SOLUTION OF GRAPH-COMPOSED MARKOV MODELS USING SYMMETRY DETECTION AND SYMBOLIC DATA STRUCTURES

BY

Michael Graham McQuinn

B.S., University of Illinois at Urbana-Champaign, 2005

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Adviser:

Professor William H. Sanders

ABSTRACT

Continuous time Markov chains (CTMCs) are a common mathematical model for studying the dependability of many complex processes and have been especially successful in modeling large computer systems. CTMCs are typically generated from more natural modeling formalisms that often better represent the system they model. This results in the famous state-space explosion problem, in which the number of states in the CTMC depends exponentially on the number of models in the higher-level formalism. The problem affects numerical analysis in two ways: the space needed to represent the transition rate matrix, and the space needed to represent the iteration vectors.

The goal of this thesis is to develop new techniques to extend the size of models that can be studied using CTMCs generated from higher-level formalisms. To combat the state-space explosion problem, we present a combination of a largeness-avoidance and a largeness-tolerance technique to address the size of the transition rate matrix. In the first technique, we present a representation of the model called the *model composition graph*, which separates a model into its public and private state. We then use symmetry detection to generate the smallest CTMC possible using model-level symmetry. In the second technique, we use symbolic data structures to represent the state space and transition rate matrix using minimal memory, leaving the rest for iteration vectors.

We combine the techniques for the case of graph-composed Markov models with state-sharing composition. Using several example models, we study the space and time efficiency of the techniques. We present results showing an orders of magnitude decrease in running time due to lumping of model-level symmetry, and an orders of magnitude decrease in space due to symbolic data structures. Since our techniques are complementary, we are able to get the benefits of both, which greatly improves our ability to solve these types of models.

ACKNOWLEDGMENTS

This work is based upon work supported in part by the National Science Foundation through the Central Europe Summer Research Institute, Pioneer Hi-Bred International, and France Telecom. Any opinions, findings, and conclusions or recommendations expressed in this material are mine and do not necessarily reflect the views of the National Science Foundation. I am grateful for the support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Modern society depends on the continuous functioning of many interdependent systems. Some of these systems have always been considered critical to our well-being (such as air traffic control), but many others have become equally critical to the functioning of society. For example, we rely on the security and availability of e-commerce systems to sustain our economy. We need our communication systems to be secure and reliable during states of emergency. Our critical infrastructures, such as electric power and gas, must be secure and available at all times. In short, modern society demands that many complex and interconnected systems be dependable.

To develop dependable systems, engineers need the ability to quantify the safety, security, and availability of the systems they are designing. There are two main methods to accomplish that: prototyping and modeling. Prototyping involves building an experimental version of the system and testing its functionality against the designed functionality. That process is often prohibitively expensive, especially for systems like satellites or aircraft.

A more cost-effective method is modeling, which represents the functionality of the system using a mathematical construct called a *model*. Models can describe the functionality of the system at varying degrees of abstraction. For example, fault trees can be used to describe the coarse relationship between nearly independent systems, while stochastic activity networks (SANs) can be used to detail the inner-workings of a single satellite. Regardless of the level of abstraction, a number of *measures* are defined on the model, which describe the characteristics of the system the modeler wants to study. Finally, algorithms compute the measures based on the model representation.

Continuous time Markov chains (CTMCs) [1] are a type of model com-

monly used to study the dependability of computer systems. A CTMC is defined by its *state space* and a set of transitions. The state space defines the set of valid states of the CTMC, while the set of transitions defines the exponentially distributed rates from a source state to a destination state.

In real systems, there are often too many states and transitions to determine manually, so efficient *state-space generators* (SSGs) have been developed to find the state space and the set of transitions. Many high-level formalisms have been developed to represent different systems, including SANs [2], stochastic Petri nets [3, 4, 5], and stochastic process algebras [6, 7, 8]. High-level formalisms give the modeler the power to easily represent complex systems, but efficient state-space generators are needed to convert each particular formalism into its CTMC representation.

High-level formalisms are not enough, however, to describe large, complex systems. Often, models of individual subsystems are created, and then "composed" to form larger models. Each individual submodel describes only a portion of the system, but when the submodels are composed together, the resulting model describes the entire system. The composition of models is arbitrary: models can be replicated as in a Replicate-Join network [9], or composed in a hypercube or any other general composition [10]. Some composed models can be solved more efficiently due to symmetries in the composition, which require special state-space generation algorithms.

Given a (possibly composed) model, state-space generators can create a representation of the underlying CTMC. Unfortunately, the size of the state space of the CTMC is often exponentially related to the number of components in the model, which results in the famous state-space explosion problem. The state-space explosion problem manifests itself in two ways: the memory required to store the *iteration* vector and the space required to store the transition rate matrix of the CTMC. Iteration vectors are needed in numerical analysis routines to store temporary vectors that are as big as the state space. The transition rate matrix stores the rates from a source state to a destination state. If the size of the state space is too large, then current techniques cannot represent the CTMC in main memory, which limits the size of systems that can be studied. The goal of this thesis is to reduce the effects of the state-space explosion problem for arbitrarily composed Markov models.

1.2 Related Work

There are two general techniques to reduce the effects of the state-space explosion problem. The first is called *largeness avoidance*, which seeks to reduce the size of the underlying CTMC to an equivalent, but smaller form. The second technique is called *largeness tolerance* and uses efficient data structures to represent the CTMC, so that it uses less main memory. We classify the related work into those two categories.

1.2.1 Largeness avoidance

Many techniques have been developed to "avoid" the state-space explosion problem by reducing the size of the underlying CTMC to a smaller, but equivalent CTMC. The reduction comes from the fact that often two states are the same and can be replaced with an equivalent "lumped" state. Techniques can exist at several levels of the model description: at the composed model level, at the individual model level, and at the level of the underlying CTMC. Often, operations at the level of the underlying CTMC are too costly, so model-level lumping is most productive.

Model-level lumping techniques use some property of the composed model to produce a smaller, equivalent CTMC. Although they do not always find the smallest lumped CTMC, they are often effective because the costs are much smaller than those of finding the smallest possible lumped CTMC. For example, for Replicate-Join composed models that share states, the reduction comes from replicas of the same submodel [9, 4], and similar techniques exist for process algebras [11, 12]. Also, model-level lumping techniques exist for hierarchical queuing networks [13], stochastic automata networks [14], and hierarchical Kronecker representations [15]. For more general composition, more complex techniques are needed to identify the lumpable states. For graph-composed models with shared state, Obal identifies symmetries in the composition that lead to lumpable states [10].

1.2.2 Largeness tolerance

Although largeness-avoidance techniques can reduce the size of the state space of the CTMC by orders of magnitude, the resulting CTMC may still

be too large to represent in main memory. Largeness tolerance techniques aim to reduce the size of the *representation* of the CTMC without altering its functional equivalence (like model-level lumping does, for example). Typically, new algorithms and data structures are needed to work on the new representation.

Decision diagrams have been successfully used to efficiently represent the state space of a large CTMC. For example, binary (BDD) and multi-valued decision diagrams (MDD) encode the states of a CTMC as paths in a graph. A single path through the graph corresponds to a selection of state variable values, which represents a single state.

MDD-based techniques have been used to represent the state space of models composed with action synchronization [16] and of Replicate-Join composition models with state sharing [17]. For action sharing composition, a technique called *saturation* explores local actions separately from global actions, which speeds up the techniques by a few orders of magnitude [18]. In [16], an assumption was made that local state spaces were known a priori (the so-called *logical product form property*), but this was later relaxed in [19]. Fewer results exist for state-sharing composition, but Derisavi et al. used extensions of the saturation procedure to develop efficient techniques for Replicate-Join composed models with state-sharing composition [20].

Many techniques also exist to efficiently represent the transition rate matrix of the underlying CTMC. Techniques include Kronecker representations, on-the-fly generation of entries, path-based techniques, disk-based representations, and distributed techniques. See [21, 22, 23] for recent overviews. Analogous to the MDD representation of the state space is the Matrix Diagram (MD) approach of Ciardo and Miner, which represents the transition rate matrix [24] of action-shared models that satisfy the product form property. Derisavi et al. presented time-efficient algorithms that use MDs for state-sharing Replicate-Join composed models based on the SAN formalism [17].

1.3 Contributions

The goal of this thesis is to extend the applicability of Markov models to study complex, distributed computer systems. We focus on a combination of

largeness-avoidance and largeness-tolerance techniques to study the largest systems possible. Specifically, in this thesis, we have:

- Developed model-level lumping techniques that work on general, graph-based composition formalisms. This technique uses model-level symmetry to produce the best lumping possible.
- Extended previous work in MDD and MD symbolic data structures for state-sharing graph-composed models.
- Integrated the model-level lumping technique of symmetry detection with symbolic data structures for graph-composed models.
- Investigated, though the use of example models, the space and time savings of our techniques.

1.4 Outline

The goal of this thesis is the combination of largeness-avoidance and largeness-tolerance techniques to combat the state-space explosion problem for graph-composed models. In Chapter 2, we present background material necessary to discuss the techniques, including symmetry detection, lumping, and the symbolic data structures. In Chapter 3, we present our model-level lumping techniques based on symmetry detection of the model, while in Chapter 4 we present the integration of symbolic data structures with the lumped representation. In Chapter 5, we investigate the benefits of the technique using several example models in the Möbius modeling tool. We conclude in Chapter 6.

CHAPTER 2

BACKGROUND

In this chapter, we present notation, terms, and concepts to discuss the largeness-avoidance and largeness-tolerance techniques presented in Chapters 3 and 4. We assume that the reader has a working knowledge of the following concepts: CTMC, state space, transition rate matrix, and numerical analysis of CTMCs. Therefore, we only present terms necessary to the presented discussion. For a detailed review of those topics, see [1] and [25].

2.1 Graph-Composed Markov Models

In this thesis, we study arbitrary, graph-composed Markov models. A composed model may consist of an arbitrary number of submodels, each of which is "connected" to a (possibly empty) set of other submodels through sharing of portions of its state. Less general compositions may restrict the types of systems that can be studied.

The model specification language is meant to simplify the discussion by providing the minimum notation needed to discuss the composition of models and the construction of the underlying CTMC. Many different formalisms for describing discrete event systems can be mapped onto the 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* $\mathbb{N}$*. The state of the model is defined as a mapping* $\mu : S \rightarrow \mathbb{N}$*, where for all* $s \in S$*,* $\mu(s)$ *is the value of state variable* s*. Let* $M = \{\mu \mid \mu : S \rightarrow \mathbb{N}\}$ *be the set of all such mappings.*

- E *is the set of events that may occur.*

- $\varepsilon : E \times M \rightarrow \{0, 1\}$ *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.*

- $\lambda : E \times M \rightarrow (0, \infty)$ *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*.*

- $\tau : E \times M \rightarrow M$ *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*.*

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 $\tau(e, \mu) = \mu'$. The probability is calculated as

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

Models are connected together through shared state variables to form "composed models." Figure 2.1 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 has $\{A.1, A.2\}$ and B has $\{B.1, B.2\}$. 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 that specification, the values of state variables $A.2$ and $B.1$ will always be equal and can conveniently be labeled 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 2.1, $C1$ is the connection representing the superposition of $A.2$ and $B.1$.

Systems composed of multiple identical subsystems exhibit symmetry. For example, consider Figure 2.2(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 2.2 serves well to demonstrate

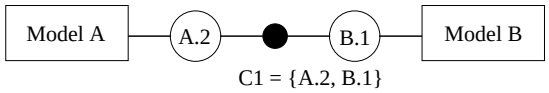


Figure 2.1: Models are connected through shared state variables

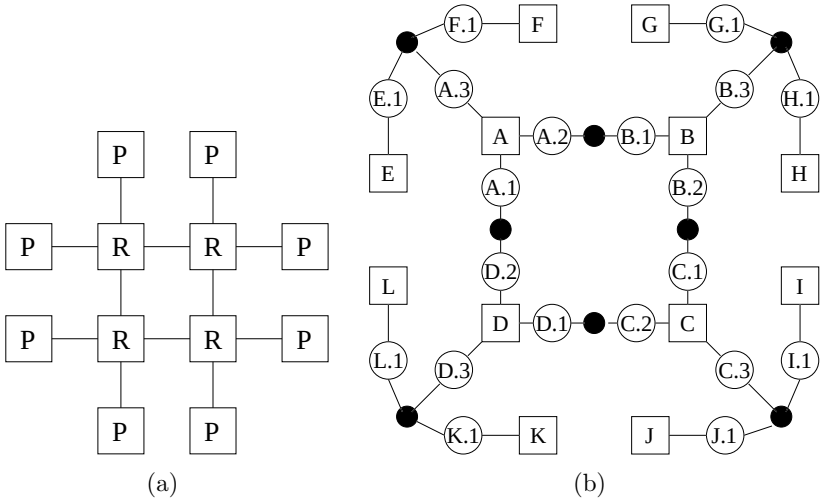


Figure 2.2: (a) Ring of dual processors system and (b) Model composition graph

symmetry ideas that will be more completely explored in Chapter 3. 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 that 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.*
- $\kappa : I \rightarrow \Sigma$ *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 2.1 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, each of which is 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 that approach to modeling, we use the example of Figure 2.2. 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 [10]; 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 2.2(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 the model composition graph, we have assumed that the ring has direction and the pro-

cessors 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. That connection indicates symmetric access to the network node. From Figure 2.2, 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 was the case with models, the *composed model state* is defined as a mapping $\mu : S \rightarrow I\!N$ 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 by 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 by μ_A.

Definition 3 *The* composed model state transition function *is defined as*

$\tau : E \times M \rightarrow M$, 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. 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 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 [26]. 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 chapter.

2.2 Group and Graph Theory

In the composed model formalism of Section 2.1, the structure of the model is exposed in the model composition graph. 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. 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. If that is the 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 that 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. The overloading is justified by the fact that elements of Γ are restricted by definition to mapping 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 re-naming 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

$$\mu^\gamma = \mu \circ \gamma, \tag{2.1}$$

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 2.1 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.

2.3 Symbolic Data Structures

MDDs and MDs are members of a family of data structures named *decision diagrams* that are rich in variation, theory, and terminology. 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 [23, 21]. MDDs and MDs are both rooted directed acyclic graphs. Both are associated with K variables $v_1, \ldots, v_K$. An *MDD* encodes a function $f : \times_{k=1}^{K} S_k \to M$ where $S_k = \{0, 1, \ldots, |S_k| - 1\}$ is the finite set of values that variable v_k can have[1] and M is the finite set of values of f. An *MD* is similar and encodes function

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

$g : \times_{k=1}^{K}(S_k \times S'_k) \rightarrow M$. We will focus on the special case of MDDs and MDs that are quasi-reduced, are ordered, and have Boolean outputs. Such diagrams have nonterminal 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 MD 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 MD 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 MDs shall both carry a numerical value per 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 MD, 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).

CHAPTER 3

SYMMETRY DETECTION AND EXPLOITATION

In this chapter, we present algorithms to detect and exploit model-level symmetry in graph-composed Markov models. Detecting symmetry involves the theory of finding the set of states that are lumpable, while exploiting symmetry involves the development of efficient algorithms to perform the lumping automatically.

3.1 Detecting Symmetry

We will now show how Γ, the set of automorphisms of the model composition graph, 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 1 *See [10].*

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 the action 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 by $\mathcal{D}(\mu_A)$. 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 2 *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 3 *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

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

Now, recalling that $B^\gamma = A$, it follows from Proposition 2 that

$$\tau(e,\mu)^\gamma = [\mu^\gamma]_{S-B} \cup \tau_B(e^\gamma, [\mu^\gamma]_B). \tag{3.1}$$

Finally, after rewriting (3.1) 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

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

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 $H \cap \Delta_\mu \neq \emptyset$. H will then 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 those 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.*

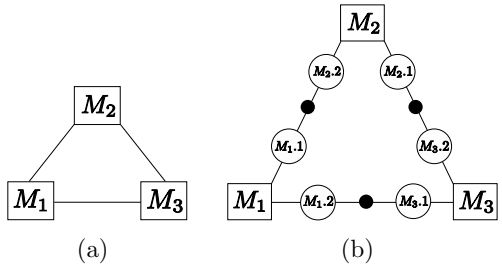
Proof 4 *$\mu_1 L \mu_2$ implies there exists γ such that $\mu_2 = \mu_1^\gamma$. Therefore, the transition functions $\tau(\cdot, \mu_1)$ and $\tau(\cdot, \mu_2)$ lead to the same destination classes by Proposition 3. By the definition of Γ, there is a one-to-one correspondence, $e \leftrightarrow e^\gamma$, 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 5 *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.



(a) (b)

Figure 3.1: (a) Example model and (b) Model composition graph

3.2 Exploiting Symmetry

As shown in Section 3.1, 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 indicates 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 3.1(a). Two instances of model M are connected by sharing of 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.

3.2.1 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

```
for each m ∈ I
   let V_p be the non-shared state variables in m
   CreateVertex(V_p, private)
   for each c ∈ 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(∅, connection node)
        AddEdge(node, {V_s, public})
   end for
end for
PartitionVertices()
```

Figure 3.2: Procedure to find the model composition graph of a composed model

Σ, since the connection set C may be different for different instances of the model. That may happen when a modeler creates a submodel for a component in a large system, but uses the submodel in slightly different ways.

Figure 3.2 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 3.1(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 represented by the private state variable fragment. The composed model is formed by joining the state variables $link1$ and $link2$ in a cycle formation.

3.2.2 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 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 [27]. A brute-force implementation could search all possible labelings 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 labelings [28, 29]. The software package *saucy* improves on the algorithm by using a sparse representation to minimize the memory used to solve the problem [30]. 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 nonidentity 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 the permutations are omitted for clarity.

3.2.3 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. 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 [31] 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 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.

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 [32]. 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, we find that 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 automorphisms can be generated by repeatedly applying the single automorphism. For example, by applying the automorphism twice, we find the second automorphism of the original group.

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 labeling to which α can be permuted using the automorphism group. Each vertex in the tree represents a possible valid labeling of the graph. An edge from node x to node y in the tree represents the automorphism that moves α from the xth position in the label to the yth 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 3.3 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 vth location of permutation p, and the methods *AddEdge* and $CreateVertex$ are defined as before. Starting from an element b of the base, we apply 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 labeling.

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 3.4 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

```
Frontier = α
NewFrontier = ∅
CreateVertex(α)
while Frontier ≠ ∅
   for each vertex v ∈ 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
```

Figure 3.3: Procedure to find the Schreier tree of a vertex α

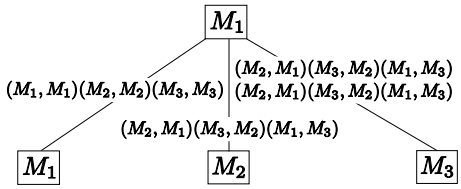


Figure 3.4: Schreier tree for base element M_1

because 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 to the position of a base element, which is exactly what is needed for the canonical labeling 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 3.5 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 3.6. An advantage of that 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 labeled state.

$B = [b_1, b_2, \ldots, b_n]$: 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

Figure 3.5: Procedure for canonical labeling 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 μ_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 μ' to canonical label
        if $\mu' \notin S$
            $S = S \cup \mu'$
            $U = U \cup \mu'$
        add arc from μ to μ' with rate $\lambda(e, \mu)$
    end for
end while

Figure 3.6: Procedure for generating compact state space for a composed model

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. That 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 labeling procedure in Figure 3.5, we only need to compare the state to two other states. The two other states are defined by the Schreier tree rooted in M_1 defined in Figure 3.4. 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 labeling are found. At that point, state-space generation begins and is carried out according to the procedure listed in Figure 3.6.

3.3 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.

3.3.1 Computing the automorphism group

To compute the automorphism group, we must first create the model composition graph. Analyzing Figure 3.2, we see that 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 [33]. However, our experience is that computing the automorphism group of the model composition graph rarely consumes more than 1 CPU second. This time is insignificant 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).

There are many different implementations of the Schreier-Sims algorithms for finding a base and strong generating set of an automorphism group. The fastest implementations have running times on the order of $O(n^2log^3(g) + tnlog(g))$, where g is the order of the group and t is the number of generators. Our simpler implementation has a slower $O(n^8logn)$ 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 one to calculate the Schreier trees for each element of the base. Analyzing Figure 3.3, we must apply every permutation to each node in the tree in the worst case. That leads to an $O(tn)$ 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 $O(btn)$.

3.3.2 Finding a canonical label

The canonical labeling procedure in Figure 3.5 is critical in the run time of our current implementation. It 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 labeling) 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, the cost 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 found 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. However, if we can dramatically reduce the number of states, we can dramatically reduce the running time of the total algorithm.

CHAPTER 4

SYMBOLIC DATA STRUCTURES

In this chapter, we describe methods to generate the state space and transition rate matrix for the lumped CTMC presented in Chapter 3 using symbolic data structures. We will represent the state space using an MDD and the transition rate matrix with an MD. We first present the techniques to generate the unlumped CTMC representation, and then show the projection to the lumped CTMC representation. We also utilize a small example model to illustrate the concepts.

4.1 Generation of Unlumped State Space and Generator Matrix

Based on the graph composed modeling technique discussed in the previous chapter, we now present methods to generate an MDD representation of the unlumped state space $\mathcal{S}$ and an MD representation of the generator matrix $\mathbf{Q}$. In Section 4.2, 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 that 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 [17]. Based on that distinction, we create a symbolic representation of the state space $\mathcal{S}$ and generator matrix $\mathbf{Q}$ by using

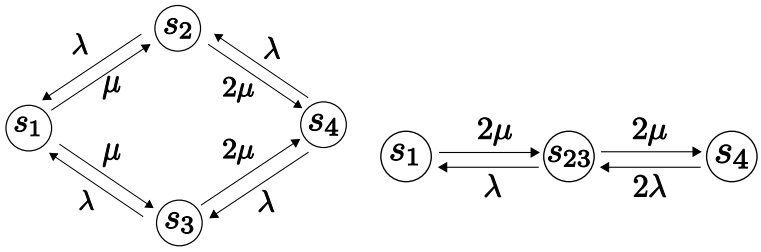


Figure 4.1: (a) Unlumped and (b) Lumped CTMC

known techniques from symbolic state space exploration for action-sharing formalisms [18] and state-sharing formalisms [17].

We consider an example model to help 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 $(0, v_{i2}, 1) \rightarrow (1, v_{i2}, 0)$ with rate $(1 + v_{i2})\mu$ and $(1, *, 0) \rightarrow (0, *, 1)$ 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 reach 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 4.1(a) shows the corresponding CTMC.

4.1.1 Unlumped state space

Given the distinction between local and global actions, we can extend the state-space generation routine for Replicate-Join composed models of [17] to operate on graph composed models. We omit the details and instead focus on the differences between the two algorithms.

For Replicate-Join composed models, the composition naturally forms a tree structure that can be mirrored in the MDD and MD 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 MD 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 MDs; however, at this point we are not investigating the effect of different variable orderings on the size of MDDs and MDs. 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 MD, but this potential for optimization is also not addressed in this thesis.

4.1.2 Rate matrix

The procedure to build an MD representation of the rate matrix is similar to that in [17] 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 MD that represent portions of the transition rate matrix. The entry (s_i, s'_i) of a matrix at level i of an MD, say at node $mxd[(s_1, s'_1), \ldots, (s_{i-1}, s'_{i-1})]$ in the MD, points to a matrix 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, nonzero matrix entries indicate a transition. Local matrices are kept for each level of the MD, which can easily be computed during state-space generation. When a state s is found to reach state s' with

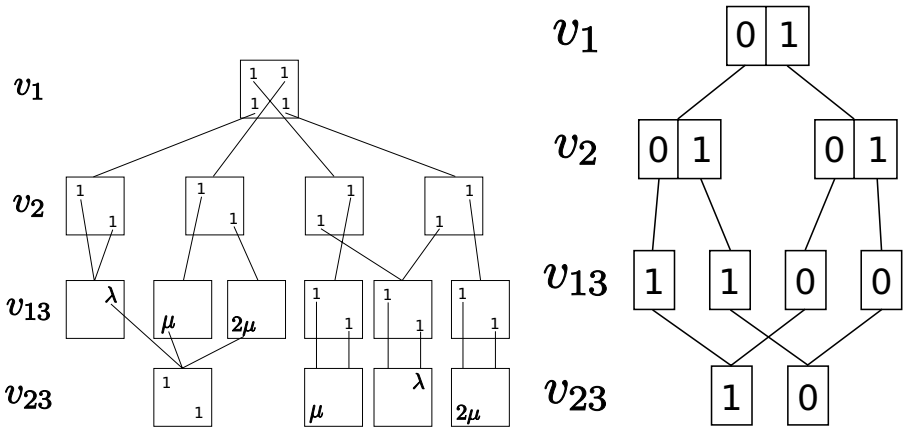


Figure 4.2: (a) MD representation of unlumped CTMC (b) Unlumped State Space

rate λ, we simply update the entries of the corresponding MD levels. Note that the differentiation between local and global actions still applies. When a local action fires, only MD nodes in one level will be affected, whereas a global action could affect many MD levels. See [17] for a complete discussion.

Example 2 *We can represent Q and $\mathcal{S}$ of the unlumped CTMC by an MD and an MDD. Figures 4.2(a) and 4.2(b) show the MD and MDD representations of the running example. Note that $K = |V_1| + |V_3|$.*

4.2 Projection on Lumped State Space

At this point, a symbolic state-space generator could be built using an MDD representation of $\mathcal{S}$ and an MD representation of $\mathbf{Q}$, and known MD-based numerical analysis techniques could be applied [24]. However, 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.

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 $\tilde{\mathcal{S}}$ 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 nonnegative 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 $\tilde{r} = \mathbf{rW}^\top$ 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 [34] for more details.

In Section 3.1, 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.

Example 3 *Continuing Example 1, states s_2 and s_3 can be lumped according to ordinary lumpability, which yields the lumped CTMC shown in Figure 4.1(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 4.1(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.

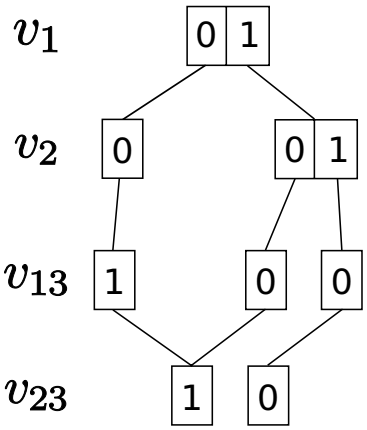


Figure 4.3: MDD representation of lumped state space

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 nonzero entry per row, namely $\mathbf{W}(i,j) = 1$ iff $s_i = rep(s_j)$. That 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.

Example 4 *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 4.3.*

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 $\tilde{\mathbf{D}}$.

4.2.1 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, ρ_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 [35] 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* [30], 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

```
REP(s_1, ..., s_K)
1  B ← Base for stabilizer chain
2  for each b_i ∈ B
3        do O^(i) ← Orbit of base element b_i
4           j ← 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 4.4: Pseudocode for finding a $rep(s)$

with respect to a lexicographic ordering of state variables.

The remaining problem is to find the set of permutations 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 4.4 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 that as an optimization, since the size of the indices is most often much smaller than the size of a state. In Figure 4.4, 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 4.5 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

```
DFSLump(S_MDD)
1  S̃_MDD ← empty MDD
2  for all (s_1, ..., s_K) in S_MDD
3        do (s̃_1, ..., s̃_K) = Rep(s_1, ..., s_K)
4           S̃_MDD ← S̃_MDD ∪ GenPath(s̃_1, ..., s̃_K)
```

Figure 4.5: Pseudocode for finding $\tilde{\mathbf{S}}$

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 cannot 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. That procedure is known and is performed without changing the structure of the MDD [23].

4.2.2 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 $\mathbf{WQ}$ represents the projection of the rows of $\mathbf{Q}$ to the lumped state space $\tilde{S}$. We are able to mimic that algorithmically with the help of an MDD of $\tilde{S}$ that guides us to select corresponding row entries from an MD 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 the computation on-the-fly, or 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 trade-off of the two approaches.

On-the-fly Calculation

The on-the-fly calculation is performed much like the $rep(s)$ procedure described in Figure 4.4. 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 that procedure is that $rep(s)$ will be called for each nonzero entry of $\mathbf{R}$ in each iteration step of an iterative solution method. The cost might 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 the 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 [17], 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 4.6 for an illustration of the mapping MDD for the running example.

4.2.3 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.

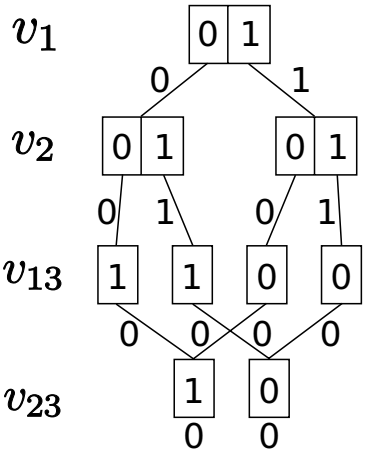


Figure 4.6: Mapping MDD

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 MD 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,[1] we focus on $\mathbf{p}(x+1) = \tilde{\mathbf{R}} \cdot \mathbf{p}(x)$.

In the algorithm presented in Figure 4.7, 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 avoids 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 $\mathbf{p}_j(x+1) = \mathbf{p}_j(x+1) +$

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

```
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 4.7: Pseudocode for matrix-vector multiplication

$\tilde{\mathbf{R}}(i,j)\mathbf{p}_i(x+1)$ 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 MD. 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.

CHAPTER 5

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 [36, 37], 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. We first study a fault-tolerant multi-processor system and then study the dependability of a lower earth orbiting (LEO) satellite network and a fault-tolerant parallel computer system. We then present results that argue for model-independent trends of the techniques.

5.1 Individual Model Results

5.1.1 Fault-tolerant multi-processor system

Figure 5.1 gives a diagram of our first example system and its associated MCG; for a complete description, see [10]. 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 cluster. A configuration of the model is specified by the number of routers, CPUs, and I/O units.

Table 5.1 summarizes the reduction in state size for different variants of the model. For example, the smallest configuration $(4, 2, 2)$ corresponds to

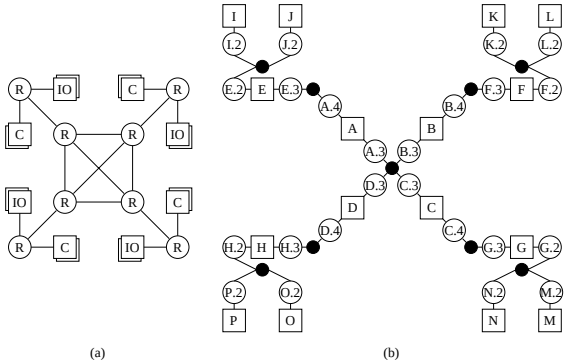


Figure 5.1: Fault-tolerant multi-processor: (a) Network with fully connected core and (b) Model composition graph

the configuration of 4 clusters, 2 CPUs, and 2 I/O units. The third column gives the cardinality of the unlumped state space $\mathcal{S}$, while the fourth gives that of the lumped state space $\tilde{S}$. The fifth 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 [10], 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.

Figure 5.2 shows the number of lumped and unlumped states for different configurations of the model. For the unlumped case, the state space explosion problem is easily seen, as the number of states depends exponentially on the size of the model. For the lumped case, however, the number of states remains much smaller because of the reductions from the symmetry detection. As the number of graph nodes for a model variant increases, more symmetry exists, which leads to more opportunities to lump states. This leads to a greater reduction in the number of states, which keeps the total number of states much lower than the unlumped case.

Table 5.2 presents the results from the computational group theory rou-

Table 5.1: Fault-tolerant multi-processor: 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

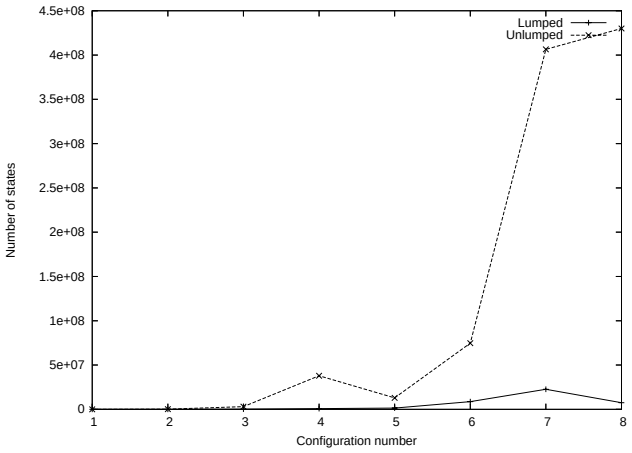


Figure 5.2: Fault-tolerant multi-processor: Number of states for different configurations

tines discussed in Section 4.2. 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 theoretically exponential running times, but for our class of problems $|V| + |E|$ is rather small, and we frequently observed running times of less than 1 s. 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.

The number of Schreier tree nodes dominates the running time of the $rep(s)$ canonical labeling procedure, which in turn dominates the running

Table 5.2: Fault-tolerant multi-processor: 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

time of the matrix-vector multiplications. Figure 5.3 summarizes how the number of nodes varies for different configurations of the model. It is important to see that the number of nodes remains relatively small even as the number of unlumped states grows exponentially. Also, note that as the symmetry in the model increases (as in configurations 4 and 8), the number of Schreier tree nodes increases as well, but the number of nodes still remains very small. 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.

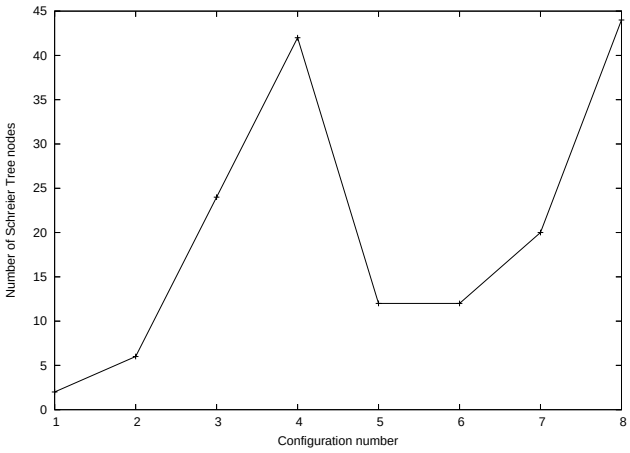


Figure 5.3: Fault-tolerant multi-processor: Number of Schreier tree nodes vs. N

The previous two figures depict the results for the on-the-fly mapping of states to representative states. Figure 5.2 shows how the number of states

Table 5.3: Fault-tolerant multi-processor: Mapping MDD results

	# Nodes	KB	# States
1	1,928	102,976	3,163
2	68,257	3,780,938	104,410
3	459,038	25,229,488	782,252
4	1,824,469	100,308,122	3,739,254
5	2,452,545	142,351,362	4,282,417
6	6,854,320	401,535,332	27,599,496
7	19,379,315	1,134,850,078	86,876,934
8	23,422,982	1,296,254,944	37,555,212

depends on different configurations, while Figure 5.3 describes how the group theory structures vary for different configurations. We also want to study the mapping MDD approach to map a state to its representative state. Table 5.3 presents the results of the mapping MDD technique. Column 1 gives the number of nodes in the symbolic structure, column 2 gives the total number of bytes in the structure, and column 3 gives the number of states that have mappings in the MDD. Figure 5.4 shows how the number of states for the unlumped, lumped, and mapping MDD vary for different sizes of models. The number of states in the mapping MDD is the number of reachable, but possibly not representative states, so this value will always be bounded by the number of unlumped and lumped states.

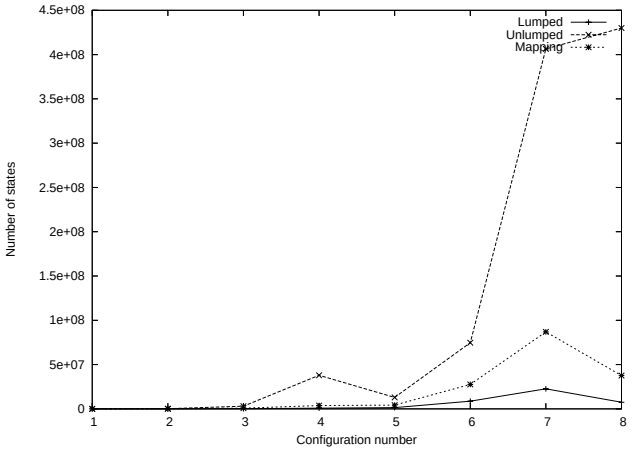


Figure 5.4: Fault-tolerant multi-processor: Number of states for unlumped, lumped, and mapping MDD

Figure 5.5 shows how the size of the symbolic structures varies for different

sizes of configurations. Although the number of states in the mapping MDD is bounded by the unlumped and lumped states, the size of the mapping MDD will typically be much greater than either of the other MDDs. This is because the edge weights that represent the mappings break up the structure, which results in less sharing among nodes. The size of the lumped state space MDD is greater than the size of unlumped state space MDD for similar reasons.

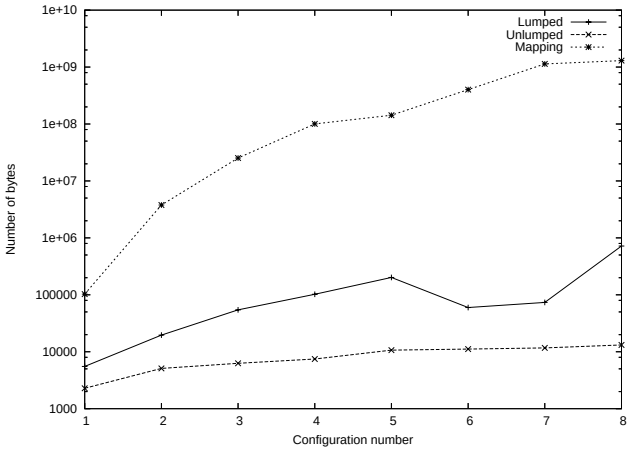


Figure 5.5: Fault-tolerant multi-processor: Number of bytes for unlumped, lumped, and mapping MDD

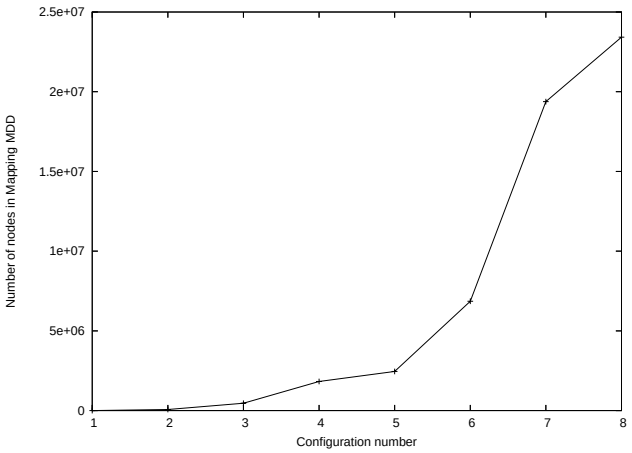


Figure 5.6: Fault-tolerant multi-processor: Number of nodes in mapping MDD for different configurations

Figure 5.6 shows how the number of nodes in the mapping MDD varies versus N. Unlike the on-the-fly mapping, the size of the data structure to

maintain the mapping grows exponentially for different configurations of the model. The number of bytes required to store the mapping MDD also increases exponentially, which can also be seen in Figure 5.5.

For the on-the-fly mapping, we have seen that the structures used to store the permutation information are small, and for the mapping MDD mapping, we have seen how the size of the MDD grows for different configurations of the model. The remaining important factor is to study the size of the MDD and MxD structures used to represent the lumped state space $\tilde{\mathcal{S}}$ and the rate matrix $\mathbf{R}$.

Tables 5.2 and 5.4 summarize the memory needed for the MDDs and MxDs to perform a matrix-vector multiplication as described in Section 4.2.3. 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 5.2 for final (column 3) and peak (column 4) memory consumption for the MDD of $\mathcal{S}$.

Table 5.4: Fault-tolerant multi-processor: Memory costs of lumped CTMC

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

Figures 5.7 shows the size of the unlumped MDD, and 5.8 shows the size of the lumped MDD sizes for different configurations of the model. What is important to see is that for large state spaces, the size of the structures remains very small, nearly 1 MB for the largest configuration. Similarly, Figure 5.9 shows the size of the MxD for different configurations of the model. Once again, even for the largest configuration, the space required for the rate

matrix MxD is still only about 1 MB, which is expected for symbolic data structures.

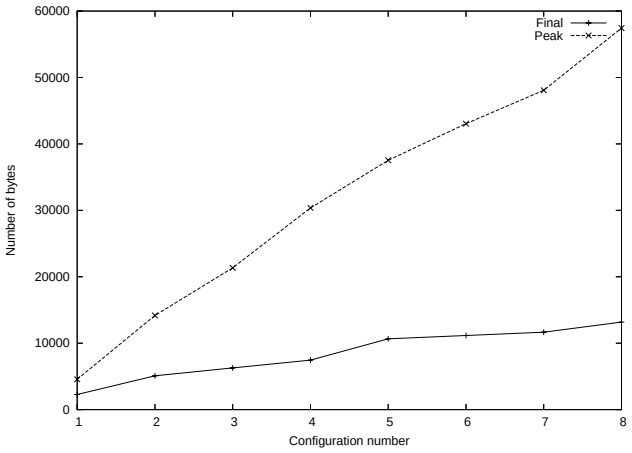


Figure 5.7: Fault-tolerant multi-processor: Number of peak and final bytes in unlumped MDD for different configurations

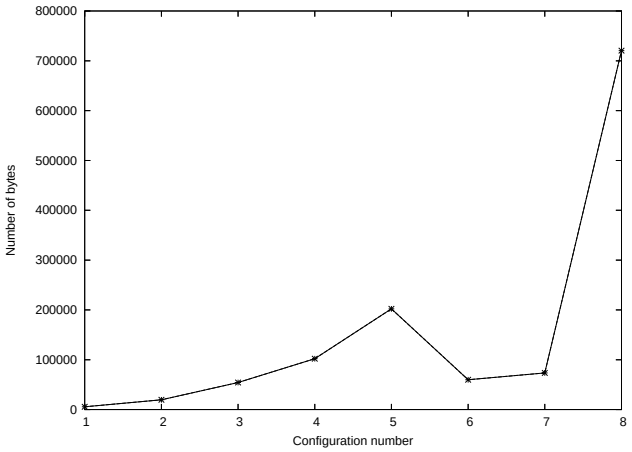


Figure 5.8: Fault-tolerant multi-processor: Number of bytes in lumped MDD for different configurations

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 algorithms 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

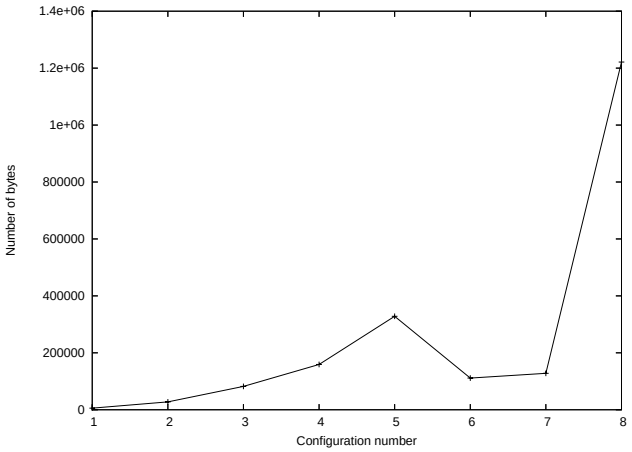


Figure 5.9: Fault-tolerant multi-processor: Number of bytes in MxD for different configurations

Table 5.5: Fault-tolerant multi-processor: Solution time per iteration for different configurations

Config.	Unlumped State Space	Lumped State Space	Time/Iter. (sec) On-the-fly	Time/Iter. (sec) Mapping MDD
(4, 2, 2)	3,600	1,830	.01740	.00110
(6, 3, 3)	216,000	37,820	1.21169	.04508
(6, 6, 3)	2,985,984	224,841	26.12247	.32824
(6, 6, 6)	37,933,056	835,983	210.81600	1.41522
(8, 4, 4)	12,960,000	1,406,294	129.14847	2.49144
(8, 6, 4)	74,649,600	8,652,240	1113.43573	9.74440
(8, 6, 6)	406,425,600	22,668,210	4861.35396	*
(8, 8, 4)	429,981,696	7,473,433	3493.42991	*

compute reward values for transient and steady-state distributions of such models. See Table 5.5 column 3 for a summary of the times to execute a single iteration of a numerical solver using the on-the-fly mapping of states and column 4 for times to execute a single iteration using the mapping MDD. Note that the mapping MDD technique could not be used for configurations 7 or 8 because it is prohibitively expensive. Figure 5.10 shows a logarithmic plot of the solutions times per iteration for both techniques. The mapping MDD performs much better for configurations 1-6, but becomes infeasible for configurations 7 and 8 because of the exponential growth of the size of the mapping MDD.

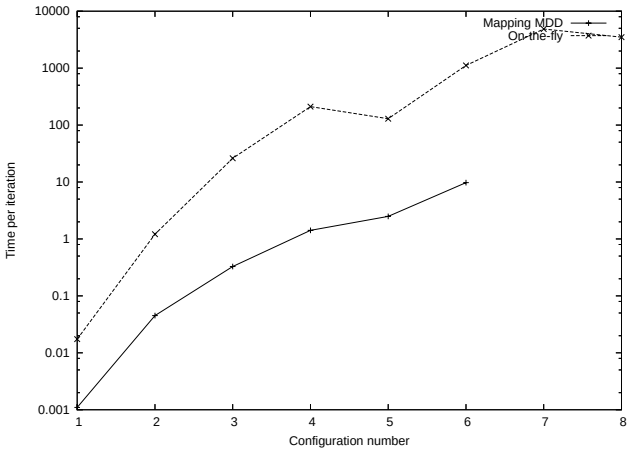


Figure 5.10: Fault-tolerant multi-processor: Time per iteration for both solution techniques

5.1.2 Dependabilty of fault-tolerant parallel computer system

The second example is a well-studied fault-tolerant parallel computer system [17], where a number of different models can be studied by varying the number of computers in the system. This model is formed using Replicate-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 Replicate-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 [17]. In this model, we observe slowdowns of 1.3 to 5 times for different variants.

5.1.3 Dependability of LEO satellite

A more recent study is that of a network of LEO satellites [38], 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 accommodate our techniques, the model in [38] 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 s per iteration.

5.2 General Results

It is believed that the order of a symbolic data structure's levels affects its size and efficiency. In this section, we study this phenomenon using the example models studied in the previous section to gain insight into trends that may be model independent.

In the procedure to find the unlumped state space, we performed a DFS on a random node. To study the effect of the ordering of levels, we performed a DFS on each of the private state variable fragments in the model composition graph. In addition, for each decision in the DFS (which neighbor to pick), we have a different ordering. We used this technique to study the (6,3,3) configuration of the fault-tolerant multi-processor. Figure 5.11 shows how the number of mapping MDD nodes varies for different DFS starting positions, while Figure 5.12 shows the total solution time. In the set of orderings, there is a factor of two difference between the ordering with the most nodes in the mapping MDD and the ordering with the least nodes. This results in a factor of two difference in solution time between all orders, as seen in Figure 5.12.

Similarly, we studied this phenomenon using the above technique on the second configuration of the parallel computer system (2 computers) to study how the mapping MDD is affected by different DFS starting positions. Figure 5.13 shows how the number of mapping MDD nodes changes for different starting positions, while Figure 5.14 shows how the solution time is affected. For this model, the variation is more pronounced. There is a factor of seven difference between the ordering with the most nodes and the ordering with the least nodes. This variation leads to a factor of 3.5 difference in total

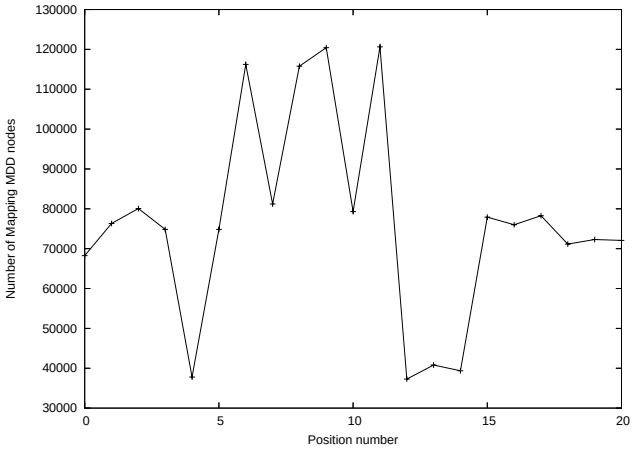


Figure 5.11: Fault-tolerant multi-processor: Number of mapping MDD nodes for various DFS starting positions for different configurations

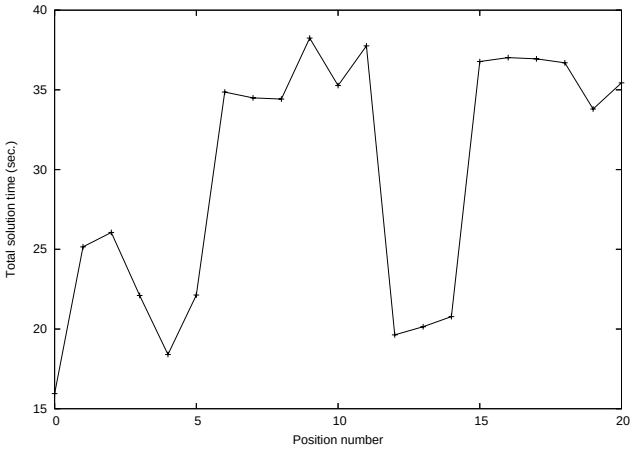


Figure 5.12: Fault-tolerant multi-processor: Mapping MDD solution times for various DFS starting positions

solution time.

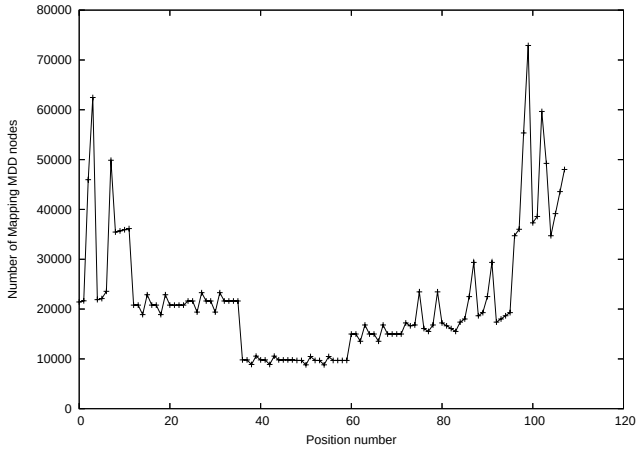


Figure 5.13: Parallel computer system: Number of mapping MDD nodes for various DFS starting positions for different configurations

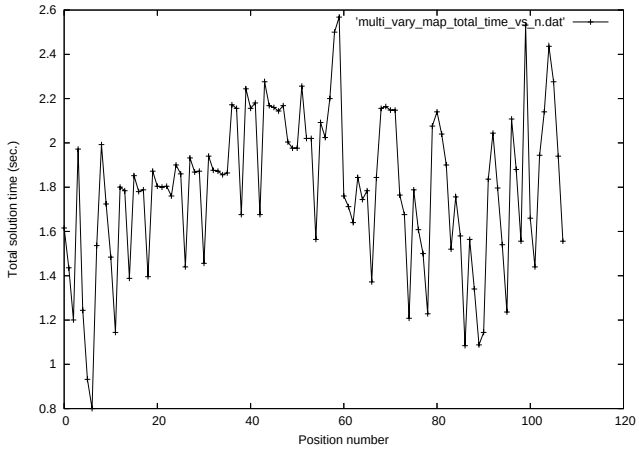


Figure 5.14: Parallel computer system: Mapping MDD solution times for various DFS starting positions

CHAPTER 6

CONCLUSION

The main goal of this thesis was to develop a combination of new techniques to efficiently solve large Markov models that are connected in an arbitrary fashion. Existing techniques limit the ways models can be composed and still be amenable to largeness-avoidance and largeness-tolerance techniques.

In Chapter 2, we presented a brief overview of models, specifically the topics related to the discussion of symmetry detection and symbolic data structures. We introduced the concept of the model composition graph, which separates the local and shared state of a model and allows us to use existing symmetry detection routines. Also, we provided an overview of symbolic (MDD and MD) data structures, which we use to represent the state space and transition rate matrix of the underlying CTMC.

In Chapter 3, we presented algorithms to detect symmetry in the model composition graph of composed models with state-sharing composition. We also presented a proof that the routines find all available model-level symmetry, and that they produce the coarsest lumpable CTMC with respect to the model-level symmetry. Also, we gave a detailed complexity analysis of the routines, which is important to understanding the implications of the added complexity.

In Chapter 4, we discussed techniques to apply symbolic data structures to the representation of the lumped CTMC created in Chapter 3. For efficiency reasons, we first create a symbolic representation of the unlumped CTMC and then create the representation of the lumped CTMC. We also presented routines to generate matrix entries on-the-fly using the canonical labeling procedures outlined in Chapter 3.

In Chapter 5, we tested the combination of our techniques using several example models already described in the literature. In each case, our symmetry detection routines were able to identify existing model-level symmetries, which reduced the size of the CTMC by several factors. Also, the remain-

ing CTMC could be represented in orders of magnitude less memory using symbolic data structures.

In the field of computer performability modeling, many techniques can be used to help solve large Markov models. In this thesis, we have presented two complementary techniques to help a modeler study a complex system. In addition, we have presented the *combination* of the two techniques, which helps us get the benefits of both. For the studied example models, we were able to reduce both the size of the CTMC and its representation by several orders of magnitude. Because the techniques are complementary, we believe they can be an effective technique for studying large, complex computer systems.

REFERENCES

[1] W. J. Stewart, *Introduction to the Numerical Solution of Markov Chains*. Princeton, NJ: Princeton University Press, 1994.

[2] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: Structure, behavior, and application," in *Proceedings of Petri Nets and Performance Models*, 1985, pp. 106–115.

[3] M. A. Marsan, G. Conte, and G. Balbo, "A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems," *ACM Transactions of Computer Systems*, vol. 2, no. 2, pp. 93–122, 1984.

[4] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad, "Stochastic well-formed colored nets and symmetric modeling applications," *IEEE Transactions on Computers*, vol. 42, no. 11, pp. 1343–1360, 1993.

[5] S. Donatelli, "Superposed generalized stochastic Petri nets: Definition and efficient solution," in *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*. London, UK: Springer-Verlag, 1994, pp. 258–277.

[6] M. Bernardo and R. Gorrieri, "A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time," *Theoretical Computer Science*, vol. 202, no. 1-2, pp. 1–54, 1998.

[7] P. Buchholz, "Markovian process algebra: Composition and equivalence," in *Proceedings of the 2nd International Conference on Process Algebras and Performance Modelling*, 1994, pp. 11–30.

[8] H. Hermanns and M. Rettelbach, "Syntax, semantics, equivalences, and axioms for MTIPP," in *Proceedings of the 2nd International Conference on Process Algebras and Performance Modelling*, 1994, pp. 71–87.

[9] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for Stochastic Activity Networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 1, pp. 25–36, 1991.

[10] W. Douglas Obal II, M. G. McQuinn, and W. H. Sanders, "Detecting and exploiting symmetry in discrete-state Markov models," in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 26–38.

[11] S. Gilmore, J. Hillston, and M. Ribaudo, "An efficient algorithm for aggregating PEPA models," *IEEE Transactions on Software Engineering*, vol. 27, no. 5, pp. 449–464, 2001.

[12] H. Hermanns and M. Ribaudo, "Exploiting symmetries in stochastic process algebras," in *Proceedings of the 12th European Simulation Multiconference on Simulation - Past, Present and Future*. SCS Europe, 1998, pp. 763–770.

[13] P. Buchholz, "Lumpability and nearly-lumpability in hierarchical queueing networks," in *Proceedings of the International Computer Performance and Dependability Symposium on Computer Performance and Dependability Symposium*. Washington, DC, USA: IEEE Computer Society, 1995, p. 82.

[14] A. Benoit, L. Brenner, P. Fernandes, and B. Plateau, "Aggregation of stochastic automata networks with replicas," *Linear Algebra and its Applications*, vol. 386, pp. 111–136, Jul. 2004.

[15] P. Buchholz, "A framework for the hierarchical analysis of discrete event dynamic systems (habilitations thesis)," Ph.D. dissertation, Universitat Dortmund, 1996.

[16] A. S. Miner and G. Ciardo, "Efficient reachability set generation and storage using decision diagrams," in *Proceedings of the 20th International Conference on Application and Theory of Petri Nets*. London, UK: Springer-Verlag, 1999, pp. 6–25.

[17] S. Derisavi, P. Kemper, and W. H. Sanders, "Symbolic state-space exploration and numerical analysis of state-sharing composed models," *Linear Algebra and Its Applications*, vol. 386, pp. 137–166, 15 July 2003.

[18] G. Ciardo, R. Marmorstein, and R. Siminiceanu, "The saturation algorithm for symbolic state-space exploration," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 1, pp. 4–25, 2006.

[19] G. Ciardo, R. M. Marmorstein, and R. Siminiceanu, "Saturation unbound," in *TACAS*, ser. Lecture Notes in Computer Science, H. Garavel and J. Hatcliff, Eds., vol. 2619. Warsaw, Poland: Springer, 2003, pp. 379–393.

[20] S. Derisavi, "Solution of large Markov models using lumping techniques and symbolic data structures," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2005.

[21] P. Buchholz and P. Kemper, "Kronecker based matrix representations for large Markov models," in *Validation of Stochastic Systems*. Berlin, Germany: Springer, 2004, pp. 256–295.

[22] R. Mehmood, "Serial disk-based analysis of large stochastic models," in *Validation of Stochastic Systems*. Berlin, Germany: Springer, 2004, pp. 230–255.

[23] A. S. Miner and D. Parker, "Symbolic representations and analysis of large probabilistic systems." in *Validation of Stochastic Systems*. Berlin, Germany: Springer, 2004, pp. 296–338.

[24] G. Ciardo and A. Miner, "A data structure for the efficient Kronecker solution of GSPNs," in *International Workshop on Petri Nets and Performance Models*, 1999, pp. 22–31.

[25] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. Chichester, UK: John Wiley and Sons Ltd., 2002.

[26] B. E. Aupperle and J. F. Meyer, "Fault-tolerant BIBD networks," in *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, Tokyo, Japan, 1988, pp. 306–311.

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

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

[29] T. Miyazaki, "The complexity of McKay's canonical labeling algorithm," in *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, 1996, pp. 239–256.

[30] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov, "Exploiting structure in symmetry generation for CNF," in *Proceedings of the 41st Design Automation Conference*, June 2004, pp. 530–534.

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

[32] S. H. Murray, "The Schreier-Sims Algorithm," master's thesis, Australian National University, 2003.

[33] E. W. Weisstein, "Isomorphic graphs," July 17, 2007, http://mathworld.wolfram.com/IsomorphicGraphs.html.

[34] P. Buchholz, "Exact and ordinary lumpability in finite Markov chains," *Journal of Applied Probability*, vol. 9, pp. 59–75, 1994.

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

[36] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster, "The Möbius framework and its implementation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 956–969, 2002.

[37] S. Derisavi, P. Kemper, W. H. Sanders, and T. Courtney, "The Möbius state-level abstract functional interface," *Performance Evaluation*, vol. 54, no. 2, pp. 105–128, 2003.

[38] E. Athanasopoulou, P. Thakker, and W. H. Sanders, "Evaluating the dependability of a LEO satellite network for scientific applications," in *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*. Washington, DC, USA: IEEE Computer Society, 2005, p. 95.