor and I/O device models are the same as the router model, except that they have one link instead of two. We assume these devices must be distinguished for the dependability measure.

The composed model for the network with a fully connected core is constructed by connecting the router, processor, and I/O device models together via their shared state variables. The model composition graph is shown in Figure 10-b. In Figure 10-b, A–H are instances of the router model, I, K, M, O are processor instances, and J, L, N, P are instances of the I/O device model. The fully interconnected core is modeled by a single connection node representing the superposition of $A.3$, $B.3$, $C.3$, and $D.3$. The other routers are each connected to a core router through a superposition of link variables. For example, router E is connected to core router A via the connection node $\{A.4, E.3\}$.

This system has multiple symmetries that can be exploited. The first detected symmetry is among the four clusters extending from the core. Interchanging any two of the four core routers and their associated clusters produces an automorphism. In addition to this core symmetry, analysis of each cluster detects the symmetry among redundant processors and redundant I/O devices within a cluster. Thus, for the simple configuration of eight routers, four processors, and four I/O devices, the only symmetry is the interchange of clusters, which produces twenty-four automorphisms. Adding a redundant processor to each of the four clusters yields the (8,8,4) configuration and results in 2^4 additional automorphisms. Combined with the twenty-four permutations of the four clusters, the order of the detected automorphism group grows to 384.

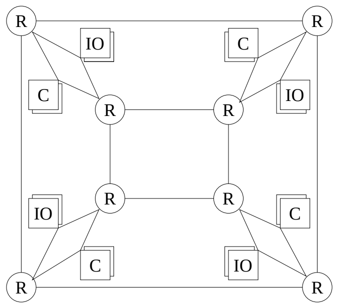

Fig. 12. Double ring network

TABLE II
STATE-SPACE SIZES FOR CONFIGURATIONS (ROUTERS, CPU, I/O) OF THE DOUBLE RING SYSTEM

Config.	Detailed	Symm.	Compact	Rel. Size
(8, 4, 4)	5,308,416	8	1,708,032	32%
(8, 8, 4)	157,351,936	128	8,895,748	6%

Table I shows the state-space compression that we achieved using the procedures presented in Sections III and IV for many different configurations.[1] Each configuration is specified by the number of routers, CPUs, and I/O units. For each configuration, the column labeled "Detailed" lists the size of the state space generated by the procedure of Figure 3. The "Symmetry" column lists the order of the automorphism group of the model composition graph for the system, and the "Compact" column lists the size of the state space generated by detecting and exploiting symmetry. Finally, we list the relative size of the compact state space in the last column of the table. As can be seen in Table I, our techniques produced reduced state spaces that were small relative to the detailed state spaces. The compression increases with the size of the state space and the order of the automorphism group.

Figure 12 is a diagram of a system where the routers are configured in a double ring. Clusters are the same as in the first example, but each one of the processors and I/O devices is connected to both rings. Failure propagation within a cluster is modeled, so the failure of a component in a cluster can potentially disrupt the entire cluster. Table II shows the results for the double ring system. The symmetry for the basic system with no redundancy within the cluster consists of four rotations and an interchange of inner and outer rings, yielding an automorphism group of order eight. Adding an extra processor to each cluster introduces 2^4 processor configurations, which increases the automorphism group to 128. For the configuration using eight routers, eight CPUs, and four I/O devices, the compact state space was six percent of the size of the detailed state space.

VII. CONCLUSION

We have presented a new approach to detecting and exploiting symmetry. As demonstrated in the last section, when there is symmetry in a model we can exploit it to achieve very good compression of the state space. In our approach, models retain the structure of the system, and all symmetry inherent in the structure of the model is detected and exploited for the purposes of state-space reduction. Many types of symmetries are detected and exploited, and the developed techniques do not require any assistance from the modeler. Results from group and graph theory are used as a rigorous foundation for the presented techniques. Specifically, we create a model composition graph from the model specification and then analyze the graph to find its automorphism group. Each model state is converted to its canonical label via a procedure using a stabilizer chain for the automorphism group, so that it is possible to generated a reduced state without visiting every detailed state. Using an implementation within Möbius, we obtained a large reduction in the size of the state space for several example models.

[1]The numerical results differ from previously published results due to an implementation error

REFERENCES

[1] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster, "The Möbius modeling tool," in *Proc. of the 9th International Workshop on Petri Nets and Performance Models*, 2001, pp. 241–250.

[2] D. M. Nicol, W. H. Sanders, and K. S. Trivedi, "Model-based evaluation: From dependability to security." *IEEE Trans. Dependable Sec. Comput.*, vol. 1, no. 1, pp. 48–65, 2004.

[3] N. Park and F. Lombardi, "Analysis of stratified testing for multichip module systems," *IEEE Transactions on Reliability*, 2002.

[4] A. Ramesh, D. Twigg, U. Sandadi, and T. Sharma, "Reliability analysis of systems with operation-time management," *IEEE Transactions on Reliability*, 2002.

[5] X. Teng, H. Pham, and D. Jeske, "Reliability modeling of hardware and software interactions, and its applications," *IEEE Transactions on Reliability*, 2006.

[6] M. Davio, "Kronecker products and shuffle algebra," *IEEE Trans. on Computers*, vol. C-30, Feb. 1981.

[7] B. Plateau and K. Atif, "Stochastic automata network for modeling parallel systems," *IEEE Transactions on Software Engineering*, vol. 17, no. 10, pp. 1093–1108, October 1991.

[8] B. Plateau and J.-M. Fourneau, "A methodology for solving Markov models of parallel systems," *Journal of Parallel and Distributed Computing*, vol. 12, no. 4, pp. 370–387, August 1991.

[9] P. Buchholz and P. Kemper, "Numerical analysis of stochastic marked graph nets," in *Proceedings of the Sixth International Workshop on Petri Nets and Performance Models*, 1995, pp. 32–41.

[10] P. Buchholz, "Equivalence relations for stochastic automata networks," in *Proc. of the 2nd International Workshop on the Numerical Solution of Markov Chains*, Raleigh, North Carolina, January 1995.

[11] S. Donatelli, "Superposed stochastic automata: A class of stochastic Petri nets with parallel solution and distributed state space," *Performance Evaluation*, vol. 18, no. 1, pp. 21–36, July 1993.

[12] P. Kemper, "Numerical analysis of superposed GSPNs," *IEEE Transactions on Software Engineering*, vol. 22, no. 9, pp. 615–628, September 1996.

[13] D. Deavours and W. H. Sanders, "'On-the-fly' solution techniques for stochastic Petri nets and extensions," in *Proceedings of the 7th International Workshop on Petri Nets and Performance Models (PNPM '97)*, St. Malo, France, June 3–6 1997, pp. 132–141.

[14] ——, "An efficient disk-based tool for solving very large Markov models," in *Computer Performance Evaluation: Proceedings of the 9th International Conference on Modelling Techniques and Tools (TOOLS '97)*, St. Malo, France, June 1997, pp. 58–71.

[15] G. Ciardo and A. S. Miner, "Storage alternative for large structured state spaces," in *Proc. of the 9th Int. Conf. of Modelling Techniques and Tools for Computer Performance Evaluation*, 1997.

[16] S. Derisavi, P. Kemper, and W. H. Sanders, "Lumping matrix diagram representations of markov models," in *Proc. of the 2005 Int. Conf. on Dependable Systems and Networks*, 2005.

[17] E. de Souza e Silva and P. M. Ochoa, "State space exploration in Markov models," *Performance Evaluation Review*, vol. 20, no. 1, pp. 152–166, June 1992.

[18] B. R. Haverkort, “In search of probability mass: Probabilistic evaluation of high-level specified Markov models,” *The Computer Journal*, vol. 38, no. 7, pp. 521–529, 1995.

[19] B. E. Aupperle and J. F. Meyer, “Fault-tolerant BIBD networks,” in *Proc. of the 18th International Symposium on Fault-Tolerant Computing (FTCS-18)*, Tokyo, Japan, 1988, pp. 306–311.

[20] ——, “State space generation for degradable multi-processor systems,” in *21st International Symposium on Fault-Tolerant Computing (FTCS-21)*, 1991, pp. 308–315.

[21] P. Buchholz, “Hierarchical Markovian models: Symmetries and reduction,” *Performance Evaluation*, vol. 22, no. 1, pp. 93–110, February 1995.

[22] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad, “Stochastic well-formed colored nets and symmetric modeling applications,” *IEEE Trans. on Computers*, pp. 1343–1360, November 1993.

[23] J. Hillston, *A Compositional Approach to Performance Modelling*, ser. Distinguished Dissertations in Computer Science. Cambridge, UK: Cambridge University Press, 1996.

[24] W. H. Sanders and J. F. Meyer, “Reduced base model construction methods for stochastic activity networks,” *IEEE Journal on Selected Areas in Communication*, vol. 9, no. 1, pp. 25–36, January 1991.

[25] A. K. Somani, “Reliability modeling of structured systems: Exploring symmetry in state-space generation,” in *Proc. of the 9-th Pacific Rim Conference*, Dec. 1997.

[26] G. Ciardo and K. S. Trivedi, “A decomposition approach for stochastic reward net models,” *Performance Evaluation*, vol. 18, pp. 37–59, 1993.

[27] W. Obal, II., “Measure-adaptive state-space construction methods,” Ph.D. dissertation, The University of Arizona, 1998.

[28] J. S. Leon, “Permutation group algorithms based on partitions, I: Theory and algorithms,” *Journal of Symbolic Computation*, vol. 12, pp. 533–583, 1991.

[29] B. D. McKay, “Practical graph isomorphism,” *Congressus Numerantium*, vol. 30, pp. 45–87, 1981.

[30] T. Miyazaki, “The complexity of mckay’s canonical labeling algorithm,” in *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, 1996.

[31] 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*, June 2004, pp. 530–534.

[32] G. Butler, *Fundamental Algorithms for Permutation Groups*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1991, vol. 559.

[33] S. H. Murray, “The schreier-sims algorithm,” 2003.

[34] http://mathworld.wolfram.com/IsomorphicGraphs.html.

W. Douglas Obal II received his Bachelor degree in Electrical Engineering from the Pennsylvania State University and received his Masters (1993) and Ph.D. (1998) in Electrical and Computer Engineering from the University of Arizona. After graduation, he joined Hewlett-Packard Labs until he passed away in 2005.

Michael G. McQuinn received his Bachelor degree in Computer Engineering in 2005 from the University of Illinois. He is currently a Ph.D. student in the Department of Electrical and Computer Engineering at the University of Illinois. His research interests include dependability modeling and efficient techniques to solve large Markov chains.

William H. Sanders is a Donald Biggar Willett Professor of Engineering and the Director of the Information Trust Institute at the University of Illinois. He is a professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory. He is a Fellow of the IEEE and the ACM. He serves as the Vice-Chair of IFIP Working Group 10.4 on Dependable Computing. In addition, he serves on the editorial boards of *Performance Evaluation* and *IEEE Security and Privacy* and is the Area Editor for Simulation and Modeling of Computer Systems for the *ACM Transactions on Modeling and Computer Simulation*. He is a past Chair of the IEEE Technical Committee on Fault-Tolerant Computing. Dr. Sanders’s research interests include performance/dependability evaluation, dependable computing, and reliable distributed systems. He has published more than 160 technical papers in these areas. He is a co-developer of three tools for assessing the performability of systems represented as stochastic activity networks: METASAN, *UltraSAN*, and Möbius. Möbius and *UltraSAN* have been distributed widely to industry and academia; more than 300 licenses for the tools have been issued to universities, companies, and NASA for evaluating the performance, dependability, security, and performability of a variety of systems. He is also a co-developer of the Loki distributed system fault injector and the AQuA/ITUA middlewares for providing dependability/security to distributed and networked applications.