# SOLUTION OF LARGE MARKOV MODELS USING LUMPING TECHNIQUES AND SYMBOLIC DATA STRUCTURES

BY

SALEM DERISAVI

M.S., University of Illinois, 2003
B.S., Sharif University of Technology, 1999

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

# Abstract

Continuous time Markov chains (CTMCs) are among the most fundamental mathematical structures used for performance and dependability modeling of communication and computer systems. They are often constructed from models described in one of the various high-level formalisms. Since the size of a CTMC usually grows exponentially with the size of the corresponding high-level model, one often encounters the infamous state-space explosion problem, which often makes solution of the CTMCs intractable and sometimes makes it impossible. In state-based numerical analysis, which is the solution technique we have chosen to use to solve for measures defined on a CTMC, the state-space explosion problem is manifested in two ways: 1) large state transition rate matrices, and 2) large iteration vectors.

The goal of this dissertation is to extend, improve, and combine existing solutions of the state-space explosion problem in order to make possible the construction and solution of very large CTMCs generated from high-level models. Our new techniques follow largeness avoidance and largeness tolerance approaches. In the former approach, we reduce the size of the CTMC that needs to be solved in order to compute the measures of interest. That makes both the transition matrix and the iteration vectors smaller. In the latter approach, we reduce the size of the representation of the transition matrix by using symbolic data structures.

In particular, we have developed the fastest known CTMC lumping algorithm with the running time of $\mathcal{O}(m \log n)$, where $n$ and $m$ are the number of states and non-zero entries of the generator matrix of the CTMC, respectively. The algorithm can be used both in isolation and along with all compositional lumping algorithms, including the one we have proposed in

this dissertation. We have also combined the use of multi-valued decision diagram (MDD) and matrix diagram (MD) symbolic data structures with state-lumping techniques to develop an efficient symbolic state-space exploration algorithm for state-sharing replicate/join composed models that exploits lumpings that are due to equally behaving components created by the replicate operator. Finally, we have developed a new compositional algorithm that lumps CTMCs represented as MDs. Unlike other compositional lumping algorithms, our algorithm does not require any knowledge of the modeling formalisms from which the MDs were generated. Our approach relies on local conditions, i.e., conditions on individual nodes of the MD, which are often much smaller than the state transition rate matrix of the overall CTMC. We believe that our new approach has a simpler formulation, and thus is easier to understand.

To my parents, symbols of strong will and unconditional love.

To my sisters and symbols of care, Nejat and Elham.

To Reza, whose brotherhood is the most invaluable treasure I have.

Last but not least, to Roya, my dream of a happier life.

# Acknowledgments

I would like to thank my advisor, Professor William H. Sanders, for his invaluable technical advice and personal support throughout my studies. He was always eager and skillful in teaching me some of his grand technical visions and intuitions. I would also like to thank Professor Peter Kemper, from the University of Dortmund, and Professor Holger Hermanns from Saarland University, for all the precious ideas and comments without which the contributions of this dissertation would not have been realized. A lot of thanks goes to Ms. Jenny Applequist for the time she spent correcting my endless writing mistakes, not only in this dissertation but also in all my published works. She was always very patient answering all my questions regarding writing style. She, more than anybody else, taught me how to write more clearly.

I would also like to thank those members of the Möbius group with whom I had the honor to work: Amy Christensen, Graham Clark, Tod Courtney, David Daly, Dan Deavours, Jay Doyle, Shravan Gaonkar, Mark Griffith, Vinh Lam, Eric Rozier, and Patrick Webster. I am also grateful to other people in the PERFORM group who were such great people to interact with: Michel Cukier, Vishu Gupta, Michael Ihde, Kaustubh Joshi, Sudha Krishnamurthy, Ryan Lefever, James Lyons, Hari Ramasamy, and Sankalp Singh.

This material is based upon work supported by the National Science Foundation (NSF) under Grant Nos. 9975019 and CCR-00-86096. 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 NSF's support of my research.

Finally, I would also like to thank all my other friends, without whom I would not be the

person I am and without whom I would not enjoy my life in Urbana-Champaign. To name a few: Nawal Ali, Atef Alrashidi, Abbas Aminmansour, Hamed Asghari, Shadi Ashnai, Amir Behgooy, Vahid Behravan, Hamid Reza Chitsaz, Mohsen Dadfarnia, Silva Dushku, Reza Etebari, Mahnaz Fallah, Babak Farzad, Azadeh Farzan, Azadeh Fotohi, Yashar Ganjali, Bijan Ghahreman, Erfan Ghazinezami, Foojan Ghahramani, Zahra Golshani, Shirin Habib, Afsaneh Hajiamin, Hannaneh Hajishirazi, Hadi Jorati, Fatemeh Khalili, Faezeh Koohestani, Reza Mahani, Sima Mahani, Arash Mahdian, Mohammadhossein Mazarei, Maziar Mirhosseini, Hossein Namazi, Alireza Namazifard, Romina Nikoukar, Abdolreza Osouli, Mahdi Rastad, Bardia Sadri, Nasrin Sarrafi, Mayssam Sayyadian, Azadeh Shakery, Hojjat Sharifi, Babak Shotorban, Ali Toossi, Arash Termehchi, Roozbeh Touri, Farshid Zaker, and Reza Ziaei.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

Computer and communication systems are a central part of the infrastructure on which information technology is built. As the demand for this technology grows, particularly through the widespread use of the Internet, the need for faster, smaller, cheaper, and more dependable systems also grows, and the resulting improvements often translate into more sophisticated system designs.

Building such systems is an expensive process, and errors are often unacceptable and sometimes even catastrophic (for example, in a mission-critical system). Therefore, deviation from the requirements specification, i.e., the desired behavior, must be kept to a minimum. However, it is impossible to detect the difference between the desired and the actual behavior before building some working version of the system. Two major solutions for this problem are prototyping and modeling. In prototyping, an experimental functional version, or prototype, of the whole or part of the system is built. It is then tested to verify that it meets its specifications. Even though it is less expensive to build a prototype than to build the fully functional final version of the system, it can still be too costly in terms of time and money.

In modeling, a mathematical entity, or *model* (such as a fault tree or a Markov chain), is used to represent the functionality of the system with some level of abstraction. Based on the requirements specification, a number of *measures* of interest are also defined on the model. Then, a model analysis algorithm computes those measures and shows the modeler the extent to which the requirements specification is achieved by the model. By modeling

the different design alternatives and/or changing the parameters of the design, a modeler can eventually find the design that satisfies the goals of the system in an acceptable possible manner.

Determining the level of abstraction at different stages of the modeling process is an art and is left up to the modeler. If the modeler includes more details in the model, i.e., makes the model less abstracted, the model becomes larger and more complicated, and its analysis will be more difficult or even intractable. Analysis of such large models has motivated much research in areas in which modeling proves more useful than prototyping.

Continuous time Markov chains (CTMCs) [87, 90] are a model type widely used in modeling various types of computer and network systems of different complexities. A CTMC consists of a set of states, called the *state space*, and a set of exponentially distributed state transitions, each of which is specified by a source state, a destination state, and a rate. One of the main advantages of using CTMCs is the vast amount of research that has been done on discovering their properties and designing efficient analysis techniques for them. Those techniques mean that a wide variety of requirements specifications can be verified by CTMC-based modeling.

Since the states and transitions of a CTMC are too fine-grained to be mapped to elements of a real-world system, CTMCs are rarely used directly to specify a system's model in a typical modeling process. Numerous high-level modeling formalisms (languages) have been created to fill the gap between CTMC specification and system design specification. Examples of those formalisms include variants of stochastic Petri nets [2, 24, 45, 69], variants of stochastic process algebras [9, 13, 52, 55], and interactive Markov chains (IMCs) [51]. There are automated tools, called *state-space generators* (or *explorers*), that convert the high-level specification of a model into its equivalent underlying CTMC.

There are two approaches to building models of systems using high-level formalisms. In the first approach, a "flat" model is built monolithically, such that all parts of the system are modeled as part of a single entity. As a result, a flat model cannot be divided into smaller,

logically distinguishable submodels. In the second approach, which is often used in practice, a "composed" model of a system is built by combining a number of smaller interacting submodels, each of which serves a specific function in the overall model. Generally, each submodel models a part of the original system. By breaking a model into submodels, the modeler makes the modeling process more manageable and easier to understand. Most importantly, special analysis algorithms exist that solve composed models more efficiently.

Like the CTMC-based models themselves, measures related to the performance, availability, and dependability of a model can be specified both at the state level and using a high-level formalism. For example, in Markov reward processes (MRPs) [50, 83] a CTMC is defined along with a set of *rate rewards* (reward values associated with a state) for each state of the CTMC. There are also high-level stochastic logics such as CSL [3] and CSRL [4] that enable the modeler to define sophisticated measures on the model. Computation of a wide variety of measures on a CTMC often requires reachability analysis (i.e., state-space exploration), steady-state analysis, and transient analysis algorithms.

There are three approaches to performing steady-state and transient analysis of a CTMC-based model: 1) closed-form analytical solution, 2) discrete event simulation, and 3) state-based numerical analysis. The first approach is the fastest approach, but also the most restricted. It works only with models that satisfy a set of strict structural properties. Simulation is the least restrictive approach, in the sense that it can be applied to models with a wide variety of characteristics. However, it can be computationally expensive in the case of rare events, which are due to different time scales in the model. State-based numerical analysis techniques, or *numerical analysis* techniques for short, can compute a wide variety of measures for a relatively large class of models. All the techniques above have their own advantages and disadvantages. In this dissertation, we will focus on relaxing some of the practical limitations of solving high-level Markovian models using numerical analysis techniques by improving upon and extending existing related work.

Despite many breakthroughs, numerical analysis is still limited by the *state-space explo-*

*sion* problem. The size of the model representation in the high-level formalisms grows as the complexity of systems grows. However, a linear increase in the size of the high-level representation of the model could cause an exponential increase in the size of the underlying CTMC. For example, adding a single place to a stochastic Petri net can multiply the size of the state space of the CTMC by a constant factor.

Furthermore, numerical analysis algorithms often require some representation of the state transition rate matrix of the CTMC and one or more *iteration* vectors, each of which has as many elements as the number of states of the CTMC. The main goal of research on the topic of state-space explosion is to make the representation of the state transition matrix and/or the iteration vectors as small as possible. The topic of this dissertation is to extend and improve existing methods of attacking the state-space explosion problem.

It is worth mentioning that there are mathematical models that are more expressive than Markov chains. In other words, they can be used to model some systems that are impossible to model at the same level of abstraction using CTMCs. For instance, non-probabilistically-specified non-determinism cannot be expressed using CTMCs. An example of the more expressive modeling formalisms is the MoDeST language [35]. However, there is a trade-off here: the more expressive a modeling language is, the more difficult it is to analyze. Although some languages have more expressive power than CTMCs, a narrower spectrum of measures can be computed for them than for CTMCs using existing analysis algorithms. In particular, some measures that can be computed efficiently for a CTMC-based modeling formalism would be impossible to compute for a more expressive modeling formalism. We believe that CTMCs maintain an appropriate balance between expressiveness and analytical power, since they have been used to model and analyze a wide variety of computer and network systems.

## 1.2 Related Work

We classify existing construction and solution techniques for CTMCs built from high-level formalisms into two categories: largeness avoidance and largeness tolerance. We describe the related work in both categories below.

### 1.2.1 Largeness Avoidance

Many CTMC construction and solution techniques that have been developed for large CTMCs can be classified as *largeness avoidance* techniques, in which certain properties of some representation of the model (ranging from the high-level specification of the model to the underlying CTMC itself) are exploited to reduce the size (in number of states and transitions) of the underlying CTMC that needs to be solved to obtain the measures of interest for the model. In fact, the amount of information at the level of the CTMC is often too detailed for the computation of measures; therefore, minimal representations that still give correct results for the measures are desired.

For example, state lumping (e.g., [12, 62]) is an approach that reduces the size of a CTMC by considering the quotient of the CTMC with respect to an equivalence relation (i.e., it replaces a set of states with a single lumped state) that preserves the Markov property and supports the desired performance measures defined on the CTMC. By solving the smaller CTMC, it is possible to compute exact results for the larger CTMC, and therefore measures of interest for the model.

A *state-level* lumping technique is a lumping technique that exploits the lumping properties at the CTMC level. The main advantage of state-level lumping techniques is that they generate the optimal (i.e., smallest possible) lumped CTMC. However, since they can perform efficiently only on a sparse matrix representation of a CTMC, they have prohibitive space requirements for very large CTMCs; therefore, they are usually used along with other CTMC solution techniques. Buchholz [19] gives a state-level lumping algorithm with $\mathcal{O}(mn)$

time complexity and $\mathcal{O}(m+n)$ space complexity for computing the optimal (i.e., coarsest) lumping of a CTMC represented as a sparse matrix, where $n$ is the number of states and $m$ is the number of non-zero entries of the generator matrix of the CTMC. For fairly large CTMCs that can be stored in memory (a CTMC with roughly $10^7$ transitions), the algorithm's use is limited.

Several authors have also addressed the problem of computing bisimilarity [70], which is, in some ways, similar to the problem of state-level CTMC lumping. Kanellakis and Smolka gave a partition refinement algorithm with time complexity $\mathcal{O}(mn)$ [61][1]. They conjecture that an algorithm exists that reduces the time complexity to $\mathcal{O}(m \log n)$. A few years later Paige and Tarjan designed such an algorithm [76]. An implementation of Paige and Tarjan's algorithm can be found in [46]. Supposedly based on [76], $\mathcal{O}(m \log n)$ complexity has been claimed without proof by Bernardo and Gorrieri [9] for CTMCs and by Huynh and Tian [58] for discrete time Markov chains (DTMCs).

In contrast, *model-level* lumping techniques identify appropriate lumping properties by operating on a higher-level formalism and directly constructing a lumped CTMC, rather than by constructing the unlumped CTMC and then operating on it. The lumping equivalence relation is established by the modeling formalism itself in some model-level lumping techniques. That holds for stochastic well-formed nets (SWNs) [24] and replicate/join operators in stochastic activity network-based composed models (SANs) [82] in which the lumping results from equivalence of the replicas of a particular submodel. Extending the work of [82], Obal proposes [60] the graph composition formalism and uses the symmetry detection technique, a type of model-level lumping technique. The technique automatically identifies and exploits all the structural symmetries that are due to the interaction between submodels of a state-sharing composed model, that is, a model consisting of submodels that share a subset of their state variables. Restricted versions of a symmetry detection technique similar to the one described in [82] have also been used for process algebras in [47, 53]. Model-level

[1]For bisimilarity computation, $m$ is the number of transitions.

lumping techniques have also been proposed for SANs (stochastic automata networks) [7], hierarchical queuing networks [16], and hierarchically Kronecker representations [17]. Unlike state-level lumping, model-level lumping techniques do not always find the optimal lumping, because they are limited to those symmetries that can be identified from a given model description and they do not operate at the CTMC level.

Other lumping techniques, which we call *compositional lumping* techniques, can be applied to composed models provided that the specific high-level formalism satisfies a particular set of assumptions. In these techniques, each of the individual interacting submodels is lumped separately from the others using a state-level lumping algorithm, and is then replaced in the overall model by its lumped version. Like model-level lumping techniques, most compositional lumping techniques are formalism-dependent; specifically, they rely on properties of the composition operator. For example, based on the fact that lumping is a congruence with respect to parallel composition in a number of process algebra formalisms and stochastic automata networks (SANs), compositional lumping can be used in those formalisms to generate lumped state spaces [13–15, 17, 18, 51].

Most of the work on compositional lumping applies only state-level lumping inside the submodels. In some cases, in addition to the lumping of each of the submodels, the structural symmetry of the interaction among the submodels may also be exploited to achieve even smaller CTMCs. In other words, for some composed models, a lumping algorithm that applies compositional lumping and model-level lumping techniques at the same time could give an extra opportunity to shrink the CTMC. Therefore, a fairly general algorithm that integrates the two techniques for a compositional formalism is desirable.

Another approach classified under largeness avoidance is called *aggregation*[2]. In this approach, as in lumping, a set of conditions for partitioning the set of states of a CTMC is given such that a smaller CTMC is constructed by replacing the sets of states in each block

[2]Different authors use the terms *lumping* and *aggregation* differently. In this dissertation, *lumping* is used in the same sense as in [62] and *aggregation* is used as a general term for all other techniques for "fusing" a set of states.

of the partition with a single state. Aggregation differs from lumping in that the solution of the aggregated CTMC gives approximate results (with or without bounds) on the original CTMC, as opposed to the exact results that would be obtained from the lumped CTMC. Moreover, some aggregation techniques are applicable only to CTMCs that satisfy a strict set of conditions. However, aggregation conditions could result in a coarser partition, and therefore a smaller CTMC, compared to the lumping conditions. An aggregation technique for steady-state analysis of a general CTMC has been proposed in [32, 33]. It gives the best known bounds on the result but is computationally costly, and the bounds are tight only if the matrix satisfies some strict constraints. Daly et al. [34] give a more general aggregation technique that can solve for both steady-state and transient measures of general CTMCs. They introduced a new partial order on the set of states of a CTMC that is a generalization of the concept of lumping. However, in terms of time, the partial order computation algorithm is still much more expensive than optimal lumping computation. Further work on the algorithm to improve its running time is necessary.

### 1.2.2 Largeness Tolerance

Even a lumped CTMC can be extremely large, and further work on *largeness tolerance* techniques is needed to practically support large CTMCs. In those techniques, new algorithms are designed to manipulate large CTMCs while special data structures and/or representations are utilized to reduce the space requirements of the state space, the state transition matrix, and the iteration vectors. Those techniques are usually, but not always, associated with compositional modeling.

Binary [10] and multi-valued decision diagram [86] (BDD and MDD) data structures have been successfully applied to efficiently explore and represent large unlumped state spaces. The key idea is to encode states as paths in a directed acyclic graph. Techniques that generate state spaces using decision diagrams are referred to as *symbolic* state-space exploration and representation techniques (e.g., [23, 30]).

MDD data structures have been used in [29] to explore large state spaces of models built using an *action synchronization* (a.k.a. *action sharing*) high-level compositional formalism in which submodels interact by synchronized firing of a subset of their actions. Saturation, a state-space exploration technique that was introduced in [25], improved the running time of the algorithm given in [29] by up to a few orders of magnitude, thus enabling the exploration of even larger state spaces. In [25, 29], it was assumed that the state spaces of individual submodels were known a priori, i.e., the state spaces of the submodels computed by exploring the submodels in isolation are finite and are the same as the state spaces that would result if the submodels were explored in interaction with the rest of the model. This assumption was relaxed later in [26]. Those state-space exploration techniques are applicable to action-synchronization composed models that conform to a particular set of structural restrictions called the *logical product form* property [29].

One approach in space-efficient representation of state transition matrices is to follow a divide-and-conquer strategy and represent the matrix with a set of relatively small component matrices that are appropriately combined. The earliest attempt using that approach was made by Plateau [77, 78], who proposed a technique in which the transition matrix of a CTMC generated from a specific compositional high-level formalism need not be explicitly stored. Instead, the matrix is implicitly represented as a mathematical expression consisting of *Kronecker* operators and a number of relatively small matrices derived from the structure of submodels. Later, the "Kronecker representation" technique was extended to more general formalisms, and a number of its shortcomings were resolved [11, 20, 21, 44, 45, 63, 87]. The approach is also applicable only to action-synchronization models that satisfy certain structural constraints.

Another successful attempt was made by Miner and Ciardo [28], who proposed the matrix diagram (MD) data structure to store the state transition matrix of action-synchronization composed models. An MD is structurally similar to an MDD and, along with an MDD, represents the set of states and transitions of a very large CTMC. Efficient algorithms to

manipulate MDDs and MDs have been given in [28]. One of the advantages of using MDs and MDDs instead of Kronecker operators to represent large CTMCs is that the problem of potential state space is solved without the logarithmic overhead generally incurred by Kronecker representation techniques. In [28], the algorithm that generates the MD data structure is time-efficient, but works only for composed models that hold the logical product form property. Later, Miner [73] developed canonical MDs (CMDs), a proper subset of MDs, and presented an algorithm to store virtually any matrix in the form of a CMD. In particular, he used the algorithm to generate the CMD representation of the transition matrix of models based on the generalized stochastic Petri net (GSPN) formalism, which is a fairly general Markov modeling formalism. Since the algorithm added matrix elements to the CMD data structure one by one, and without exploiting any structural information, its running time was prohibitive. As mentioned above, both CMDs and MDs can represent virtually any transition matrix regardless of the modeling formalism from which it is generated. The challenge is then to develop algorithms that build (C)MD representations of transition matrices of other formalisms in a time-efficient manner. See [71] for a recent overview paper on symbolic representations.

The *disk-based* approach of [39] performs steady-state solution by storing the state transition rate matrix of the CTMC in the disk instead of the memory while using a variant of block Gauss-Seidel as the iterative solution algorithm. To increase the utilization of the CPU, the algorithm implementation concurrently fetches parts of the matrix from the disk and performs computation on other in-memory parts of the matrix. By using disk instead of memory to store the matrix, the technique enables the solution of CTMCs that are one to two orders of magnitude larger than would be possible using only memory.

The "on-the-fly" technique of [40] completely avoids the storage of the transition matrix by (re)generating the elements of the matrix as they are needed in an iterative solution algorithm (steady state or transient solution). The elements are computed on-the-fly from the model, which is given in a high-level formalism. Repetitive calculations of the elements

incur a substantial computational overhead.

The path-based approach is yet another largeness tolerance technique for performing transient analysis of CTMCs while avoiding the storage of the CTMC and possibly the iteration vector. In this approach, a limited number of paths (i.e., sequences of transitions) of the CTMC that make a major contribution toward the measures of interest are enumerated. Then, the reward is computed only for those paths. The first notable work based on that approach was given by de Souza e Silva and Gail [36], and they later improved it in [37]. Later, Qureshi improved the numerical stability and computational complexity of [37] and removed its requirement of generating the complete state space in [79]. Lam et al. [65] use Kronecker operators to represent both the CTMC and the iteration vector to compute approximate results for transient analysis of an action-synchronization composed model. They also compute a bound on the reward using the information about the amount of contribution of those paths to the reward [65]. Given enough computational resources, their approach can theoretically tighten the bounds as much as desired.

### 1.2.3 Combination of Techniques

For each of the techniques we described so far to be applicable to a specific model, there are always some requirements and restrictions that limit the applicability of the technique. These factors may include, for example, the time and space complexities of the technique, the formalism to which the technique is applicable, or possibly a set of additional restrictions on the formalism such as a special structure that the model should have or a special property that the model should hold. Such requirements mean that each technique can exploit only a limited subset of properties of a given model to make it more manageable to solve. We can observe a rough analogy between the techniques and tools in a toolbox. Each tool in a toolbox has its own strength, but it does not make other tools in the toolbox unnecessary because it also has its own limitations.

In the ideal world, we would like to be able to combine all the techniques on any given

model. However, because of the very specific limitations of each technique, that is not possible. A more realistic approach is to study the largeness tolerance and largeness avoidance techniques and study ways we can combine a subset of them. We can hope that that will lead to new techniques that have fewer requirements and/or restrictions such that they enable us to solve models that were impossible to solve before.

Quite a few researchers have successfully tried that approach. For example, compositional lumping is combined with Kronecker representation for stochastic automata networks (SANs) in [14, 18], for Markovian process algebra in [13], for hierarchical Markovian models in [15], and for hierarchical Kronecker models in [17]. Recently, a combination of (model-level) symmetry detection and Kronecker representation for the SAN formalism has been studied in [8], where, as in [47, 53, 82], only structural symmetries that involve replicas of a component are exploited. In [41], the symmetry detection technique of the SWN formalism has incorporated MDD data structures and Kronecker operators to represent the set of reachable states and the transition rate matrix, respectively.

We believe that the next breakthrough in the realm of largeness avoidance and largeness tolerance techniques will be the development of a tool that, given a model, automatically picks a number of techniques in the "toolbox" based on the properties of the model and also the strengths and limitations of each technique. Then, it will apply those selected techniques in the right order. In order to develop such a sophisticated tool, we at least need to formalize, as precisely and thoroughly as possible, the requirements and the restrictions of all available techniques and develop tools that automatically extract properties of a given model.

## 1.3 Our Contribution

The focus of this dissertation is to extend, improve, and combine existing solutions of the state-space explosion problem in order to make possible the construction and solution of very large CTMCs generated from high-level models. In Section 1.2, we described how numer-

ous techniques based on largeness avoidance and largeness tolerance approaches have been applied to solve very large CTMCs. Many of those primitive techniques were "orthogonal," meaning that they could be applied simultaneously to solve a single problem. In some of the past work, the orthogonality was not exploited to increase the size of analyzable CTMCs. In some of the other work, a few of the techniques were used at the same time, but the target high-level formalism was fairly restricted in structure. We believe that there is a need to design algorithms that incorporate as many primitive orthogonal techniques as possible and are applicable to as many and as general high-level formalisms as possible. In particular, we have:

- Demonstrated that the claim in the literature for the optimal CTMC (state-level) lumping algorithm with a running time $\mathcal{O}(m \log n)$ [9, 54, 58] is not valid to the best of our knowledge, and that the correct running time of those algorithms is in fact $\mathcal{O}(m \log^2 n)$.

- Presented and implemented a new optimal CTMC (state-level) lumping algorithm that improves upon previous work and is so far the fastest known, with a proven worst-case running time of $\mathcal{O}(m \log n)$. The algorithm has far-reaching implications in that it is also the fastest known algorithm for computing the coarsest lumping in DTMCs (discrete-time Markov chains), the coarsest bisimulation in probabilistic models such as WSCCS (Weighted Synchronous Calculus of Communicating Systems) [88, 89] and PCCS (Probabilistic CCS) [5], and Markovian models such as PEPA (Performance Evaluation Process Algebra) [55] and TIPP (Timed Processes and Performance Evaluation), and the coarsest Larsen/Skou-style bisimilarity [66] on general weighted automata.

- Proved a lower bound of $\Omega(m + n \lg n)$ for the state-level lumping problem. Although that is not a very tight bound compared to the time complexity of our newly developed algorithm, it gives us a sense of how difficult the problem is.

- Investigated, using experiments, the performance differences between the two variants of our state-level lumping algorithm and the relationship between the running time of the algorithm, its input CTMC, and its output (lumped) CTMC.

- Extended the previous work [28, 29, 72] on MDD and MD symbolic data structures to state-sharing composed models that do not have some of the structural restrictions required by previous approaches.

- Integrated, for the first time, the use of symbolic data structures with the model-level lumping technique of [82], and developed a symbolic state-space exploration algorithm for state-sharing replicate/join composed models. In particular, the algorithm generates the MDD representation of the lumped state space and the MD representation of the state transition rate matrix of a replicate/join composed model. We formally proved that our symbolic model-level lumping algorithm is correct.

- Devised an algorithm and a symbolic data structure, which we call the "sorting MDD," for efficient enumeration of the MD data structure mentioned above.

- Presented a new compositional lumping algorithm that is useful for exact and ordinary lumping of Markov chains represented as MDs. The advantage of our algorithm over other formalism-specific ones is that our formulation is simpler and easier to understand. More importantly, it is applicable to an MD regardless of the formalism of the model from which the MD has been generated. More specifically, it works for any model formalism for which there is a state-space generation algorithm that generates an MD or Kronecker[3] representation of the underlying CTMC's state-transition rate matrix [14, 28, 72, 73].

- Finished the implementation of the compositional lumping algorithm that uses symbolic data structures to represent CTMCs.

[3]Since any Kronecker expression can also be represented as an MD.

- Integrated our implementation of the MD-based compositional lumping algorithm with the implementation of our other algorithm that combines symbolic representation and model-level lumping. Given a model based on sharing state variables, the resulting implementation automatically and simultaneously applies the three techniques to reduce the state-space size and also the representation of the CTMC that needs to be solved.

To understand the contribution of this dissertation, it is crucial to note that, with some minor restrictions, all the techniques we will describe throughout this dissertation are orthogonal. More specifically, we can apply our compositional and model-level lumping techniques simultaneously on a compositional model formalism and always keep the representations symbolic. The result is that we will be able to exploit two types of lumping for a composed model: 1) lumping due to the symmetries present among the various components of the composed model induced by the replicate operator, and 2) lumping present in each individual component. That will enable us to generate potentially smaller MD representations of CTMCs than would be possible using either of the techniques individually. Moreover, the combination is again orthogonal to our state-level lumping algorithm in the sense that any partially lumped CTMC that is computed by the model-level and compositional lumping algorithms can still be fed to the state-level algorithm to obtain the smallest possible lumped CTMC.

## 1.4 Outline

As we mentioned earlier, the focus of this dissertation is extending, improving, and combining existing solutions of the state-space explosion problem in order to make possible the construction and solution of very large CTMCs generated from high-level models.

In Chapter 2, we present our notation, definitions, and properties of various concepts that we are going to need in the rest of the dissertation, including CTMCs, MRPs (Markov Reward Processes), two types of lumping on MRPs (namely, ordinary and exact lumping), and

three symbolic data structures, namely MDDs (Multi-valued Decision Diagrams), MTMDDs (Multi-Terminal MDDs), and MDs (Matrix Diagrams).

In Chapter 3, we describe our new optimal state-level CTMC lumping algorithm and prove that its worst-case running time complexity is $\mathcal{O}(m \log n)$. We show how the use of splay trees [85] for transition rate sorting enables us to improve upon existing algorithms if they use general balanced trees. Our proposed algorithm can be used in isolation or in any compositional lumping algorithm, such as the one described in Chapter 5.

In Chapter 4, we extend the previous work on MDD and MD symbolic data structures to state-sharing composed models. We also combine the application of model-level lumping properties of state-sharing replicate/join composed models with symbolic data structures. In particular, we present a symbolic state-space exploration algorithm that builds MDD and MD representations of the state space and the state transition rate matrix of the lumped CTMC, respectively. We also describe a technique for efficient enumeration of the MD data structure for the purpose of numerical analysis.

In Chapter 5, we present a new compositional lumping algorithm that computes the exact and ordinary lumping of Markov chains represented as MDs without knowledge of the modeling formalism from which the MDs were generated. Our approach relies on local conditions, i.e., conditions on individual levels of the MD. Since our algorithm locally processes MD nodes which often have dramatically smaller sizes than the matrix represented by the MD, it is computationally inexpensive. It uses the algorithm developed in Chapter 3 to compute, for each level of the MD, a lumpable partition that satisfies the local conditions.

We conclude in Chapter 6 and point out the main contributions of this dissertation and how they fulfill the goals of the ongoing research on the state-space explosion problem. A number of ways to extend the work of this dissertation are also discussed.

# Chapter 2

# Background

In this chapter, we will present our notation, definitions, and properties of various concepts that we are going to need in the rest of the dissertation, starting with CTMCs and followed by MRPs. Then, we define the two types of lumping on MRPs that are the focus of our dissertation, namely ordinary and exact lumpability. Finally, we describe the MDD (Multi-valued Decision Diagrams) and MD (Matrix Diagram) data structures. These symbolic data structures are used to compactly represent sets of vectors (or sets of states) and matrices respectively.

We assume that the reader is already familiar with the following concepts: CTMC, state space, transition rate matrix, probability distribution vector, steady-state and transient analysis, steady-state and transient probability vectors, and state-based measures (or rate rewards). For detailed coverage of those subjects, see, for example, [87, 90].

We use the following notations in the rest of the dissertation. All matrices and vectors are typeset with bold characters (upper-case letters for matrices and lower-case letters for vectors), and their rows and columns are indexed starting from 1. The element of matrix $\mathbf{A}^{(n\times m)}$ (of size $n \times m$) in row $i$ and column $j$ is referred to as $\mathbf{A}(i,j)$. For a matrix $\mathbf{A}^{(n\times m)}$, $rs(\mathbf{A}) = \mathbf{B}^{(n\times n)}$ is a diagonal matrix ($rs$ stands for row sum) such that $\mathbf{B}(i,i) = \sum_{1\leq j\leq m} \mathbf{A}(i,j)$; $rs$ stands for "row sum". Finally, all sets are assumed to be finite.

## 2.1 CTMCs and MRPs

We specify a CTMC by a state space $\mathcal{S} = \{1, \ldots, |\mathcal{S}|\}$ and a state transition rate matrix $\mathbf{R}$, or a generator matrix $\mathbf{Q} = \mathbf{R} - rs(\mathbf{R})$, in which $\mathbf{R}(i,j)$ is the (positive) rate of the transition that changes the state of the CTMC from $i$ to $j$. By definition, we have $\mathbf{Q}(i,i) = -\sum_{1 \le j \le |\mathcal{S}|} \mathbf{R}(i,j)$ and $\sum_{1 \le j \le |\mathcal{S}|} \mathbf{Q}(i,j) = 0$. Let $n = |\mathcal{S}|$ denote the number of states of the CTMC. The number of transitions of a CTMC is the number of non-zero entries of $\mathbf{R}$.

Consider a partition $\mathcal{P} = \{C_1, \ldots, C_{\tilde{n}}\}$ of the state space $\mathcal{S}$. $C_1, \cdots, C_{\tilde{n}}$ are the equivalence classes of partition $\mathcal{P}$, or in short, *classes* of $\mathcal{P}$. Since the elements of the classes of a partition $\mathcal{P}$ are actually states of the CTMC, we will sometimes use the words *state* and *element* interchangeably when the meaning is clear from the context. Let $[x]_{\mathcal{P}}$ denote the (unique) class index $i$ such that $x \in C_i$. Any two states $x, \hat{x}$ in a class $C_i$ of $\mathcal{P}$ are called *equivalent*, and that is shown by $x \approx_{\mathcal{P}} \hat{x}$, or simply $x \approx \hat{x}$ if $\mathcal{P}$ is obvious from the context. Partition $\mathcal{P}'$ is a *refinement* of partition $\mathcal{P}$ (or *finer* than $\mathcal{P}$) if every class of $\mathcal{P}'$ is a subset of some class of $\mathcal{P}$. $\mathcal{P}$ is said to be *coarser* than $\mathcal{P}'$. Note that if $\mathcal{P}'$ is finer than $\mathcal{P}$, we can refine $\mathcal{P}$ to $\mathcal{P}'$ by partitioning some of $\mathcal{P}$'s classes into smaller classes.

Given a CTMC, a partition $\mathcal{P}$ on its state space induces a quotient stochastic process with regard to $\mathcal{P}$, but not all partitions result in a process with the Markov property. In the next section, we will see what sets of conditions on a partition can result in the quotient CTMC being a Markov process.

Often, the final goal of a CTMC analysis is not the computation of the steady-state or transient probability of its states. Instead, it is the computation of high-level measures such as performance, dependability, and/or availability. Many of those high-level measures can be computed using reward values associated with each state of the CTMC (i.e., rate rewards) and the stationary and transient probability vectors [57]. By incorporating rate rewards and an initial probability vector with the CTMC, we will have an MRP for which stationary and transient measures can be computed.

**Definition 2.1** *A Markov reward process (MRP) $M$ is a 4-tuple $(\mathcal{S}, \mathbf{Q}, \mathbf{r}, \boldsymbol{\pi}^{\text{ini}})$ where*

- $\mathcal{S} = \{1, \ldots, |\mathcal{S}|\}$ *is the state space of a CTMC,*
- $\mathbf{Q}^{(|\mathcal{S}|\times|\mathcal{S}|)}$ *is the generator matrix of the CTMC,*
- $\mathbf{r}^{(|\mathcal{S}|)}$ *is a row vector of size $|\mathcal{S}|$ that assigns reward value $\mathbf{r}(i) \in \mathbb{R}$ to state $i$, and*
- $\boldsymbol{\pi}^{\text{ini}}(i)$ *is the probability that the CTMC is in state $i$ at time 0 $(1 \leq i \leq |\mathcal{S}|)$.*

■

Given an MRP and a number of high-level measures expressed in terms of $\mathbf{r}$ and $\boldsymbol{\pi}^{\text{ini}}$, we can compute the measures by numerically solving the underlying CTMC. The larger the state space of the CTMC, the more time-consuming that numerical solution will be. Sometimes, as described in the introduction, a lumped CTMC can be used to obtain the desired measures. For that to be possible, the MRP that is based on that CTMC should satisfy a set of conditions. Three of the most important sets of conditions (and the types of lumpings they lead to) are outlined in Definition 2.2. The lumped MRP, which includes the lumped CTMC, is obtained using Theorem 2.2. Notice that each set of conditions may result in a different lumped MRP.

## 2.2 Ordinary and Exact Lumpability

For a compact notation, we use the following identities:

$$\mathbf{A}(i, C) = \sum_{j \in C} \mathbf{A}(i, j) \qquad \mathbf{A}(C, j) = \sum_{i \in C} \mathbf{A}(i, j) \qquad \mathbf{a}(C) = \sum_{i \in C} \mathbf{a}(i)$$

where $\mathbf{A}$ is a matrix, $\mathbf{a}$ is a vector, and $C$ is a set of valid column or row indices.

**Definition 2.2** *Consider an MRP $M = (\mathcal{S}, \mathbf{Q}, \mathbf{r}, \boldsymbol{\pi}^{\text{ini}})$ and a partition $\mathcal{P}$ of $\mathcal{S}$. Then, with respect to $\mathcal{P}$, $M$ is*

*a. ordinarily lumpable if and only if for all $C, C' \in \mathcal{P}$ and all (equivalent states) $s, \hat{s} \in C$,*

$$\mathbf{r}(s) = \mathbf{r}(\hat{s}) \qquad \textit{and} \tag{2.1}$$

$$\mathbf{Q}(s, C') = \mathbf{Q}(\hat{s}, C') \tag{2.2}$$

*b. exactly lumpable if and only if for all $C, C' \in \mathcal{P}$ and all (equivalent states) $s, \hat{s} \in C$,*

$$\boldsymbol{\pi}^{\text{ini}}(s) = \boldsymbol{\pi}^{\text{ini}}(\hat{s}) \qquad \textit{and} \tag{2.3}$$

$$\mathbf{Q}(C', s) = \mathbf{Q}(C', \hat{s}) \tag{2.4}$$

*c. strictly lumpable if $M$ is ordinarily and exactly lumpable with respect to $\mathcal{P}$.* ■

Strictly speaking, Definition 2.2 defines lumping for an MRP and not for the CTMC embedded in it. However, in the rest of the dissertation, we will also speak of lumping of CTMCs or of their corresponding $\mathbf{Q}$ or $\mathbf{R}$ matrices, under the assumption that the reward and initial probability vectors are such that they satisfy the requirements of Definition 2.2.

Definition 2.2 gives lumpability conditions in terms of the $\mathbf{Q}$ matrix of a CTMC. We will use it in Chapter 3 in order to compute the coarsest lumpable partition of an MRP. However, in Chapter 5, when we discuss compositional lumping, we will need to check the lumpability conditions and compute lumpable partitions in terms of $\mathbf{R}$ instead of $\mathbf{Q}$. In Theorem 2.1, we specify conditions that are both necessary and sufficient for ordinary lumpability of an MRP in terms of $\mathbf{R}$. However, the conditions given for exact lumpability are only sufficient.

**Theorem 2.1** *Consider an MRP $M = (\mathcal{S}, \mathbf{Q}, \mathbf{r}, \boldsymbol{\pi}^{\text{ini}})$ such that $\mathbf{Q} = \mathbf{R} - rs(\mathbf{R})$, where $\mathbf{R}$ is the state transition rate matrix of the CTMC of $M$. Then, with respect to a partition $\mathcal{P}$, $M$ is*

*a. ordinarily lumpable if and only if (1) $\mathbf{r}(s) = \mathbf{r}(\hat{s})$ and (2) $\mathbf{R}(s, C') = \mathbf{R}(\hat{s}, C')$ for all $C, C' \in \mathcal{P}$ $(C \neq C')$ and all (equivalent states) $s, \hat{s} \in C$.*

*b. exactly lumpable if (1)* $\mathbf{R}(s,\mathcal{S}) = \mathbf{R}(\hat{s},\mathcal{S})$ *and* $\boldsymbol{\pi}^{\text{ini}}(s) = \boldsymbol{\pi}^{\text{ini}}(\hat{s})$ *and (2)* $\mathbf{R}(C',s) = \mathbf{R}(C',\hat{s})$ *for all* $C, C' \in \mathcal{P}$ *and all (equivalent states)* $s, \hat{s} \in C$.

**Proof.** (a) *If*: We need to prove that $\mathbf{Q}(s,C') = \mathbf{Q}(\hat{s},C')$ for all $C, C' \in \mathcal{P}$ and all $s, \hat{s} \in C$. If $C = C'$, since the sum of elements in any row of any generator matrix $\mathbf{Q}$ is zero, we have:

$$\begin{aligned}
\mathbf{Q}(s,C') = -\mathbf{Q}(s,\mathcal{S}\backslash C') &= -\sum_{\substack{C''\in\mathcal{P}\\ C''\neq C'}} \mathbf{Q}(s,C'') \\
[\text{since } s \notin C''] &= -\sum_{\substack{C''\in\mathcal{P}\\ C''\neq C'}} \mathbf{R}(s,C'') \\
[\text{since } C'' \neq C'] &= -\sum_{\substack{C''\in\mathcal{P}\\ C''\neq C'}} \mathbf{R}(\hat{s},C'') \\
[\text{since } \hat{s} \notin C''] &= -\sum_{\substack{C''\in\mathcal{P}\\ C''\neq C'}} \mathbf{Q}(\hat{s},C'') \\
&= -\mathbf{Q}(\hat{s},\mathcal{S}\backslash C') = \mathbf{Q}(\hat{s},C')
\end{aligned}$$

However, if $C \neq C'$, then $s, \hat{s} \notin C'$. Therefore, we have $\mathbf{Q}(s,C') = \mathbf{R}(s,C') = \mathbf{R}(\hat{s},C') = \mathbf{Q}(\hat{s},C')$.

*Only if*: We need to prove that $\mathbf{R}(s,C') = \mathbf{R}(\hat{s},C')$ for all $C \neq C' \in \mathcal{P}$ and all $s, \hat{s} \in C$. Since $C \neq C'$, then $s, s' \notin C'$, and therefore, $\mathbf{R}(s,C') = \mathbf{Q}(s,C') = \mathbf{Q}(\hat{s},C') = \mathbf{R}(\hat{s},C')$.

(b) If $C \neq C'$, then $s, \hat{s} \notin C'$, and therefore, $\mathbf{Q}(C',s) = \mathbf{R}(C',s) = \mathbf{R}(C',\hat{s}) = \mathbf{Q}(C',\hat{s})$. Otherwise, $\mathbf{Q}(C',s) = \mathbf{R}(C',s) - \mathbf{R}(s,\mathcal{S}) = \mathbf{R}(C',\hat{s}) - \mathbf{R}(\hat{s},\mathcal{S}) = \mathbf{Q}(C',\hat{s})$ since $\mathbf{R}(C',s) = \mathbf{R}(C',\hat{s})$ and $\mathbf{R}(s,\mathcal{S}) = \mathbf{R}(\hat{s},\mathcal{S})$. ■

Note that condition $(a2)$ above does not need to hold for $C = C'$. Moreover, condition $\mathbf{R}(s,\mathcal{S}) = \mathbf{R}(\hat{s},\mathcal{S})$ in $(b1)$ means that the total outgoing rates of equivalent states $s$ and $\hat{s}$ (or sums of all elements in rows $s$ and $\hat{s}$) are equal. This condition was not required by $(a)$. Finally, the converse of Theorem 2.1$(b)$ does not hold.

Now that we know when a partition satisfies the ordinary or exact lumpability conditions, we need to know how to derive a lumped MRP, given the original MRP and a lumpable

partition $\mathcal{P}$, that gives us the same results for the reward measures as the original MRP does.

**Theorem 2.2** *Let $M = (\mathcal{S}, \mathbf{Q}, \mathbf{r}, \boldsymbol{\pi}^{\text{ini}})$ be ordinarily or exactly lumpable with respect to a partition $\mathcal{P} = \{C_1, \ldots, C_{\widetilde{n}}\}$ of $\mathcal{S}$. Then $\widetilde{M} = (\widetilde{\mathcal{S}}, \widetilde{\mathbf{Q}}, \widetilde{\mathbf{r}}, \widetilde{\boldsymbol{\pi}}^{\text{ini}})$ is the lumped MRP such that*

$$\widetilde{\mathbf{Q}}(\tilde{i}, \tilde{j}) = \begin{cases} \mathbf{Q}(i, C_{\tilde{j}}) & \textit{for arbitrary } i \in C_{\tilde{i}} \quad \textit{(ordinary)} \\ \mathbf{Q}(C_{\tilde{i}}, j) & \textit{for arbitrary } j \in C_{\tilde{j}} \quad \textit{(exact)} \end{cases}$$

$$\widetilde{\mathcal{S}} = \{1, \ldots, \widetilde{n}\} \qquad \widetilde{\mathbf{r}}(\tilde{i}) = \mathbf{r}(C_{\tilde{i}})/|C_{\tilde{i}}| \qquad \widetilde{\boldsymbol{\pi}}^{\text{ini}}(\tilde{i}) = \boldsymbol{\pi}^{\text{ini}}(C_{\tilde{i}})$$

**Proof.** See [12]. ■

Using Theorem 2.2, one can show that all reward measures of $M$ that are based on $\mathbf{r}$ and $\boldsymbol{\pi}^{\text{ini}}$ can be computed using $\widetilde{\mathbf{r}}$ and $\widetilde{\boldsymbol{\pi}}^{\text{ini}}$. In [12], there are also theorems on how the state probabilities of $M$ are related to those of $\widetilde{M}$. The following proposition is the counterpart of Theorem 2.2 in terms of matrix $\mathbf{R}$ instead of $\mathbf{Q}$.

**Theorem 2.3** *Given the assumptions of Theorem 2.2, we have:*

$$\widetilde{\mathbf{R}}(\tilde{i}, \tilde{j}) = \begin{cases} \mathbf{R}(i, C_{\tilde{j}}) & \textit{for arbitrary } i \in C_{\tilde{i}} & \textit{(ordinary)} & \textit{if } \tilde{i} \neq \tilde{j} \\ \mathbf{R}(C_{\tilde{i}}, j) & \textit{for arbitrary } j \in C_{\tilde{j}} & \textit{(exact)} & \textit{if } \tilde{i} \neq \tilde{j} \\ 0 & \textit{by definition} & \textit{(both)} & \textit{if } \tilde{i} = \tilde{j} \end{cases}$$

$$\widetilde{\mathcal{S}} = \{1, \ldots, \widetilde{n}\} \qquad \widetilde{\mathbf{r}}(\tilde{i}) = \mathbf{r}(C_{\tilde{i}})/|C_{\tilde{i}}| \qquad \widetilde{\boldsymbol{\pi}}^{\text{ini}}(\tilde{i}) = \boldsymbol{\pi}^{\text{ini}}(C_{\tilde{i}})$$

**Proof.** (a) Ordinary lumping: If $\tilde{i} \neq \tilde{j}$, then $C_{\tilde{i}} \cap C_{\tilde{j}} = \emptyset$. Therefore, for an arbitrary $i \in C_{\tilde{i}}$, we have:

$$\widetilde{\mathbf{R}}(\tilde{i}, \tilde{j}) = \widetilde{\mathbf{Q}}(\tilde{i}, \tilde{j}) = \mathbf{Q}(i, C_{\tilde{j}}) = \sum_{j \in C_{\tilde{j}}} \mathbf{Q}(i, j) = \sum_{j \in C_{\tilde{j}}} \mathbf{R}(i, j) = \mathbf{R}(i, C_{\tilde{j}}).$$

(b) Exact lumping: Similar to (a). ■

## 2.3 Symbolic Data Structures

The goal of this dissertation is to devise new techniques that enable the analysis of large Markovian models. These techniques, as we discussed earlier in Chapter 1, can be classified as largeness avoidance and largeness tolerance techniques. Largeness tolerance techniques include the use of symbolic data structures (e.g., [23, 30]) that give a compact representation for the state space, the transition rate (or generator) matrix, and the reward vector of an MRP. In the following, we give an overview of the three data structures, MDDs (Multi-valued Decision Diagrams) [86], MTMDDs (Multi-Terminal MDDs), and MDs (Matrix Diagrams) [28], that we are going to use to represent sets of states, reward vectors, and transition rate matrices, respectively. For more information about these data structures and algorithms to manipulate them, see [29], [73], and [72].

### 2.3.1 (Multi-Terminal) Multi-valued Decision Diagrams

MTMDDs are a generalization of BDDs (Binary Decision Diagrams) [10]. They represent functions of the form $f : \times_{c=1}^{m} \mathcal{S}_c \rightarrow T$ in which the sets $\mathcal{S}_c = \{0, \ldots, |\mathcal{S}_c| - 1\}$ and $T \subset \mathbb{R}$ are finite and $c \in \{1, \ldots, m\}$. MTMDDs are ordered, i.e., the order of $\mathcal{S}_c$'s is fixed; we consider the order $\mathcal{S}_1, \ldots, \mathcal{S}_m$.

In terms of data structure representation, an (MT)MDD is a rooted directed acyclic graph with $|T|$ *terminal* nodes and one or more *non-terminal* nodes. Each of the $|T|$ terminal nodes has an associated value taken from $T$. The nodes are called *terminal* because they do not point to any other nodes. Each non-terminal node is associated with a variable $x_c \in \mathcal{S}_c$ and is a vector of $|\mathcal{S}_c|$ pointers, each of which points to a node with a variable $x_{c'}$ $(c' > c)$ or to a terminal node. All nodes associated with variable $x_c$ constitute *level* $c$ of the (MT)MDD, and any node in level $c$ of the (MT)MDD is called a *level-c node*.

The main advantage of (MT)MDDs is that a reduction operation is used to represent isomorphic subgraphs only once. That reduction is based on a notion of equality and redundancy for nodes; two terminal nodes are *equal* if they have the same value and two non-terminal nodes are equal if they are in the same level and all their corresponding pointers point to equal nodes. A non-terminal node is *redundant* if all of its pointers point to the same node. We follow [28] and consider ordered (MT)MDDs, for which equal nodes have been merged and redundant nodes are retained only to ensure that a pointer of a level-$c$ node can lead only to a level-$(c+1)$ node or to a terminal node. The value of function $f(s_1, \ldots, s_m)$, where $s_c \in \mathcal{S}_c$ for all $c \in \{1, \ldots, m\}$, is derived by following a path in an (MT)MDD graph starting at the root node and ending, after at most $m$ edges (node connections), at a terminal node that represents the resulting value in $T$. At each non-terminal level-$c$ node, a successor node is selected according to $s_c$.

The case where $T = \{0, 1\}$ is a special one in which the data structure is called an MDD (Multi-valued Decision Diagram). The function $f$ could be used to encode a set of tuples $\mathcal{S} \subseteq \times_{c=1}^{m} \mathcal{S}_c$; a tuple $(s_1, \ldots, s_m)$, in which $s_c \in \mathcal{S}_c$, is an element of $\mathcal{S}$ if and only if $f(s_1, \ldots, s_m) = 1$. Typical set operations like union, intersection, and difference can be performed on MDDs efficiently, as described in [29].

We will use MDDs throughout the dissertation to represent a finite set of states of a discrete-state Markovian model that (directly or indirectly) uses state variables to represent its state. The widely used idea is that we partition the set of state variables into $m$ classes. For each class $C_c (1 \leq c \leq m)$ of the partition, we enumerate all the possible valuations of the state variables in $C_c$ and index them from 0 to $|\mathcal{S}_c| - 1$. With such a configuration, a state of the model can be uniquely represented as a tuple $(s_1, \ldots, s_m)$. Thus, as explained above, we can use the MDD data structure to represent an arbitrary (finite) set of states of the model.

(MT)MDDs are often enhanced by a so-called *offset function* $\rho : \mathcal{S} \rightarrow \{0, 1, \ldots, |\mathcal{S}| - 1\}$, in which the elements of $\mathcal{S}$ are given indices based on the (total) order induced by the $\leq_{lex}$

(lexicographical order) operator. For two elements $s = (s_1, \ldots, s_m)$ and $s' = (s'_1, \ldots, s'_m)$, $s$ is called *(lexicographically)* smaller than or equal to $s'$ and denoted by $s \leq_{lex} s'$ if and only if

$$s = s' \text{ or } (\exists\, i \in \{1, \ldots, m\} : s_1 = s'_1 \text{ and } \cdots \text{ and } s_{i-1} = s'_{i-1} \text{ and } s_i < s'_i)$$

$\rho$ is encoded through assignment of an additional weight $\rho_c(s_1, \ldots, s_c)$ to each pointer of a level-$c$ node, and the offset of $s$ is the sum of weights along the corresponding path in the (MT)MDD, i.e., $\rho(s) = \sum_{c=1}^{m} \rho_c(s_1, \ldots, s_c)$.

### 2.3.2 Matrix Diagrams

An MD [28] is a symbolic data structure that represents real-valued matrices, or equivalently, functions of the form $f : \times_{c=1}^{m}(\mathcal{S}_c \times \mathcal{S}_c) \to \mathbb{R}$. MDs are related to MDDs, edge-valued decision diagrams [64], and Kronecker representations [78]. MDs have proven to be particularly useful in representing large state transition rate matrices of compositional Markovian models. Like (MT)MDDs, MDs are ordered, and we always consider the same order and the same $\mathcal{S}_c$ sets as in the (MT)MDDs. That makes the MDs and (MT)MDDs that our algorithms in later chapters generate compatible with one another. Our formal definition of MDs in the following is slightly different from that in [28, 72, 73]. The resulting data structures remain the same, but the formal treatment is more concise.

In terms of representation, an MD is a rooted directed acyclic graph with one or more *terminal* nodes and zero or more *non-terminal* nodes. Nodes of an MD are matrices, whereas nodes of (MT)MDDs are vectors. We assign a unique index to each node of an MD, and we refer to the node with index $n$ as $R_n$. $R_1$ is the root node. As in (MT)MDDs, each node (whether terminal or non-terminal) is associated with a variable $x_c$, all nodes associated with variable $x_c$ constitute *level* $c$ of the MD, and any node in level $c$ of the MD is called a *level-c node*. We use $N_c$ to denote the set of indices of level-$c$ nodes. The top level (level 1) has only the root node in it, i.e., $N_1 = \{1\}$. For $1 \leq c < m$, all (non-terminal) nodes in level

$c$ point to nodes in level $c+1$.

For level $c \neq m$, matrix entries are finite formal sums

$$R_{n_c}(s_c, s'_c) = \sum_{n_{c+1} \in N_{c+1}} r_{n_c,n_{c+1}}(s_c, s'_c) \cdot R_{n_{c+1}}$$

with real values $r_{n_c,n_{c+1}}(s_c, s'_c)$ and references to nodes at level $n_{c+1}$. Terms of the form $r_{n_c,n_{c+1}}(s_c, s'_c) \cdot R_{n_{c+1}}$ with $r_{n_c,n_{c+1}}(s_c, s'_c) \neq 0$ correspond to a pointer from element $(s_c, s'_c)$ of $R_{n_c}$ to $R_{n_{c+1}}$. At level $c = m$, $R_{n_c}(s_c, s'_c)$ is a real value, and therefore $R_{n_c}$ is a real-valued matrix.

We assume that the MD is *reduced*, i.e., at any level $c$, no two nodes are equal. Otherwise, one node can be removed, and references at level $c-1$ can be appropriately renamed. Two nodes $R_{n_c}$ and $R_{n'_c}$ in level $c \in \{1, \ldots, m-1\}$ are considered equal if for all $n_{c+1} \in N_{c+1}$ and all $s_c, s'_c \in \mathcal{S}_c$, $r_{n_c,n_{c+1}}(s_c, s'_c) = r_{n'_c,n_{c+1}}(s_c, s'_c)$. For level $c = m$, two nodes $R_{n_c}$ and $R_{n'_c}$ are equal if all their corresponding real-valued elements are equal. The assumption that the MD has been reduced is the basis of the space efficiency of the MD data structure. In MD theory, the removal of duplicates is called *quasi-reduction*.

Each MD node $R_{n_i}$ results in a real-valued matrix $\mathbf{R}_{n_i}$. By definition, that holds for nodes at level $c = m$ by definition. For level $c < m$, we merge adjacent levels from the bottom up. Observe that for $c < m$ and $n_c \in N_c$, $\mathbf{R}_{n_c}$ is a block structured matrix in which each block is defined as $\mathbf{R}_{n_c}(s_c, s'_c) = \sum_{n_{c+1} \in N_{c+1}} r_{n_c,n_{c+1}}(s_c, s'_c) \cdot \mathbf{R}_{n_{c+1}}$. In other words, we replace node references with matrices and resolve the formal sum by scalar matrix multiplication and matrix addition. We can thus merge levels $c$ through $m$ by replacing nodes $R_{n_c}$ with matrices $\mathbf{R}_{n_c}$ (for all $n_c \in N_c$) and removing all level $c+1$ to $m$ without affecting the matrix that the overall MD represents.

For $c = 1$, we reduce the number of levels to one and end up with the real-valued matrix $\mathbf{R}_1$, which corresponds with the root node $R_1$. $\mathbf{R}_1$ is a flat representation of the matrix the original MD represents. $\mathbf{R}_1$ has a nested block structure and its column and row indices are

each chosen from the set $\times_{c=1}^{m} \mathcal{S}_c$. Therefore, a row or a column index can be represented as $s = (s_1, \ldots, s_m)$. When an MD is used to represent the $\mathbf{R}$ or the $\mathbf{Q}$ matrix of a CTMC, $s = (s_1, \ldots, s_m)$ is a state of the CTMC, and we call $s_c$ a *substate* of $s$.

## 2.4 Summary

In this chapter, we defined the fundamental mathematical entities and introduced the notations that we will need throughout the rest of the dissertation. In particular, we presented our notation for CTMCs, state spaces, and partitions. To be able to compute reward measures from a CTMC, we augmented the CTMC with a rate (i.e., state-based) reward and an initial probability vector and called the package an MRP. Then we gave the conditions under which an MRP is ordinarily and/or exactly lumpable with respect to a given partition of its state space. We also mentioned how the lumped MRP can be computed given the original MRP and the lumpable partition. Lumping of states enables us to compute the desired reward measure by analyzing the potentially smaller lumped MRP instead of the original MRP; the coarser the partition, the smaller the lumped MRP and the faster the analysis will be. In the next chapter, we will present an efficient algorithm that computes the coarsest partition with respect to which a given MRP is ordinarily and/or exactly lumpable. Finally, we gave a brief overview of a few symbolic data structures, that is, MDDs, MTMDDs, and MDs. As our main largeness tolerance technique, those data structures will be used in Chapters 4 and 5 to represent large state spaces, reward vectors, and state-transition rate matrices.

# Chapter 3

# Optimal State-level Lumping of MRPs

In this chapter, we present an optimal algorithm for *state-level* ordinary and exact lumping of MRPs. By state-level algorithm, we mean an algorithm that operates directly, i.e., at the state level, on the generator or transition rate matrix of the MRP. The algorithm has worst-case time complexity of $\mathcal{O}(m \log n)$ and space complexity of $\mathcal{O}(m + n)$, where $n$ and $m$ are the number of states and non-zero entries of the generator matrix of the underlying CTMC, respectively. The algorithm is optimal in the sense that it computes the smallest possible lumped MRP, i.e., a lumped MRP with the smallest possible state space. That is equivalent to saying that it generates the coarsest possible lumpable partition, i.e., a lumpable partition with the smallest possible number of classes. As we discussed in Chapter 1, the algorithm can be used in isolation or in any compositional lumping algorithm, such as the one described in Chapter 5.

To the best of our knowledge, as of the writing of this dissertation, our algorithm is the fastest known (in terms of time complexity) algorithm for the lumping problem. At the other hand, there are no results on a tight lower bound for the time complexity of the problem, although we will later prove a loose lower bound of $\Omega(m + n \log n)$.

In the remainder of the chapter, we focus on MRPs, which are based on CTMCs. However, the definitions, algorithms, and analyses can be extended in a straightforward manner to a variety of related problems. In particular, our algorithm is the fastest known algorithm for computing the coarsest lumping in DTMCs (discrete-time Markov chains), the coarsest bisimulation in probabilistic models such as WSCCS (Weighted Synchronous Calculus of Communicating Systems) [88, 89] and PCCS (Probabilistic CCS) [5], and other Markovian

models such as PEPA (Performance Evaluation Process Algebra) [55] and TIPP (Timed Processes and Performance Evaluation), and the coarsest Larsen/Skou-style bisimilarity [66] on general weighted automata.

The algorithm is based on the Paige/Tarjan algorithm for computing bisimilarity in labeled transition systems [76], and relies on the use of splay trees [85] to sort transition weights. Its running time is an improvement over existing lumping algorithms (for instance, see [19]), with a running time of $\mathcal{O}(mn)$. The $\mathcal{O}(m \log n)$ result for Markov chains (continuous or discrete) is apparently known, since the result has been claimed by various authors [9, 54, 58]. None of their publications, however, contain a proof of the $\mathcal{O}(m \log n)$ time complexity, or a description of the algorithm that is sufficiently detailed to make the claim obviously true. In [19], Buchholz mentions that CTMC lumping requires checking for identical rates that cannot be represented with binary relations as in labeled transition systems. He concludes that the $\mathcal{O}(m \log n)$ algorithm for bisimilarity computation cannot be adopted (as it is) for the problem of CTMC lumping.

Although the state-level lumping algorithm that we have presented in [42] works only for ordinary lumping of CTMCs (and not MRPs), it works fundamentally the same way as the one we will give here. In this chapter, we have extended it in two ways: 1) to support exact lumping in addition to ordinary lumping, and 2) to work on MRPs as opposed to CTMCs. The latter means that reward values and initial probability vectors are also taken into consideration. Because of these extensions, the reader will notice slight changes in some notations and details.

After starting with some definitions and notations in Section 3.1, we describe the core of the lumping algorithm in Section 3.2. Then, in Section 3.3, we show that a naïve algorithm has a time complexity of $\mathcal{O}(m \log^2 n)$, and we do not know how to rigorously prove the claimed better time complexity for the naïve algorithm. We believe that such a proof would require a clever way to sort weights that has not yet been devised, to the best of our knowledge. By using statically optimal trees (e.g., splay trees) [85] for the sort operation,

we are able to attain the $\mathcal{O}(m \log n)$ worst-case time complexity. This new algorithm turns the complexity bottleneck of the optimal MRP lumping problem from time to space. We also prove a loose lower bound for the time complexity of the MRP lumping problem. In Section 3.4, we study the running time performance of implementations of the naïve algorithm, our new algorithm, and the algorithm in [19] by applying them on an MRP of a high-level Markovian model of a hypercube multiprocessor system. Finally, we will summarize in Section 3.5.

## 3.1 Background

We consider an MRP $M = (\mathcal{S}, \mathbf{Q}, \mathbf{r}, \boldsymbol{\pi}^{\text{ini}})$. We define a relation $\rightarrow \subset \mathcal{S} \times \mathcal{S}$ such that $i \rightarrow j$ iff $\mathbf{Q}(i, j) \neq 0$. State $i$ is called a *predecessor* of state $j$ and state $j$ is called a *successor* of state $i$. We use $m$ to denote the cardinality of relation $\rightarrow$, i.e., the number of non-zero entries of $\mathbf{Q}$, the generator matrix of the CTMC. We further define the forward and backward images $e, e^{-1} : \mathcal{S} \mapsto 2^{\mathcal{S}}$ of $\rightarrow$ by $e(i) = \{j |\ i \rightarrow j\}$, and $e^{-1}(i) = \{j |\ j \rightarrow i\}$.

The partitions of $\mathcal{S}$ satisfying the conditions in Definition 2.2 form a complete lattice, in which the trivial finest partition $\{\{i\} |\ i \in \mathcal{S}\}$ is the minimal element, and the coarsest partition, corresponding to the smallest (i.e., with the smallest number of states) possible lumped MRP, is the maximal element. This lattice is akin to the lattice induced on a labeled transition system by the notion of bisimulation [55, 70], where bisimilarity is the maximal element in the lattice of bisimulations. Like bisimilarity, the coarsest lumping partition can be characterized as a fixed point of successively finer partitions; this characterization is the basis of the algorithmic procedure we will discuss. For bisimilarity, Kanellakis and Smolka have given such a partition refinement algorithm with time complexity $\mathcal{O}(mn)$ [61]. They conjectured that an algorithm exists that reduces the time complexity to $\mathcal{O}(m \log n)$. A few years later Paige and Tarjan designed such an algorithm [76].

## 3.2 Algorithm Description

In this section, we describe our algorithm for coarsest ordinary and exact lumping. Like other algorithms that claim the $\mathcal{O}(m \log n)$ time complexity, our algorithm is based on Paige and Tarjan's partition refinement algorithm to compute bisimilarity of labeled transition systems. The proof of the correctness of the algorithm is not given here, but can be adapted from [51].

Based on the different conditions of ordinary and exact lumping (see Definition 2.2), one can tell that each type of lumping requires a separate algorithm. However, since the idea of both algorithms is the same, a unified algorithm is presented as a number of procedures that combine the similar parts of both algorithms. When a procedure needs to perform differently for ordinary and exact lumping, we provide two versions of it.

In some cases, we provide two versions of a procedure that produce exactly the same results: a "simple" and an "efficient" version. The simple version is easier to understand but not time-efficient. The efficient version is used for the purpose of computing time complexities.

The pseudocode of our lumping algorithm is given in Figure 3.1. LumpMRP takes the original MRP $M$ as its only parameter and returns the lumped MRP $\widetilde{M}$ as its output. It uses the conditions of Definition 2.2 for computing the initial partition and for the repetitive refinement of the initial partition, and uses Theorem 2.3 to compute $\widetilde{M}$.

The algorithm works in three stages. The first stage (line 1) computes the initial partition $\mathcal{P}^{\text{ini}}$ based on the reward vector $\mathbf{r}$ for ordinary lumping (condition (*a1*) of Definition 2.2) or based on initial probability distribution $\boldsymbol{\pi}^{\text{ini}}$ for exact lumping (condition (*b1*) of Definition 2.2). The second stage (line 2) is the main and novel part of the algorithm. The ComputeCoarsestPartition procedure computes the coarsest partition $\mathcal{P}$ with respect to which MRP $M$ is lumpable (condition (*a2*) or (*b2*) of Definition 2.2) by repetitive refinement of $\mathcal{P}^{\text{ini}}$. In the last stage (lines 3-5), the lumped MRP $\widetilde{M}$ is computed according to

```
LumpMRP(M)
 1  P^ini := ComputeInitialPartition(M)
 2  P := ComputeCoarsestPartition(Q, S, P^ini)
 3  R̃ := ComputeLumpedCTMC(R, S, P)
 4  S̃ := {1, ..., |P|}
 5  Compute r̃ and π̃^ini
 6  return M̃ = (R̃ − rs(R̃), S̃, r̃, π̃^ini)
```

Figure 3.1: Pseudocode of the lumping algorithm

Theorem 2.3 by construction of each of its components. Finally, $\widetilde{M}$ is returned in line 6.

ComputeInitialPartition computes the coarsest partition $\mathcal{P}^{\text{ini}}$ that satisfies condition (*a1*) of Definition 2.2 for ordinary lumping and condition (*b1*) for exact lumping. More formally, $\mathcal{P}^{\text{ini}}$ is computed such that it is the coarsest partition of $\mathcal{S}$ that satisfies the following condition:

$$\begin{aligned} \text{for ordinary lumping:} \quad & \forall\, C \in \mathcal{P}^{\text{ini}}, s, \hat{s} \in C : \mathbf{r}(s) = \mathbf{r}(\hat{s}) \\ \text{for exact lumping:} \quad & \forall\, C \in \mathcal{P}^{\text{ini}}, s, \hat{s} \in C : \boldsymbol{\pi}^{\text{ini}}(s) = \boldsymbol{\pi}^{\text{ini}}(\hat{s}) \end{aligned} \tag{3.1}$$

To compute $\mathcal{P}^{\text{ini}}$, ComputeInitialPartition simply sorts the states based on an appropriately chosen key. For ordinary and exact lumping, the sorting key of a state $s$ is $\mathbf{r}(s)$ and $\boldsymbol{\pi}^{\text{ini}}(s)$, respectively. The pseudocode for ComputeInitialPartition is not given here. Note that any refinement of $\mathcal{P}^{\text{ini}}$, particularly the result $\mathcal{P}$ computed by ComputeCoarsestPartition, also satisfies condition 3.1.

The pseudocode for the ComputeCoarsestPartition procedure, the second stage of the overall algorithm, is given in Figure 3.2. Lines 1-3 do the necessary initializations and lines 4-7 repetitively refine $\mathcal{P}$ until it satisfies condition (*a2*) or (*b2*) of Definition 2.2. Finally, line 8 returns the resulting partition.

Line 1 initializes $\mathcal{P}$ to $\mathcal{P}^{\text{ini}}$ because the desired partition has to be a refinement of $\mathcal{P}^{\text{ini}}$. Line 2 assigns to $L$ (a list of sets) the classes of the initial partition $\mathcal{P}^{\text{ini}}$. Line 3 allocates a vector $\boldsymbol{\sigma}$ of size $|\mathcal{S}|$. The roles of $L$ and $\boldsymbol{\sigma}$ will become clear below.

```
ComputeCoarsestPartition(Q, S, P^ini)
 1  P := P^ini
 2  L := P^ini
 3  allocate σ^(|S|)
 4  while L ≠ ∅
 5      Ψ := Pop(L)
 6      ComputeKeys(Q, Ψ, L', σ)
 7      Split(P, L, L', σ)
 8  return P
```

Figure 3.2: Pseudocode for computing the coarsest lumpable partition starting from a given initial partition $\mathcal{P}^{\text{ini}}$

$\mathcal{P}$ may be refined in each iteration of the **while** loop in Figure 3.2. Refining is a step in which at least one of the classes of $\mathcal{P}$ is partitioned into at least two (smaller) classes. $L$ plays the role of a list of "potential" *splitters* of $\mathcal{P}$. A splitter $\Psi$ with respect to a partition $\mathcal{P}$ is a set that satisfies the following condition:

$$\text{for ordinary lumping:} \quad \exists C \in \mathcal{P},\ s, s' \in C \text{ s.t. } \mathbf{Q}(s, \Psi) \neq \mathbf{Q}(s', \Psi) \text{ or} \tag{3.2}$$

$$\text{for exact lumping:} \quad \exists C \in \mathcal{P},\ s, s' \in C \text{ s.t. } \mathbf{Q}(\Psi, s) \neq \mathbf{Q}(\Psi, s'), \tag{3.3}$$

which means that the condition (*a2*) or (*b2*) in Definition 2.2 is violated by the pair $s, s'$, and hence $C$ needs to be split.

To make the unification of the ordinary and exact lumping algorithm easier and also to extend the algorithm in a way that is needed in Chapter 5, we define a function $K(\mathbf{Q}, s, \Psi)$. Its arguments are respectively the generator matrix $\mathbf{Q}$, a state index $s$, and a set of state indices $\Psi$. By choosing this function appropriately, we can customize the algorithm to compute partitions that satisfy a set of desired conditions. The range of $K$ can be any arbitrary "data type"[1] $T$ on which equality testing is well-defined.

[1]We do not formally introduce data types here because we do not believe it would improve the readability of the material. A concept of data types similar to that in the C language suffices for our discussion.

Based on Equations (3.2) and (3.3), we define $K$ as follows:

$$K(\mathbf{Q}, s, \Psi) = \begin{cases} \mathbf{Q}(s, \Psi) & \text{for ordinary lumping} \\ \mathbf{Q}(\Psi, s) & \text{for exact lumping} \end{cases} \tag{3.4}$$

That means that data type $T$, the range of $K$, is the set of non-negative real numbers, $\mathbb{R}^{\geq 0}$, on which equality testing is obviously well-defined. With such a definition of $K$, both Equations (3.2) and (3.3) can simply be put as

$$\exists\, C \in \mathcal{P},\ s, s' \in C \text{ s.t. } K(\mathbf{Q}, s, \Psi) \neq K(\mathbf{Q}, s', \Psi). \tag{3.5}$$

In Chapter 5, when we describe compositional lumping algorithms on the MD data structure, we will need to apply our lumping algorithm to individual nodes of an MD. That necessitates more sophisticated definitions of function $K$ that results in a different data type for $T$.

The **while** loop of lines 4-7 refines partition $\mathcal{P}$ with respect to all potential splitters $\Psi \in L$. Line 5 picks a splitter from the list of potential splitters $L$, deletes it from $L$, and assigns it to $\Psi$. The order in which the splitters are picked from $L$ does not affect the correctness or the time complexity of the algorithm. For each splitter $\Psi \in L$, line 6 (ComputeKeys) computes $K(\mathbf{Q}, s, \Psi)$ for each state $s$ in some subset of $\mathcal{S}$ and stores it in $\boldsymbol{\sigma}(s)$, which is of data type $T$. Then, Split partitions each class $C \in \mathcal{P}$ that satisfies Equation (3.5) into subclasses according to vector $\boldsymbol{\sigma}$. Split may also add a number of new potential splitters to $L$. When $L$ is empty, $\mathcal{P}$ has been refined w.r.t. all potential splitters, and the **while** loop finishes. The list of states $L'$, which is computed by ComputeKeys and is used by Split, is the set of all states whose corresponding element in vector $\boldsymbol{\sigma}$ we need to examine for refinement purposes in Split.

Figure 3.3 gives the simple version of the pseudocode of ComputeKeys for ordinary and exact lumping. In Figure 3.3(a), $L'$ is assigned the set of all predecessors of states in $\Psi$, i.e., $\bigcup_{s \in \Psi} e^{-1}(s)$, whereas in Figure 3.3(b), $L'$ is the set of all successors of states in $\Psi$, i.e.,

```
ComputeKeys(Q, Ψ, L', σ)
1  L' := ⋃_{s∈Ψ} e^{-1}(s)
2  foreach s ∈ L'
3      σ(s) := K(Q, s, Ψ)
```
(a) Ordinary lumping

```
ComputeKeys(Q, Ψ, L', σ)
1  L' := ⋃_{s∈Ψ} e(s)
2  foreach s ∈ L'
3      σ(s) := K(Q, s, Ψ)
```
(b) Exact lumping

Figure 3.3: Simple version of ComputeKeys for ordinary and exact lumping

```
ComputeKeys(Q, Ψ, L', σ)
1  L' := ∅
2  foreach s ∈ Ψ
3      foreach predecessor s' of s
4          σ(s') := 0
5  foreach s ∈ Ψ
6      foreach predecessor s' of s
7          σ(s') := σ(s') + Q(s', s)
8          L' := L' ∪ {s'}
```
(a) Ordinary lumping

```
ComputeKeys(Q, Ψ, L', σ)
1  L' := ∅
2  foreach s ∈ Ψ
3      foreach successor s' of s
4          σ(s') := 0
5  foreach s ∈ Ψ
6      foreach successor s' of s
7          σ(s') := σ(s') + Q(s, s')
8          L' := L' ∪ {s'}
```
(b) Exact lumping

Figure 3.4: Efficient version of ComputeKeys for ordinary and exact lumping

$\bigcup_{s\in\Psi} e(s)$. Figure 3.4 shows the efficient version of ComputeKeys. Notice how $L'$ and $\boldsymbol{\sigma}(s)$ are computed.

The pseudocode for the Split procedure is given in Figure 3.5. It refines $\mathcal{P}$ with respect to splitter $\Psi$, and may also add a number of new potential splitters to $L$. More specifically, Split finds every class $C \in \mathcal{P}$ that satisfies condition (3.5), and splits each such class $C$ into subclasses $\{C^1, \ldots, C^k\}$ such that

$$\forall\, 1 \le c \le k,\ \forall\, s, s' \in C^c, \qquad \boldsymbol{\sigma}(s) = \boldsymbol{\sigma}(s')$$
$$\forall\, 1 \le c \ne c' \le k,\ \forall\, s \in C^c, s' \in C^{c'}, \qquad \boldsymbol{\sigma}(s) \ne \boldsymbol{\sigma}(s),$$

and adds all subclasses of $C$ except the largest one to $L$. If there is more than one subclass of maximal size, we add all of them except for one.

Line 1 of Split initializes $L''$, which is the list of all classes of $\mathcal{P}$ that are partitioned in a refinement step. Lines 2-6 split each class $C$ based on the value of $\boldsymbol{\sigma}$ of its elements.

```
SPLIT(P, L, L', σ)
 1  L'' := ∅
 2  foreach s ∈ L'
 3      Let C be an alias for C_[s]_P
 4      delete s from C
 5      INSERT(C_T, (σ(s), s))
 6      L'' := L'' ∪ C
 7  foreach C ∈ L''
 8      C' := largest class of {C, V_k1, ..., V_k|C_T|}
 9      L := L ∪ {C, V_k1, ..., V_k|C_T|} − {C'}
10      P := P ∪ {C, V_k1, ..., V_k|C_T|}
```

Figure 3.5: Pseudocode of refinement step

Each class $C$ has a corresponding binary search tree (BST) $C_T$, which we call the *subclass tree*. Each node of $C_T$ stores the list of elements of $C$ that have the same value of $\boldsymbol{\sigma}$. Any implementation of a BST data structure (e.g., the red-black tree [31]) can be used as the subclass tree. In Section 3.3, we will see how exploiting the full potential of an appropriate BST helps us achieve the $\mathcal{O}(m \log n)$ running time for LUMPMRP.

The set of nodes of $C_T$ is denoted by $\{k_1, \ldots, k_{|C_T|}\}$. Each node $k_j$ of $C_T$ contains a key $sum_{k_j}$ and also a set $V_{k_j}$ ($k_j$'s *V-set*) of states, each of which has the same value in vector $\boldsymbol{\sigma}$ (Figure 3.6). Accordingly, the set of $V$-sets of $C_T$'s nodes is denoted by $\{V_{k_1}, \ldots, V_{k_{|C_T|}}\}$. For any node $k_j$ and any $s \in V_{k_j}$, we have $sum_{k_j} = \boldsymbol{\sigma}(s) = K(\mathbf{Q}, s, \Psi)$. INSERT$(B_T, (\boldsymbol{\sigma}(s), s))$ in line 5 adds $s$ to the $V_{k_j}$ whose key is equal to $\boldsymbol{\sigma}(s)$, i.e., $sum_{k_j} = \boldsymbol{\sigma}(s)$.

For any element $s \notin L'$ and $s \in C$, we will have $\boldsymbol{\sigma}(s) = 0$; $s$ will not be removed from $C$, and therefore will not be inserted into $C_T$ either. All such elements (if any) will remain in $C$, and, along with the $V$-sets of $C_T$, constitute the subclasses of the original class $C$. Lines 7-10 update $\mathcal{P}$ and the list of potential splitters $L$ by adding to $L$ all the subclasses of a class except for (one of) the largest one(s)[2]. Excluding the largest subclass from the set of potential splitters is similar to the "process the smaller half" strategy given by Hopcroft [1, 56]. The difference is that in Hopcroft's strategy a set $C$ is partitioned

[2]There could be two or more subclasses that have the largest size.

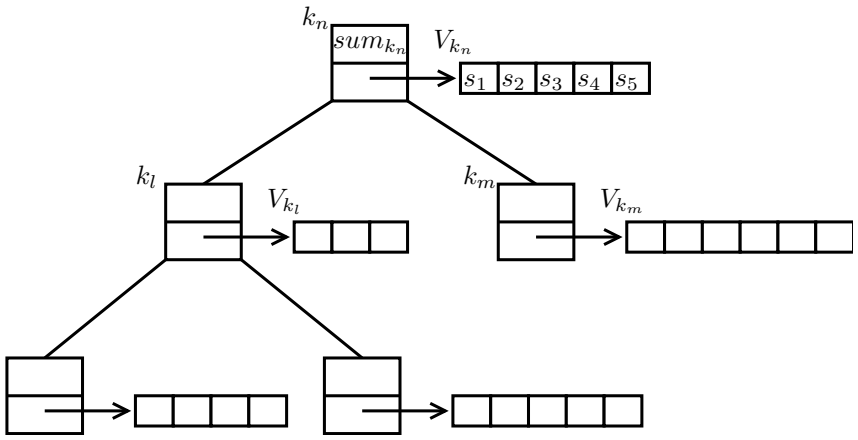



Figure 3.6: The subclass tree $C_T$ associated with each class $C$

into two subsets, while in our case, $C$ can be partitioned into more than two subsets. The similarity is that in both approaches the largest subset is not processed, ensuring that for each subset $A \subset C$ to be processed, $|A| \leq |C|/2$. We can neglect the largest class (or any other single class), because its power of splitting other classes is maintained by the remaining subclasses. For example, in Figure 3.6, $V_{k_m}$ is the largest $V$-set (having 6 states); therefore, whichever is larger, $V_{k_m}$ or the set of remaining elements of $C$, will be the only subclass that is excluded from the set of new potential splitters that are added to $L$ in line 9.

After the lumping partition is computed by ComputeCoarsestPartition in line 2 of LumpMRP (Figure 3.1), the next stage is to compute the $\widetilde{\mathbf{R}}$ matrix of the MRP lumped with respect to $\mathcal{P}$. That computation is based on Theorem 2.3. Figure 3.7 shows the simple and efficient versions of the ComputeLumpedCTMC procedure for ordinary lumping. Similarly, Figure 3.8 shows the simple and efficient versions of the same procedure for exact lumping ($\widetilde{n} = |\mathcal{P}| \leq n$). It is not difficult to see why the simple and efficient versions of each procedure produce the same results.

```
ComputeLumpedCTMC(R, S, P)
 1  foreach ĩ ∈ {1, ..., ñ}
 2      R̃(ĩ, ĩ) := 0
 3  foreach ĩ, j̃ ∈ {1, ..., ñ}, ĩ ≠ j̃
 4      s := arbitrary element of C_ĩ ∈ P
 5      R̃(ĩ, j̃) = K(R, s, C_j̃)
 6  return R̃
```

(a) Simple version

```
ComputeLumpedCTMC(R, S, P)
 1  allocate matrix R̃^(ñ×ñ)
 2  initialize R̃ to zero
 3  foreach ĩ ∈ {1, ..., ñ}
 4      s := arbitrary element of C_ĩ ∈ P
 5      foreach successor s' of s
 6          j̃ := [s']_P
 7          R̃(ĩ, j̃) = R̃(ĩ, j̃) + R(s, s')
 8  return R̃
```

(b) Efficient version

Figure 3.7: ComputeLumpedCTMC's pseudocode for ordinary lumping

## 3.3 Time Complexity

In this section, we will prove that when we use splay trees [85] as the subclass trees, the lumping algorithm, that is, the LumpMRP procedure, runs in $\mathcal{O}(m \log n)$ time. In order to show how the properties of splay trees (compared with other balanced BSTs) affect the running time of the algorithm, we are going to analyze the running time of LumpMRP in two configurations. In the first configuration, we use a general balanced BST for the subclass trees. We assume that insertion and query of a node in a general balanced BST take $\mathcal{O}(\log k)$ time in the worst case, where $k$ is the number of nodes in the tree. For the first configuration, we will show that the worst-case running time of the algorithm is $\mathcal{O}(m \log^2 n)$. In the second configuration, we use a splay tree for the subclass trees. Using the *static optimality* property of splay trees, we show that we can take an $\mathcal{O}(\log n)$ factor out of the time complexity we obtained when we used a general balanced BST; in other words, we achieve the $\mathcal{O}(m \log n)$ running time. Since the subclass tree is only used in stage 2

```
ComputeLumpedCTMC(R, S, P)
 1  foreach ĩ ∈ {1, ..., ñ}
 2      R̃(ĩ, ĩ) := 0
 3  foreach ĩ, j̃ ∈ {1, ..., ñ}, ĩ ≠ j̃
 4      s := arbitrary element of C_j̃ ∈ P
 5      R̃(ĩ, j̃) = K(R, s, C_ĩ)
 6  return R̃
```

(a) Simple version

```
ComputeLumpedCTMC(R, S, P)
 1  allocate matrix R̃^(ñ×ñ)
 2  initialize R̃ to zero
 3  foreach j̃ ∈ {1, ..., ñ}
 4      s := arbitrary element of C_j̃ ∈ P
 5      foreach predecessor s' of s
 6          ĩ = [s']_P
 7          R̃(ĩ, j̃) = R̃(ĩ, j̃) + R(s', s)
 8  return R̃
```

(b) Efficient version

Figure 3.8: ComputeLumpedCTMC's pseudocode for exact lumping

of LumpMRP (i.e., the ComputeCoarsestPartition procedure), the choice of the tree data structure affects only the time complexity of that stage. As our algorithm is a variant of the one proposed by Paige and Tarjan, both of our running time proofs are based on the running time proof given in [76].

Knowledge of the data structures we use to represent the entities in our algorithm is essential to the time complexity proof. Except for the tree data structure used for the subclass tree, all of the data structures are common between the two configurations. A partition is stored as a doubly-linked list whose elements are the classes. Each class points to a doubly-linked list whose elements are the elements of the class. Each element (state) has a pointer to its class. Therefore, finding the class $C_{[s]_\mathcal{P}}$ of an element $s$, or adding or deleting a class or an element of a class, takes constant time. The sets $L$, $L'$, and $L''$ are also stored as doubly-linked lists.

The $\mathbf{Q}$ matrix can be stored in any data structure that gives access to the elements of

a specific row or column of the matrix in $\mathcal{O}(k)$ time, where $k$ is the number of elements in that row or column. An example of such a data structure is adjacency lists in which each vertex (state $s$) has two associated linked lists: one for successor vertices (given by $e(s)$) and one for predecessor vertices (given by $e^{-1}(s)$)[3]. The transition rates in $\mathbf{Q}$ are stored as edge weights in the adjacency list data structure. However, if the data structure used for the representation of $\mathbf{Q}$ gives access to a specific row or column with a time complexity larger than what was mentioned above, then the algorithm still works correctly but does not achieve the $\mathcal{O}(m \log n)$ time complexity. A matrix diagram is an example of such a data structure.

Since only ComputeCoarsestPartition (line 2 of LumpMRP in Figure 3.1) is affected by the choice of the subclass tree data structure, we first analyze the running time of all lines of LumpMRP except line 2. As we explained before, ComputeInitialPartition in line 1 performs a sorting operation on a set of $n$ numbers. That will take $\mathcal{O}(n \log n)$ time. Lines 1 and 2 of ComputeLumpedCTMC (Figure 3.7(b) or Figure 3.8(b)) take $\mathcal{O}(\widetilde{n})$ time because the matrices are represented as adjacency lists. The **foreach** loop of lines 3-7 executes $\widetilde{n}$ times. The **foreach** loop of lines 5-7 executes at most $m$ times because each edge $s \rightarrow s'$ of $\mathbf{R}$ is considered at most once. Therefore ComputeLumpedCTMC takes $\mathcal{O}(m + \widetilde{n})$ time. Line 4 of Figure 3.1 does not actually take any time; it is shown for the sake of clarity. Based on Theorem 2.2, line 5 takes $\mathcal{O}(n)$ time.

**Lemma 3.1** LumpMRP *except line 2 (*ComputeCoarsestPartition*) takes* $\mathcal{O}(n \log n + m)$ *time to execute.*

**Proof.** As we showed above, ComputeInitialPartition takes $\mathcal{O}(n \log n)$ time, ComputeLumpedCTMC takes $\mathcal{O}(m + \widetilde{n})$ time, and lines 4-6 take $\mathcal{O}(n)$ to execute. Therefore, LumpMRP except line 2 takes

$$\mathcal{O}(n \log n) + \mathcal{O}(m + \widetilde{n}) + \mathcal{O}(n) = \mathcal{O}(n \log n + m)$$

[3]In the actual implementation, only one of the lists is actually stored at any given time.

time to execute. ■

ComputeKeys is the last procedure of the lumping algorithm whose running time is independent of the subclass tree data structure.

**Lemma 3.2** *The ordinary and exact lumping versions of the* ComputeKeys *procedure in Figure 3.4 take* $\sum_{s\in\Psi}\mathcal{O}(1+|e^{-1}(s)|)$ *and* $\sum_{s\in\Psi}\mathcal{O}(1+|e(s)|)$ *time to execute, respectively.*

**Proof.** (a) Ordinary lumping: The **foreach** loop of lines 2-4 of Figure 3.4(a) executes $|\Psi|$ times and the **foreach** loop of lines 3 and 4 executes $|e^{-1}(s)|$ times for each $s \in \Psi$. Therefore, lines 2-4 take $\mathcal{O}(|\Psi| + \sum_{s\in\Psi}|e^{-1}(s)|)$ time to execute. In the implementation, each element $s'$ has a flag that shows its membership in $L'$, so line 8 takes constant time. Therefore, lines 5-8 also take $\mathcal{O}(|\Psi| + \sum_{s\in\Psi}|e^{-1}(s)|)$ time. Finally, note that $\mathcal{O}(|\Psi| + \sum_{s\in\Psi}|e^{-1}(s)|) = \sum_{s\in\Psi}\mathcal{O}(1+|e^{-1}(s)|)$.

(b) Exact lumping: Consider Figure 3.4(b). The proof is the same as the one for case (a) except that we need to replace function $e^{-1}$ with function $e$. ■

The rest of our proofs and discussions in this section deal with the time complexity of the ComputeCoarsestPartition procedure of the ordinary lumping algorithm. Its time complexity for exact lumping can be proved with only minor changes; Lemma 3.2 is one example of such a change.

### 3.3.1 Using General Balanced BSTs

In the balanced BST configuration, we choose an arbitrary balanced BST as the subclass tree. We assume that the selected data structure provides insertion and query in $\mathcal{O}(\log k)$ time in the worst case.

**Lemma 3.3** *If general balanced BSTs are used,* Split$(\mathcal{P}, L, L', \boldsymbol{\sigma})$ *takes* $\sum_{s\in\Psi}\mathcal{O}(|e^{-1}(s)|(1+\log n))$ *time.*

**Proof.** Line 1 takes constant time. The **foreach** loop of lines 2-6 of SPLIT executes $|L'|$ times. $L'$ in Split is computed in ComputeKeys. From Figure 3.3, we can see that

$|L'| \leq \sum_{s\in\Psi} |e^{-1}(s)|$. Lines 3 and 4 take constant time. By our assumption regarding general balanced BSTs, it takes at most $\mathcal{O}(\log n)$ to execute line 5. By using the same technique used for $L'$, we can execute line 6 in constant time. Therefore, lines 2-6 take $\mathcal{O}((1+\log n)\sum_{s\in\Psi} |e^{-1}(s)|)$. Note that at the beginning of line 7, $|L''| \leq |L'| \leq \sum_{s\in\Psi} |e^{-1}(s)|$. Lines 7-10 add all the new potential splitters to $L$. The number of new potential splitters is bounded by $|L'|$; the worst case happens when each element in $L'$ constitutes its own $V$-set. Finally, note that the **foreach** loop of lines 7-10 executes $|L''| \leq \sum_{s\in\Psi} |e^{-1}(s)|$ times. Therefore, lines 7-10 take $\mathcal{O}(\sum_{s\in\Psi} |e^{-1}(s)|)$ time. Thus, SPLIT takes a total of $\mathcal{O}((1+\log n)\sum_{s\in\Psi} |e^{-1}(s)|) = \sum_{s\in\Psi} \mathcal{O}(|e^{-1}(s)|(1+\log n))$ time to execute. ■

**Lemma 3.4** *Regardless of the subclass tree data structure, any element $s$ can appear at most $\lceil \lg(n+1) \rceil$ times in any potential splitter $\Psi$ in all executions of* SPLIT.

**Proof.** Let $C$ be the class to which $s$ belongs at some point in the algorithm, and assume $C$ is partitioned. Then, $s$ belongs either to the largest subclass (which is no longer considered a splitter) or to any other subclass, in which case the subclass is considered a potential splitter. In the latter case, the size of the subclass is at most $|C|/2$. In the worst case, each time its class is partitioned, $s$ ends up in a subclass that is not discarded as the largest, and hence has at most half the size of the original class. Moreover, in the initial partition, none of the classes, including $s$'s class, will be larger than $n$. Therefore, for any $n$ and $k \in \mathbb{N}$ such that $2^{k-1} \leq n < 2^k$, $s$ belongs to at most $k$ potential splitters. We will then have,

$$2^{k-1} \leq n < 2^k \;\Rightarrow\; k-1 < \lg(n+1) \leq k \;\Rightarrow\; k = \lceil \lg(n+1) \rceil$$

■

Therefore, the running time of all executions of SPLIT is

$$\begin{aligned}
\sum_{\substack{\text{all splitters}\\ \Psi}} \sum_{s\in\Psi} \mathcal{O}\big(1 + |e^{-1}(s)|(1+\log n)\big) &= \mathcal{O}\Big(\lg(n+1) \sum_{s\in\mathcal{S}} \big(1 + |e^{-1}(s)|(1+\log n)\big)\Big) \\
&= \mathcal{O}\Big(\log n \times \big(n + m(1+\log n)\big)\Big) \\
&= \mathcal{O}(m \log^2 n)
\end{aligned}$$

Notice that the range of the second summation is $s \in \Psi$ and that of the third summation is $s \in \mathcal{S}$. Hence,

**Lemma 3.5** *If general balanced BSTs are used as the subclass trees, all executions of* SPLIT *take* $\mathcal{O}(m \log^2 n)$ *time.*

Similarly, based on Lemma 3.2, the running time of all executions of COMPUTEKEYS is

$$\begin{aligned}
\sum_{\substack{\text{all splitters} \\ \Psi}} \sum_{s \in \Psi} \mathcal{O}\big(1 + |e^{-1}(s)|\big) &= \mathcal{O}\Big(\lg(n+1) \sum_{s \in \mathcal{S}} \big(1 + |e^{-1}(s)|\big)\Big) \\
&= \mathcal{O}\Big(\log n \times \big(n + m\big)\Big) \\
&= \mathcal{O}(m \log n)
\end{aligned}$$

**Lemma 3.6** *If general balanced BSTs are used as the subclass trees, all executions of* COMPUTEKEYS *take* $\mathcal{O}(m \log n)$ *time.*

**Theorem 3.1** *If general balanced BSTs are used as the subclass trees,* COMPUTECOARSESTPARTITION *takes* $\mathcal{O}(m \log^2 n)$ *time.*

**Proof.** Lines 1 and 2 of Figure 3.2 take $\mathcal{O}(|\mathcal{P}^{\text{ini}}|) = \mathcal{O}(n)$ time. Line 3 takes constant time. For each iteration of the **foreach** loop of lines 4-7, line 5 takes constant time. Based on Lemmas 3.5 and 3.6, lines 6 and 7 take a total of $\mathcal{O}(m \log^2 n)$ time for all iterations. Therefore, the whole procedure takes $\mathcal{O}(m \log^2 n)$ time. ■

**Theorem 3.2** *If general balanced BSTs are used as the subclass trees,* LUMPMRP *takes* $\mathcal{O}(m \log^2 n)$ *time.*

**Proof.** Use Lemma 3.1 and Theorem 3.1. ■

The part of the lumping algorithm that increased the complexity from $\mathcal{O}(m \log n)$ to $\mathcal{O}(m \log^2 n)$ is line 5 of SPLIT (Figure 3.5). In the next section, we will show how a statically optimal tree (e.g., a splay tree) will allow us to perform a tighter analysis.

### 3.3.2 Using Splay Trees

In the second configuration, we use splay trees to represent subclass trees. Based on the "splaying" heuristic, a splay tree is a BST for which only one restructuring operation, the splaying step, is defined. Each time a node is inserted, deleted, or queried, a number of splaying steps are performed to restructure the tree. For more details see [85].

The key property of splay trees that lets us do a tighter analysis of the running time of the algorithm is given by the following theorem:

**Theorem 3.3 (Static Optimality Theorem [85])** *Suppose that $T$ is an initially empty splay tree and that a sequence of $q$ accesses (insertion, deletion, or query) to $n$ different elements is to be done on $T$. Also, suppose the number of accesses to element $i$ is $q_i > 0$ ($q = \sum_{i=1}^{n} q_i$). Then, the total time to perform the sequence of accesses is*

$$\mathcal{O}\Big(q + \sum_{i=1}^{n} q_i \log\big(\frac{q}{q_i}\big)\Big)$$ ■

If we amortize the total time given above over all the accesses we will have:

**Corollary 3.1** *Given the assumptions of Theorem 3.3, an access to element $i$ takes $\mathcal{O}(1 + \log(\frac{q}{q_i})) = \mathcal{O}(1 + \log q - \log q_i)$ amortized time.* ■

According to the analysis given in Lemma 3.3, in an execution of Split, lines 1-10 minus line 5 take $\mathcal{O}(\sum_{s\in\Psi} |e^{-1}(s)|)$ time. Thus, using Lemma 3.4, we observe that lines 1-4 and lines 6-10 take a total time of $\mathcal{O}(m \log n)$ during the execution of ComputeCoarsestPartition. Therefore, in order to prove that ComputeCoarsestPartition runs in $\mathcal{O}(m \log n)$, it is enough to prove that line 5 of Split takes a total time of $\mathcal{O}(m \log n)$ over all executions of Split.

**Lemma 3.7** *When splay trees are used as the subclass trees, all executions of* Split *in* ComputeCoarsestPartition *take $\mathcal{O}(m \log n)$ time.*

**Proof.** Consider an element $s \in \mathcal{S}$. If $s \in L'$, it will be inserted into a subclass tree in line 5. It will then be put into a (possibly smaller) class. In fact, lines 2-6 partition some of the classes into smaller classes (the smaller classes replace the previous ones). Suppose $s$'s class is partitioned $j$ times during the execution of ComputeCoarsestPartition. Let $C^i$ be the $i^{\text{th}}$ class to which $s$ belongs ($C^0 \supseteq C^1 \supseteq \ldots \supseteq C^j$). In order to partition $C^i (0 \leq i < j)$ into its subclasses, we create the tree $C_T^i$, into which at most $|C^i|$ elements will be inserted. The number of accesses to a node $k_l$ of $C_T^i$ will be $|V_{k_l}|$; that is the size of the $V$-set of $k_l$, and also equals the size of the subclass associated with $k_l$. The subclass of $s$ is $C^{i+1}$; therefore, by applying Corollary 3.1 to accesses to $C_T^i$, it takes $\mathcal{O}(1+\log|C^i|-\log|C^{i+1}|)$ time to insert $s$ into $C_T^i$, i.e., to perform line 5 in Split.

We break this running time into $\mathcal{O}(1)$ and $\mathcal{O}(\log|C^i|-\log|C^{i+1}|)$. As we showed earlier, line 5 is executed $\mathcal{O}(m \log n)$ times during the runtime of ComputeCoarsestPartition, i.e., for all elements of $\mathcal{S}$ and for all possible values of $i$. Therefore, the $\mathcal{O}(1)$ part takes $\mathcal{O}(m \log n)$. The $\mathcal{O}(\log|C^i| - \log|C^{i+1}|)$ part of all executions (i.e., for all $i$, $0 \leq i < j$) of Insert$(C_T^i, (\boldsymbol{\sigma}(s), s))$ takes

$$\sum_{i=0}^{j-1} \mathcal{O}\big(\log|C^i| - \log|C^{i+1}|\big) = \mathcal{O}(\log|C^0| - \log|C^j|).$$

Since $1 \leq |C^i| \leq n$, we have $\log|C^0| - \log|C^j| < \log n$. Therefore, the $\mathcal{O}(\log|C^i| - \log|C^{i+1}|)$ part of all executions of Insert$(C_T^i, (\boldsymbol{\sigma}(s), s))$ takes $\mathcal{O}(\log n)$ time and consequently $\mathcal{O}(n \log n)$ time for all elements of $\mathcal{S}$. Hence line 5, and therefore Split, takes a total of $\mathcal{O}(m \log n)$ time. ■

Now, we can state the following theorem, which is the counterpart of Theorem 3.2 for splay trees.

**Theorem 3.4** *Under the assumptions of the second configuration, i.e., if splay trees are used as the subclass trees,* ComputeCoarsestPartition*, and consequently* LumpMRP*, takes $\mathcal{O}(m \log n)$ time.*

**Proof.** Use Lemmas 3.1, 3.6, and 3.7. ■

### 3.3.3 Lower Bound on Complexity

The question that naturally comes to mind after analyzing the running time of the algorithm is whether that algorithm is the fastest possible. The answer is that we do not know. To the best of our knowledge, it is the fastest algorithm published to this date. Moreover, there is no known result that shows a tight lower bound for the state-level CTMC lumping problem. Nevertheless, we will prove a loose $\Omega(m + n \log n)$ lower bound for the time required to solve the problem, which we hope will give us a sense of how hard the problem is. The model of computation we consider here is a $k$-ary decision tree where $k$ is a constant.

**Theorem 3.5** *A lower bound for the time complexity of the CTMC lumping problem is* $\Omega(m + n \log n)$.

**Proof.** The main output result of the lumping algorithm is the partition $\mathcal{P}$ with respect to which the MRP is lumpable. What we are interested in is the number of all possible values of $\mathcal{P}$. In other words, we would like to know the number of all possible ways a set of $n$ elements can be partitioned. That number is known as the *Bell* number $B_n$, and according to [92]:

$$\frac{\ln B_n}{n} = \ln n - \ln\ln n - 1 + \frac{\ln\ln n}{\ln n} + \frac{1}{\ln n} + \frac{1}{2}\left(\frac{\ln\ln n}{\ln n}\right)^2 + \mathcal{O}\left(\frac{\ln\ln n}{(\ln n)^2}\right).$$

By disregarding the terms smaller than $\ln n$, we could say that $\log B_n = \mathcal{O}(n \log n)$, which is the minimum height of the computation tree that gives the partition $\mathcal{P}$ at its leaves. Moreover, all the non-zero entries of $\mathbf{Q}$ have to be accessed at least once by any lumping algorithm. Therefore, any algorithm takes at least $\Omega(m + n \log n)$ computational steps to solve the lumping problem. ■

## 3.4 Performance Results

Apart from the theoretical importance of the better time complexity of our algorithm compared to the ones known before, we are also interested in the running time of the implementation of the algorithm in practice. In particular, we investigate the effect of using splay trees and general balanced BSTs on the running time of the lumping algorithm's implementation. Moreover, we compare the performance of the implementation of Buchholz's lumping algorithm [19] with the implementation of our algorithm given in this chapter. Finally, we study the effect of the size of the resulting lumpable partition on the running time of the two variants of our algorithm and that of Buchholz's algorithm.

The specific general balanced BST that we have used in our implementation is the red-black tree [31]. It satisfies our requirement that it perform any insertion or query operation in $\mathcal{O}(\log k)$ worst-case time, where $k$ is the size of the tree. Specifically, we have used the STL's (Standard Template Library) [75] implementation of the red-black tree data structure. For the splay trees, we used a slightly improved (by a small constant factor) version of the splay tree implementation of [91].

We have performed all of our experiments on a PC with an Athlon XP 2400 processor and 1.5 GB of RAM. We have used the gcc 3.3 compiler and the -O3 optimization setting. Our implementation is based on the Möbius modeling tool [38, 48]. We obtained the implementation of the algorithm in [19] directly from its author. That implementation was compiled using the same compiler and settings we used for our implementation. In all the experiments, the generator matrix $\mathbf{Q}$ is represented in memory using the sparse matrix representation. Therefore, accessing all $k$ elements of a row or a column takes $\mathcal{O}(k)$ time.

### 3.4.1 Example Model

To provide a number of MRPs on which we can test the implementations, we have two options: 1) to build MRPs at the state level, or 2) to build high-level Markovian models and

then generate the underlying MRP. Using the first approach for very large MRPs is obviously not practical. Moreover, using the second approach, we can automatically generate a large number of MRPs with various degrees of lumpability. Therefore, the second approach is preferable.

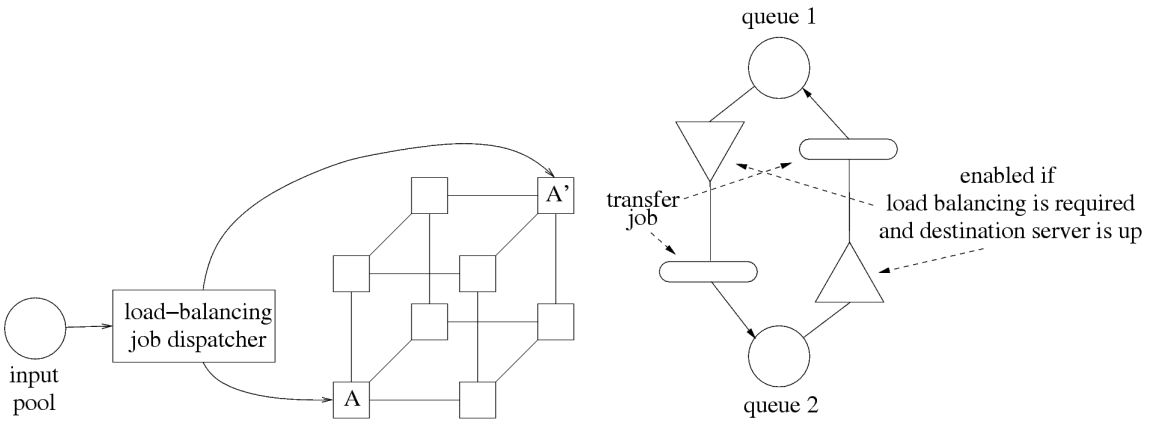
The model that we consider here is the model of a hypercube multiprocessor system. We used the Möbius tool to model the system using the SAN (stochastic activity network) [84] formalism. In the following, we give a high-level view of the model; the complete SAN specification of the model is given in Section A.1. Note that the selection of the formalism is arbitrary; what we ultimately need is some representation of an MRP that we feed to our lumping algorithm. Therefore, any other high-level Markovian model formalism could be used.

The hypercube system consists of 8 cube-connected processors, an input pool of jobs that are waiting to be serviced, and an output pool of jobs that have already been serviced by a processor in the system (Figure 3.9). The system takes jobs from its input pool, processes them using its processors, and passes them to its output pool. By superimposing the input pool and the output pool, we have made the system closed in the sense that there is always a constant number $J$ of jobs in the system.

All the processors have the same behavior (given they are in the same state) except for two processors $A$ and $A'$ (Figure 3.9(a)). $A$ and $A'$ have the same behavior as each other but are special (with respect to the other hypercube processors) in that they receive jobs from the input pool as described later. Processors in the hypercube can fail, and are repaired by a single repair facility that picks processors to repair uniformly from the pool of failed processors[4]. A failed processor keeps its jobs in its queue unless they are transferred to a neighbor processor by the load-balancing scheme described below. The subsystem is considered unavailable when two or more processors are down.

Each processor has a queue with capacity $J$. Jobs enter the processors' queues via a

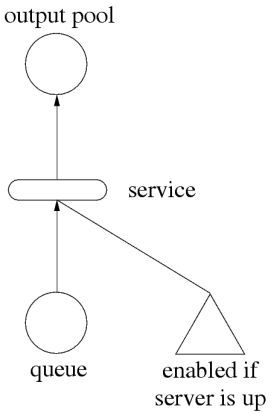
[4]Failure and repair behaviors of the system are not shown in Figure 3.9.




(a) Connection between processors




(b) Load-balancing between each two neighboring processors




(c) Service in each processor

Figure 3.9: Hypercube system (triangles specify enabling functions for transitions)

dispatcher that picks a job from the subsystem's input pool and assigns it to either processor $A$ or $A'$ with a probability distribution that favors the processor that has fewer jobs in its queue (Figure 3.9(a)). Another load-balancing scheme that governs the distribution of jobs among the processors is that whenever a processor has $> 1$ more jobs than any of its neighbors, it sends one of its jobs to one of those neighbors, again assigning higher probability to processors with fewer jobs (Figure 3.9(b)). That is how jobs are distributed among all processors. Finally, if a processor fails, it transfers jobs in its queue one by one, with an exponentially distributed delay, to a neighbor processor that is not down.

As we can see from the model specification, the lumpability of the MRP generated from the hypercube model is due to the model's symmetrically-connected equally-behaving

processors. The symmetry in fact generates an MRP that is strictly lumpable (i.e., ordinarily and exactly lumpable) with respect to some partition of the state space (see Theorem 3.4 of [17] for a proof). Moreover, all MRPs that are lumpable only due to symmetry of their underlying models (like the hypercube model) have the property that the coarsest partitions with respect to which the MRP is ordinarily, exactly, and strictly lumpable are all the same. Therefore, in all our experiments, we apply only the ordinary lumping algorithm to the MRPs generated from the hypercube model.

As we mentioned earlier, we intend to study the effect of output size (i.e., the size of the lumpable partition) on the running time of the algorithms. In order to perform this study, we need to have a number of MRPs of the same size that have the same $\rightarrow$ relation, except that they are different only in the rates of their $\mathbf{Q}$ matrices. The lumped versions of those MRPs will possibly have different sizes, due to their different $\mathbf{Q}$ matrices. By running lumping algorithms on such MRPs, we can learn how the running time of the algorithm changes depending on the size of the computed partition.

To generate such MRPs, we start with the original hypercube model that we described above. It has the maximum degree of symmetry, and therefore leads to the smallest lumpable partition. By changing the service and repair rates of some of the processors, we can achieve different degrees of symmetry up to the point that there is no symmetry in the model. Hence, the coarsest lumpable partition is of the same size as the state space, i.e., no lumping is possible.

### 3.4.2 Performance Results

First, we compare the performance of three implementations of MRP lumping algorithms: 1) the implementation of our algorithm that uses the red-black tree data structure for the subclass trees, 2) the implementation of our algorithm that uses the splay tree data structure for the subclass trees, and 3) the implementation of the lumping algorithm given in [19] by Buchholz.

Table 3.1(a) shows the sizes of the original and the lumped MRPs resulting from the hypercube model. The second and third columns are the number of states ($n$) and non-zero $\mathbf{Q}$ matrix entries ($m$) of the original MRP, and the fourth and fifth columns are the number of states ($\widetilde{n}$) and non-zero $\widetilde{\mathbf{Q}}$ matrix entries ($\widetilde{m}$) of the lumped MRP. Obviously, the computed lumped MRPs do not depend on the implementation used, since all the algorithms generate the same output.

The second through fourth columns of Table 3.1(b) show the running times of the three implementations that we mentioned above for each MRP. All times are in seconds. The fifth and sixth columns show by how much the red-black implementation is faster than the splay and Buchholz implementations, respectively. The seventh column gives the average size of the subclass trees created during the running time of our implementation. Note that subclass trees are used only by the red-black and splay tree implementations.

In Table 3.1(b), we can see that the red-black implementation is faster than the splay tree implementation by 6.6% to 9.1%. At the first look, the result appears to contradict the time complexities that we proved in Section 3.3. However, by looking at the average sizes of the subclass trees, which are very close to 1, we recognize that almost all the subclass trees generated during the running time of the algorithm are of size 1. Because of the extreme smallness of the subclass trees, the constant factors that are dropped in the theoretical time complexities in the two configurations have become a dominant factor. In fact, the source code of the red-black tree and splay tree implementations (not given here) verifies the relatively larger constant factor of the running time of the splay tree operations.

All in all, the smaller constant factor of the time complexity of the red-black tree operations and the very small subclass trees causes the red-black implementation to be faster than the splay tree implementation for our example. Our experience with other models shows that the very small average size of the subclass trees is typical for virtually all models[5].

[5]Nonetheless, it is possible to handcraft MRPs for which the splay tree implementation performs faster than the red-black implementation.

(a) Specification of the lumped and unlumped MRPs

| $J$ | unlumped MRP | | lumped MRP | |
|---|---|---|---|---|
| | $n$ | $m$ | $\widetilde{n}$ | $\widetilde{m}$ |
| 3 | 10,725 | 65,075 | 943 | 5,822 |
| 4 | 32,175 | 224,585 | 2,967 | 20,265 |
| 5 | 83,655 | 651,131 | 7,753 | 58,305 |
| 6 | 195,195 | 1,657,707 | 17,893 | 146,539 |
| 7 | 418,275 | 3,815,007 | 37,816 | 333,478 |
| 8 | 836,550 | 8,098,050 | 74,762 | 701,831 |
| 9 | 1,580,150 | 16,087,540 | 139,856 | 1,385,062 |
| 10 | 2,844,270 | 30,238,454 | 249,782 | 2,590,165 |

(b) Running times and comparisons

| $J$ | running time (s) | | | red-black is faster by a factor of | | average size of subclass trees |
|---|---|---|---|---|---|---|
| | red-black | splay | Buchholz | vs. splay | vs. Buchholz | |
| 3 | 0.07 | 0.076 | 0.20 | 8.6% | 2.9 | 1.009 |
| 4 | 0.33 | 0.36 | 4.03 | 9.1% | 12.2 | 1.0066 |
| 5 | 1.10 | 1.19 | 30.05 | 8.2% | 27.3 | 1.004 |
| 6 | 3.12 | 3.37 | 170.7 | 8.0% | 54.7 | 1.0028 |
| 7 | 8.07 | 8.67 | 803.6 | 7.4% | 99.5 | 1.0025 |
| 8 | 18.85 | 20.22 | 3369 | 7.3% | 178.7 | 1.0021 |
| 9 | 41.30 | 44.08 | 13380 | 6.7% | 324 | 1.0018 |
| 10 | 85.33 | 90.99 | 47260 | 6.6% | 553.8 | 1.0017 |

Table 3.1: Performance comparison of three different implementations of state-level lumping algorithms

Therefore, it is preferable to use the red-black tree implementation in practice.

From Table 3.1(b), we can also observe that our red-black implementation is between 2.9 and 554 times faster than Buchholz's algorithm, depending on the size of the original MRP. The speedup is attributable to the ratio of the time complexities of the two algorithms: $\mathcal{O}(mn)/\mathcal{O}(m \log n) = \mathcal{O}(n/lgn)$. That ratio also explains the fact that as the size of the original MRP grows, the speedup becomes larger too.

Our second study examines the effect of the size of the lumpable partition computed by those implementations on their running time. As we explained before, we generate a number of MRPs with the same size but with different degrees of symmetry from the hypercube

| config | lumped MRP | | running time | |
|---|---|---|---|---|
| | $\widetilde{n}$ | $\widetilde{m}$ | red-black | splay |
| 1 | 37,816 | 333,478 | 8.07 | 8.67 |
| 2 | 75,607 | 666,947 | 8.91 | 9.89 |
| 3 | 214,380 | 1,936,463 | 13.26 | 15.52 |
| 4 | 418,275 | 3,815,007 | 19.62 | 23.66 |

Table 3.2: Running time of red-black and splay implementations for MRPs with different degrees of lumpability

model. Table 3.2 shows a number of different configurations of an MRP with 418,275 states and 3,815,007 non-zero $\mathbf{Q}$ entries. For each configuration, the number of states ($\widetilde{n}$) and non-zero $\mathbf{Q}$ entries ($\widetilde{m}$) of the lumped MRP and the time to generate the lumpable partition for each implementation are given. The first configuration, which is the one used in Table 3.1 ($J = 8$), has the most symmetry. The last configuration does not have any symmetry, i.e., is not lumpable.

We can see from Table 3.2 that the larger the computed lumpable partition is (i.e., the less lumpable the MRP is), the more time it takes to generate the partition. That trend was predictable, because the algorithm starts with a coarse initial partition and refines it until it reaches the resulting lumpable partition. The finer the resulting partition, the larger the number of refinement steps needed to compute the result, and the more time the algorithm takes.

## 3.5 Summary

We gave an algorithm for computing the coarsest ordinarily and exactly lumped MRPs. We presented the algorithms in such a way that they can be reused in Chapter 5 in compositional lumping algorithms. We showed that with the use of splay trees, the time complexity of the algorithm is $\mathcal{O}(m \log n)$. We also showed that a "naïve" selection of a data structure for the subclass tree (e.g., any general balanced BST) leads to an algorithm with $\mathcal{O}(m \log^2 n)$ time complexity.

We also introduced the open question of the lower bound time complexity of the MRP lumping problem. Although we could prove the $\Omega(m + n \log n)$ lower bound, there is still a relatively large gap between that lower bound and the $\mathcal{O}(m \log n)$ time complexity we proved for our algorithm, which is the fastest known algorithm for the problem.

To understand the practical implications of our theoretically faster algorithm, we also ran two implementations of our algorithm and another one with time complexity of $\mathcal{O}(mn)$ [19] on a number of MRPs that were built from a model of a hypercube of processors. We first showed that the splay tree implementation, although theoretically faster, is virtually always slightly slower than the red-black tree implementation, because almost all the subclass trees generated during refinement are of size 1. Moreover, we showed that our implementation is significantly, and in some experiments more than two orders of magnitude, faster than Buchholz's. Finally, we observed that the larger the resulting lumpable partition is, the more time each of the implementations will take, due to the larger number of refinement steps necessary to compute the result.

Our algorithm can use any compact representation for the generator or transition rate matrix that provides column and row access. Nevertheless, even the use of compact representations for the generator matrix does not make our algorithm applicable to an MRP with arbitrary size, because the algorithm would still need memory proportional to the size of the state space. In practice, our implementation can handle MRPs with up to about 30 million states using compact representation for the generator matrix on a modern workstation with 1.5 GB of RAM. For larger MRPs, other techniques, such as compositional lumping and symmetry detection, come into play. Compositional lumping techniques use lumping algorithms as subroutines and integrate them with compositionality to lump even larger MRPs. A technique that combines symmetry detection and symbolic data structures will be described in Chapter 4. In Chapter 5, we will explain another approach that combines a compositional lumping technique and symbolic data structures.

# Chapter 4

# Symbolic CTMC Construction and Numerical Analysis

In this chapter, we extend previous work on MDDs and MDs to composed models that share state variables and support next-state and weight functions that are state-dependent in general[1]. Furthermore, our work combines model-level lumping techniques, which have been applied to state-sharing composed models, with largeness-tolerance techniques that use MDDs and MDs. In particular, we present a new algorithm that symbolically generates the state space of a hierarchical model (which is built using *join* and *replicate* operators [82]) in the form of an MDD. The replicate operator imposes symmetries that create regular structures in the state space, and therefore make symbolic exploration of the state space efficient with MDDs. By using symbolic data structures, we make the representation of the state space and the state transition rate matrix orders of magnitude smaller, so that most of the available memory will be left for the storage of iteration vectors.

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

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

lumped states in the lumped CTMC.

We begin in Section 4.1 with some definitions and notations we need to specify the Markovian modeling formalism we use. In Section 4.2, we describe how we define MRPs based on Markovian models and prove that the MRPs are ordinarily and exactly lumpable with respect to partitions that depend on the structure of the model. Then, in Section 4.3, we present the symbolic state-space exploration algorithm by extending previous work [25, 26, 28, 29] on MDDs to state-sharing composed models that do not have some of the structural restrictions required by previous approaches. We also describe how the algorithm generates the unlumped state space as an MDD. In Section 4.4, we present our approach of constructing the lumped state space from the unlumped state space by integrating the model-level lumping technique of [82] with symbolic techniques, which has never been done before. Section 4.5 discusses how the algorithm constructs an MD for the state transition rate matrix of the lumped CTMC and how to enumerate the elements of that structure for numerical analysis. The proposed approach has been implemented and used for the numerical analysis of a highly redundant fault-tolerant parallel computer system [67, 81]. We also consider a well-known performance model of a communication protocol [93]. Results for the models are presented in Section 4.6. Finally, we will summarize and present possible future work in Section 4.7.

## 4.1 Background

### 4.1.1 Hierarchical Model Specification

In this chapter, we develop a representation of the CTMC of a hierarchical composed model that is built by sharing state variables (SVs) among submodels. This composition operation is the same as the one used in SAN-based reward models [82], but is different from action-synchronization composition, which has been used in superposed generalized stochastic Petri nets, (stochastic) process algebras, and stochastic automata networks. In order to describe

precisely how hierarchical composed models of discrete event systems are constructed, we start with the definition of a model and the composition operators that we use to build those models. Note that the actual modeling formalism used to describe the models we compose together can take many forms, including stochastic extensions to Petri nets, stochastic process algebras, and "buckets and balls" models, among others. Our intent is not to create yet another modeling formalism, but simply to specify a simple model type that allows us to describe our technique. In reality, it will work with any discrete-event formalism that has the characteristics described below, including composed models with constituent models expressed in different formalisms.

**Definition 4.1** *A model* $\mathsf{M}$ *is an 8-tuple* $(V, V_{\tilde{s}}, V_s, A, s^{\text{ini}}, \delta, w, prio)$ *in which*

- $V$ *is a finite, non-empty set of SVs.* $D_v$ *is the set of possible values* $v \in V$ *can take and* $\times_{v \in V} D_v$ *is the set of all possible states of* $\mathsf{M}$.
- $V_{\tilde{s}}$ *is a set of* shared *SVs* ($V_{\tilde{s}} \subseteq V$).
- $V_s$ *is a set of* exported shared *SVs* ($V_s \subseteq V_{\tilde{s}}$).
- $A$ *is a finite, non-empty set of actions.*
- $s^{\text{ini}} \in \times_{v \in V} D_v$ *is the initial state.*
- $\delta : A \times (\times_{v \in V} D_v) \rightarrow (\times_{v \in V} D_v)$ *is the partially defined next state function that specifies a successor state for a given action and state.*
- $w : A \times (\times_{v \in V} D_v) \rightarrow \mathbb{R}^{\geq 0}$ *defines a non-negative weight for an action given the current state.*
- $prio : A \rightarrow \mathbb{N}_0$ ($\mathbb{N}_0$ *is the set of non-negative integers) defines a priority for an action using a finite subset of* $\mathbb{N}$.

*For ease of notation,* $\delta(a, s)$ *and* $w(a, s)$ *are denoted by* $\delta_a(s)$ *and* $w_a(s)$*, respectively.* ■

Note that we do not impose restrictions on $\delta$ and $w$, as is typically done for formalisms using action synchronization. For instance, action synchronization requires that enabling conditions and state changes of synchronized actions be conjunctions of local conditions and local effects, e.g., the requirement called "product-form" decomposition in [29]. Since we compose models by sharing state variables, we can allow $\delta_a(s)$ to be defined in a non-decomposable, rather arbitrary manner; e.g., $\delta_a(s)$ may be defined only for states where $\sum_{v \in V} s_v \geq \alpha$, for some constant $\alpha$. We compose models by sharing SVs. There are two ways to do so. If $\mathsf{M}$ itself consists of submodels and results from some composition of those submodels (composition operators will be defined below) then $V_{\tilde{s}}$ contains SVs that are shared among those submodels. In addition, if $\mathsf{M}$ is subject to composition itself, then certain SVs of $\mathsf{M}$ may be shared with other models; set $V_s$ identifies those externally shared SVs within $V_{\tilde{s}}$. The usefulness of subsets $V_{\tilde{s}}$ and $V_s$ of $V_{\tilde{s}}$ will become more clear after composition operators are defined below.

We will limit ourselves to consideration of models whose behaviors are Markov processes by enforcing the following two restrictions. 1) For any state $s$ and action $a$, $\delta_a(s)$ can be defined only if there is no action $a'$ such that $\delta_{a'}(s)$ is defined and $prio(a') > prio(a)$. If $\delta_a(s)$ is defined, we say that $a$ is *enabled* in $s$. 2) If $prio(a) > 0$, then $a$ is called *immediate* and action $a$ takes place (fires) with probability $w_a(s) / \sum_{a' \in E(s)} w_{a'}(s)$, where $E(s) \subseteq A$ denotes the set of actions that are enabled in $s$. If $prio(a) = 0$, then $a$ is *timed* and action $a$ takes place after a delay that is exponentially distributed with rate $w_a(s)$.

Function $\delta$ induces a reachability relation among states. Consider two states $s, s' \in \times_{v \in V} D_v$. We use $s \rightarrow s'$ to denote a transition from $s$ to $s'$ in model $\mathsf{M}$. Relation $\rightsquigarrow$ is the reflexive and transitive closure of $\rightarrow$. In other words, $s \rightsquigarrow s'$ and we say $s'$ is *reachable* from $s$, if and only if $s = s'$ or there is a sequence of one or more transition(s) $s \rightarrow \cdots \rightarrow s'$. A state $s$ is *vanishing* if there is an action in $E(s)$ that has a priority greater than 0. A *tangible* state is a state that is not vanishing. Starting from the initial state $s^{\text{ini}}$, the relation $\rightsquigarrow$ results in the *observable* state space of $\mathsf{M}$. We restrict ourselves to models that have a finite

observable state space starting from a tangible initial state $s^{\text{ini}}$ and have a structure that allows us to perform on-the-fly elimination of vanishing states. The tangible reachable state space, or simply state space, $\mathcal{S}$ of model $\mathsf{M}$ is defined as $\mathcal{S} = \{s | s^{\text{ini}} \rightsquigarrow s, \text{ and } s \text{ is tangible}\}$.

The transition rate matrix of the underlying CTMC is $\mathbf{R}$, where $\mathbf{R}(s, s')$ $(s, s' \in \mathcal{S})$ gives the sum of rates of all timed actions whose firing leads from $s$ to $s'$, possibly including a subsequent sequence of immediate actions whose probabilities are multiplied by the rate of the initial timed action. In the following, we will focus mainly on using MDs to represent $\mathbf{R}$ rather than $\mathbf{Q} = \mathbf{R} - rs(\mathbf{R})$ since the derivation of $rs(\mathbf{R})$ is straightforward.

In order to build models of complete systems from smaller and simpler models, we define two composition operators, "join" and "replicate," which are based on sharing of the SVs of the models on which they are defined [82]. The *join* operator combines a number of (possibly non-identical) models by sharing a subset of their SVs, while the *replicate* operator combines a number of copies (replicas) of the same model by sharing the same subset of all of the models' SVs. The definition of "join" uses the notion of *substate* $s_W$, the projection of $s$ on a set of state variables $W \subseteq V$. Also, $s(v)$ is the value of state variable $v$ in state $s$ $(v \in V)$.

**Definition 4.2** *The* join *operator* $\mathsf{J}(V_J, \mathsf{M}_1, \ldots, \mathsf{M}_n)$ *over models* $\mathsf{M}_j = (V_j, V_{\tilde{s}j}, V_{sj}, A_j, s_j^{\text{ini}}, \delta_j, w_j, prio_j)$, $j \in \{1, \ldots, n\}$ *with* $V_J \subseteq \cup_{j=1}^n V_{sj}$ *yields a new model* $\mathsf{M} = (V, V_{\tilde{s}}, V_s, A, s^{\text{ini}}, \delta, w, prio)$ *with*

- $V = \cup_{j=1}^n V_{sj} \cup \uplus_{j=1}^n V_j \backslash V_{sj}$,[2] *where an appropriate renaming of SVs in* $V_j \backslash V_{sj}$ *ensures unique names such that the union is over disjoint sets, and where* $\cup_{j=1}^n V_{sj}$ *means that SVs with the same names are indeed superimposed.*
- $V_{\tilde{s}} = \cup_{j=1}^n V_{sj}$.
- $V_s = V_J$.
- $A = \uplus_{j=1}^n A_j$, *where an appropriate renaming of actions in* $A_1, \ldots, A_n$ *ensures that the union is over disjoint sets.*

[2] $\uplus$ denotes the union of disjoint sets.

- $s^{\text{ini}}(v) = s_j^{\text{ini}}(v)$ *if* $v \in V_j \backslash V_{sj}$, *and* $s^{\text{ini}}(v) = max_j\ s_j^{\text{ini}}(v)$ *if* $v \in V_{\tilde{s}}$.
- $\delta_a(s) = s'$ *if there exists* $j$ *such that* $a \in A_j$, $\delta_{j,a}(s_{V_j}) = s'_{V_j}$, *and* $s_{V \backslash V_j} = s'_{V \backslash V_j}$.
- $w_a(s) = \lambda$ *if there exists* $j$ *such that* $a \in A_j$ *and* $w_{j,a}(s_{V_j}) = \lambda$.
- $prio(a) = prio_j(a)$ *for all* $j$ *and all* $a \in A_j$.

*We call* $\mathsf{M}_1, \ldots, \mathsf{M}_n$ *the* children *of model* $\mathsf{M}$. ■

We now more precisely identify the role of $V_{\tilde{s}}$ and $V_s$ in $\mathsf{M}$. Elements of $V_{\tilde{s}}$ are SVs shared among the children of $\mathsf{M}$. Suppose model $\mathsf{M}$ has two children $\mathsf{M}_j$ and $\mathsf{M}_{j'}$, each of which has an SV $x$. If $x \in V_{\tilde{s}}$, then $\mathsf{M}$ contains a single SV $x$ shared by $\mathsf{M}_j$ and $\mathsf{M}_{j'}$. On the other hand, if $x \notin V_{\tilde{s}}$ then $x$ is renamed in $\mathsf{M}_j$ and $\mathsf{M}_{j'}$ (e.g., as $x_j$ and $x_{j'}$) such that $\mathsf{M}$ contains two different SVs. Furthermore, if $\mathsf{M}$ itself is used as a child in a subsequent join operator, only the SVs in $V_s$ are visible and can be shared with other children of that join operator.

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

**Definition 4.3** *The* replicate *operator* $\mathsf{R}_n(V_J, \mathsf{M})$ *yields a new model* $\mathsf{M}' = \mathsf{J}(V_J, \mathsf{M}_1, \ldots, \mathsf{M}_n)$ *with* $\mathsf{M}_j = \mathsf{M}$ *for all* $j \in \{1, \ldots, n\}$ *and* $V_J \subseteq V_{s\mathsf{M}}$. *We call* $\mathsf{M}$ *the* child *of model* $\mathsf{M}'$, *and* $n$ *the* cardinality *of the operator.* ■

The replicate operator is a special case of the join operator, in which all children are identical; for that reason, it exhibits desirable properties with respect to the lumpability of the CTMC that its resulting model generates. We will precisely state and prove these properties in Section 4.2.

Note that the set of models is closed under the join and replicate operators, meaning that the result of each of the operators is a model itself, and therefore can be a child of another join or replicate operator. This property enables us to build composed models that are hierarchical. Such composed models require a starting set of "atomic" models that act

as building blocks. Atomic models are built without the use of replicate or join operators and have $V_{\tilde{s}} = V_s$ since there is no reason to have shared SVs that are not externally visible. For analysis of a single atomic model as such, classical CTMC analysis applies. Hence, in the following, we are interested only in composed models that contain at least one join or replicate operator.

For a composed model that is given in terms of possibly nested join and/or replicate operators, we call each occurrence of an atomic model or the result of each occurrence of an operator a *component*. Note that every component is a model. For a model that contains $m$ components we can define an *index* chosen from $\{1, 2, \ldots, m\}$ over the components of a term from left to right after expanding replicate operators into join operators. We use $RC$ and $AM$ to denote the set of all component indices that correspond to replicate operators and atomic models, respectively. For example,

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

obtains indices as in

$$= \mathsf{J}_1(V_J', \mathsf{J}_2(V_J, \mathsf{M}_3, \mathsf{M}_4), \mathsf{J}_5(V_J, \mathsf{M}_6, \mathsf{M}_7))$$

where $m = 7$, $RC = \{1\}$, the farthest-left join operator corresponds to the component with index 1, the next-to-farthest-left join operator has index 2, and the last component is $\mathsf{M}''$ to the right with index 7. Obviously, $V_J$ and $V_J'$ do not receive indices, because they are not components. In the rest of the chapter, the set of SVs, the set of shared SVs, the set of exported shared SVs, and the set of actions of component $c$ are respectively denoted by $V_c$, $V_{\tilde{s}c}$, $V_{sc}$, and $A_c$, and the set of SVs of model $\mathsf{M}$ is denoted by $V$. In the following, we consider only the non-trivial case $m > 1$. The motivation for this indexing scheme is that we wish to partition the set of SVs $V$ into $m$ disjoint subsets as follows. If component $c$ corresponds to a join operator, then $\mathcal{V}_c = V_{\tilde{s}c} \backslash V_{Jc}$. If component $c$ is an atomic model, then $\mathcal{V}_c = V_c \backslash V_{sc}$. The partition is denoted by $\Pi = \{\mathcal{V}_1, \ldots, \mathcal{V}_m\}$, and we call $\mathcal{V}_c$'s $(1 \leq c \leq m)$

the *classes* of $\Pi$.[3]

For any component $c$, we can define an injective mapping $g_c : \times_{v \in \mathcal{V}_c} D_v \rightarrow \mathbb{N}_0$ that gives an index number to any setting of SVs in $\mathcal{V}_c$. Since we consider only models with finite state spaces, the domain of $g_c$ is finite. Clearly, many such mappings exist. At this point the only condition on the mapping is that it be injective such that any state $s = (s_1, \ldots, s_m) \in \times_{c=1}^{m}(\times_{v \in \mathcal{V}_c} D_v)$ of a model, where $s_c = s_{\mathcal{V}_c}$, has a unique representation as a vector $v = (g_1(s_1), \ldots, g_m(s_m)) \in \mathbb{N}_0^m$. The $c$th component of the vector $v$, which is denoted by $v_c$, is in fact the index of substate $s_{\mathcal{V}_c}$. To simplify our notation, we will overload the symbol $s$ and use it to represent a state, a state index, and a vector of substates, whenever it is clear from the context. Similarly, we overload $s_c$ to represent a substate and also the corresponding substate index $g_c(s_c)$.

Consider an action $a \in A_c$ of an atomic component $c$. By definition, $a$ may depend not only on SVs in $\mathcal{V}_c$ but also on shared SVs in other $\mathcal{V}$ sets. The partition $\Pi$ determines $I_c$, the set of indices of $\mathcal{V}$ sets in which the SVs in $V_c$ are encoded. In particular, $I_c = \{c' |\ \mathcal{V}_{c'} \cap V_c \neq \emptyset\}$. Therefore, $V_c \subseteq \cup_{c' \in I_c} \mathcal{V}_{c'}$.

The set $I_c$ gives the set of substates that action $a$ may affect or by which it becomes affected. In other words, action $a$ may change only the substates whose indices are given in $I_c$, and the changes in those substates depend only on themselves (and not on any other substates). More formally, $\forall\, s, s' \in \mathbb{N}_0^m, a \in A_c$:

$$\delta_a(s) = s' \Rightarrow \forall\, i \notin I_c : s_i = s'_i \tag{4.1}$$

$$(\delta_a(s) = s', \delta_a(\bar{s}) = \bar{s}', \forall\, i \in I_c : s_i = \bar{s}_i) \Rightarrow \bar{s}'_i = \begin{cases} s'_i & i \in I_c \\ \bar{s}_i & i \notin I_c \end{cases} \tag{4.2}$$

[3]Depending on the composed model, some of the classes of $\Pi$ may be empty. For the sake of simplicity of the presentation, we assume that all classes are non-empty. However, the degenerate cases are addressed in our implementation.

Observe that if an action $a$ is not in the set of actions of a replicate model $c$ (i.e., $a \notin A_c$) then there exists an atomic component $c' \in AM$ such that $c' \notin \tau_c$ and $a \in A_{c'}$. In that case, action $a$ does not affect or become affected by any substate $s_i$ ($i \in \tau_c$), that is, $I_{c'} \cap \tau_c = \emptyset$. If we consider all actions not in $A_c$, we conclude

$$\forall\, c' \in AM,\ c' \notin \tau_c :\ I_{c'} \cap \tau_c = \emptyset \tag{4.3}$$

## 4.2 Lumping Properties of the Replicate Operator

An important property of the replicate operator is that it yields a behavior that enables lumping on the associated CTMC of a model [82]. The ordinary lumping property of replicate operators has been discovered and proved in [80]. In [60], Obal proves ordinary lumpability for models that have a general type of symmetry (covering the replicate operator as a special case). In this section, we will cover both ordinary and exact lumpability for MRPs that are based on models that we described in Section 4.1. Our formulation is similar to that of [60] in that the notion of state permutation has been used to determine state equivalence. Therefore, our approach is also extensible to general types of symmetry. However, we discuss only symmetries that are due to the replicate operator.

The idea emanates from the observation that a replicate operator constructs a model that has identical copies of another given model. For example, consider a model $\mathsf{M} = \mathsf{R}_3(V_J, \mathsf{M}')$, where $\mathsf{M}'$ is an atomic model. When unfolded as described above, we will have $\mathsf{M} = \mathsf{J}(V_J, \mathsf{M}', \mathsf{M}', \mathsf{M}')$. $\mathsf{M}$ has 4 components, and hence, any one of its states can be represented as a 4-tuple $(s_1, s_2, s_3, s_4)$ in which $s_2$, $s_3$, and $s_4$ correspond to the three replicas. Now consider two states $s = (2, 1, 1, 0)$ and $s' = (2, 1, 0, 1)$. In both states, the replicate component is in substate 2, two of the replicas are in state 1, and the other replica is in 0. Therefore, based on the assumptions that we do not need to distinguish the replicas and that they behave identically, the stochastic behaviors of those two "equivalent" states are

the same. In other words, the two states have transitions with rates that are equal to those of other "equivalent" states. The state $s'' = (2, 0, 1, 1)$ is another state that is equivalent to $s$ and $s'$. Notice that $s$, $s'$, and $s''$ can be converted to each other by a permutation on their state representations.

In this section, we are going to formalize the notion we just described and prove that "equivalent" states like the ones above form classes of partitions with respect to which the MRP (constructed upon the CTMC of model $\mathsf{M}$) is lumpable according to Definition 2.2. We do so by first introducing some notations and a number of definitions that help us define $R_c$, which is the equivalence relation defined on $\mathcal{S}$ and induced by a replicate component $c \in RC$, and its corresponding partition $\mathcal{P}_c$. We then compose the equivalence relations of all replicate components (all $R_c$'s) to construct another equivalence relation $R$ (with a corresponding partition $\mathcal{P}$) that relates two states if they are related via an arbitrary number of constituent equivalence relations ($R_c$'s). Finally, we prove that an MRP defined based on $\mathcal{S}$ and $\mathbf{Q}$ of the model (with some conditions) is ordinarily and exactly lumpable with respect to all $\mathcal{P}_c$'s, and more importantly, with respect to $\mathcal{P}$, according to Definition 2.2.

For a replicate operator $\mathsf{R}_{n_c}(V_J, \mathsf{M})$ with component index $c \in RC$ and cardinality $n_c$, let $l_c$ be the number of component indices used for a single replica of $\mathsf{M}$. Then, by construction, all component indices in $\tau_c = \{c, c+1, \ldots, c+n_c l_c\}$ are associated with $\mathsf{R}_{n_c}(V_J, \mathsf{M})$.

### 4.2.1 $c$-State Permutations and $RC$-State Permutations

We define a *permutation* $p$ to be a bijective function $p : \{1, \ldots, k\} \to \{1, \ldots, k\}$ for a given $k \in \mathbb{N}$. $p_I(i) = i$ for all $i \in \{1, \ldots, k\}$ is called the *identity* permutation. If two permutations $p$ and $p'$ are both defined on the set $\{1, \ldots, k\}$, then their composition is denoted by $p \circ p'$ and defined as $p \circ p'(i) = p(p'(i))$. The set of all permutations on the set $\{1, \ldots, k\}$ with the composition operator form a group in which $p_I$ is the identity element and $p^{-1}$ is the inverse of $p$.

A *c-permutation* $p_c$ corresponding to a replicate component $c \in RC$ is a permutation

with $k$ equal to $n_c$, the cardinality of the replicate component. A $c$-permutation permutes the replicas of a replicate operator. A *state permutation sp* is a permutation with $k = m$; it permutes the substates of a state $s = (s_1, \ldots, s_m)$. The application of a state permutation $sp$ on a state $s = (s_1, \ldots, s_m)$ is denoted by $\alpha(sp, s) = (s_{sp(1)}, \ldots, s_{sp(m)})$. Note that $\alpha(sp, s) = s'$ is equivalent to $\alpha(sp^{-1}, s') = s$. We can now define a special state permutation on which the definition of $R_c$ is based.

**Definition 4.4** *For any replicate component c, a* $c$-state permutation $sp_c$ *is a state permutation such that*

$$\begin{cases} sp_c(i) = i & \text{if } \begin{array}{l} i \leq c \ \ or \ \ i > c + n_c l_c \\ (or \ equivalently \ i \notin \tau_c) \end{array} \\ sp_c(c + (k-1)l_c + j) = c + (p_c(k) - 1)l_c + j & \text{if } \begin{array}{l} 1 \leq k \leq n_c \ \ and \ \ 1 \leq j \leq l_c \\ (c + (k-1)l_c + j) \in \tau_c) \end{array} \end{cases}$$

*in which* $p_c$ *is a given c-permutation.* ■

Observe that

- the second condition covers all values of $i \in \tau_c$.
- $sp_c(i) \in \tau_c$ for all $i \in \tau_c$.
- for a given $c \in RC$, there is a one-to-one correspondence between the set of $c$-permutations and the set of $c$-state permutations.
- $\alpha(sp_c, s)$ is in fact a state derived from $s$ through permutation of the children of replicate component $c$.

In the following, we prove that for a given $c$, the set of $c$-state permutations with the composition operator form a group.

**Lemma 4.1** *For a given c, the set of c-state permutations with the composition operator form a group.*

**Proof.**

a. Associativity: Trivial to see from the definition of the operator.

b. Identity: The $c$-state permutation corresponding to the identity $c$-permutation is the identity permutation.

c. Inverse: Suppose $p_c$ is a $c$-permutation and $sp_c$ is its corresponding $c$-state permutation. We prove that $sp_c^{-1}$ is equal to the $c$-state permutation corresponding to the $c$-permutation $p_c^{-1}$. Suppose $sp'_c$ is the $c$-state permutation corresponding to $p_c^{-1}$. We will prove that $sp'_c(sp_c(i)) = i$ for all $i \in \{1, \ldots, m\}$. Similarly, $sp_c(sp'_c(i)) = i$ can be proven, and thus, $sp'_c = sp_c^{-1}$. We have:

$$\begin{aligned} &sp'_c(i) = i && \text{if} \quad i \leq c \quad \text{or} \quad i > c + n_c l_c \\ &sp'_c(c + (k-1)l_c + j) = c + (p_c^{-1}(k) - 1)l_c + j && \text{if} \quad 1 \leq k \leq n_c \quad \text{and} \quad 1 \leq j \leq l_c \end{aligned}$$

If $i \notin \tau_c$, then $sp'_c(i) = i$ and $sp_c(i) = i$, and therefore, $sp'_c(sp_c(i)) = i$. Otherwise, $i = c + (k-1)l_c + j$ where $1 \leq k \leq n_c$ and $1 \leq j \leq l_c$. Therefore, $sp'_c(sp_c(i)) = sp'_c(c + (p_c(k) - 1)l_c + j) = c + (p_{c,1}^{-1}(p_c(k)) - 1)l_c + j = c + (k-1)l_c + j = i$.

d. Closure: suppose $p_{c,1}$ and $p_{c,2}$ are two $c$-permutations and $sp_{c,1}$ and $sp_{c,2}$ are their corresponding $c$-state permutations, respectively. We prove that $sp_c = sp_{c,1} \circ sp_{c,2}$ is equal to the $c$-state permutation corresponding to the $c$-permutation $p_c = p_{c,1} \circ p_{c,2}$. Suppose $sp'_c$ is the $c$-state permutation corresponding to $p_c$. We will prove that $sp'_c(i) = sp_c(i)$ for all $i \in \{1, \ldots, m\}$, and thus, $sp'_c$ and $sp_c$ are actually the same permutations. We have:

$$\begin{aligned} &sp'_c(i) = i && \text{if} \quad i \notin \tau_c \\ &sp'_c(c + (k-1)l_c + j) = c + (p_{c,1} \circ p_{c,2}(k) - 1)l_c + j && \text{if} \quad 1 \leq k \leq n_c \quad \text{and} \quad 1 \leq j \leq l_c \end{aligned}$$

If $i \notin \tau_c$, then $sp_c(i) = i = sp'_c(i)$. Otherwise, $sp_c(i) = sp_{c,1}(sp_{c,2}(i)) = sp_{c,1}(c + (p_{c,2}(k) - 1)l_c + j) = c + (p_{c,1}(p_{c,2}(k)) - 1)l_c + j = sp'_c(i)$. ■

**Definition 4.5** *An RC-*state permutation $sp_{RC}$ *is a composition of a sequence of* $c_k$*-state permutations where* $i \geq 1$, $k \in \{1, \ldots, i\}$, *and* $c_k \in RC$, *i.e.,* $sp_{RC} = sp_{c_i} \circ \ldots \circ sp_{c_1}$.

Verify that the set of $RC$-state permutations with the composition operator forms a group. The next step is to define the equivalence relations $R_c$ and $R$ that give us the partitions with respect to which the CTMC of a model is ordinarily and exactly lumpable.

### 4.2.2 Equivalence Relations $R_c$ and $R$

**Definition 4.6** *For any* $c \in RC$, *the relation* $R_c \subseteq \mathcal{S} \times \mathcal{S}$ *is defined such that* $(s, s') \in R_c$ *if and only if there exists a c-state permutation* $sp_c$ *such that* $\alpha(sp_c, s) = s'$. ■

**Theorem 4.1** *For any* $c \in RC$, $R_c$ *is an equivalence relation.*

**Proof.**

a. Reflexivity: For any $s \in \mathcal{S}$, $\alpha(p_I, s) = s$. Therefore, $(s, s) \in R_c$.

b. Symmetry: Assume $(s, s') \in R_c$. Then, there exists a $c$-state permutation $sp_c$ such that $\alpha(sp_c, s) = s'$, and hence $\alpha(sp_c^{-1}, s') = s$. By Lemma 4.1, $sp_c^{-1}$ is a $c$-state permutation. Hence, $(s', s) \in R_c$.

c. Transitivity: Assume $(s, s'), (s', s'') \in R_c$. Then, there exist two $c$-state permutations $sp_c$ and $sp'_c$ such that $\alpha(sp_c, s) = s'$ and $\alpha(sp'_c, s') = s''$. Therefore, $\alpha(sp'_c \circ sp_c, s) = s'$, in which, by Lemma 4.1, $sp_c \circ sp'_c$ is a $c$-state permutation. Hence, $(s, s'') \in R_c$. ■

In a given model $\mathsf{M}$, we have $|RC|$ replicate components, and therefore $|RC|$ equivalence relations $R_c$. Each state $s \in \mathcal{S}$ has a number of equivalent states in each of those equivalence relations. We would like to build another equivalence relation $R$ in which two states $s$ and

$s'$ are equivalent if they are "connected" through a number of equivalent states in any set of $R_c$'s. For example, let $s, s', t, t' \in \mathcal{S}$ and $c, c', c'' \in RC$ such that $(s, s') \in R_c$, $(s', t) \in R_{c'}$, and $(t, t') \in R_{c''}$. We would like to build $R$ such that all states $s, s', t, t'$ are equivalent. More formally, we define the relation $R$ by combining all equivalence relations implied by the replicate operator as follows:

**Definition 4.7** *The relation $R \subseteq \mathcal{S} \times \mathcal{S}$ is defined such that $(s, s') \in R$ if and only if there exists an RC-state permutation $sp_{RC}$ such that $s' = \alpha(sp_{RC}, s)$.*

**Theorem 4.2** *$R$ is an equivalence relation.*

**Proof.**

a. Reflexivity: for any $s \in \mathcal{S}$, $\alpha(p_I, s) = s$. Therefore, $(s, s) \in R$.

b. Symmetry: Assume $(s, s') \in R$. Then, $s' = \alpha(sp_{RC}, s)$ for some $RC$-state permutation $sp_{RC}$. Therefore, $s = \alpha(sp_{RC}^{-1}, s')$, in which $sp_{RC}^{-1}$ is itself an $RC$-state permutation. Therefore, $(s', s) \in R$.

c. Transitivity: Assume $(s, s'), (s', s'') \in R$. Then, $s' = \alpha(sp_{RC}, s)$ and $s'' = \alpha(sp'_{RC}, s')$ for some $RC$-state permutations $sp_{RC}$ and $sp'_{RC}$. Therefore, $s'' = \alpha(sp'_{RC} \circ sp_{RC}, s)$, in which $sp'_{RC} \circ sp_{RC}$ is itself an $RC$-state permutation. Therefore, $(s, s') \in R$. ■

Since $R_c$ (resp., $R$) is an equivalence relation, it induces a partition $\mathcal{P}_c$ (resp., $\mathcal{P}$) on $\mathcal{S}$. Based on Definition 4.6 (resp., Definition 4.7) any state of $\mathcal{S}$ can be transformed to any of its equivalent states, i.e., a state in the same class of $\mathcal{P}_c$ (resp., $\mathcal{P}$), by a $c$-state permutation (resp., $RC$-state permutation).

### 4.2.3 Ordinary and Exact Lumpability

We also would like to show that two equivalent states in a class of $\mathcal{P}_c$ or $\mathcal{P}$ behave exactly the same. More specifically, we would like to prove that equivalent states have transitions with the same rate to (other) equivalent states.

**Lemma 4.2** *Consider a model* $\mathsf{M} = (V, V_{\tilde{s}}, V_s, A, s^{\text{ini}}, \delta, w, prio)$*,* $c \in RC$*, a c-state permutation* $sp_c$*,* $a \in A$*, and* $s, s' \in \mathbb{N}_0^m$*. If* $\delta_a(s) = s'$*, then there exists an action* $a' \in A$ *such that* $prio(a') = prio(a)$*,* $w_{a'}(\alpha(sp_c, s)) = w_a(s)$*, and* $\delta_{a'}(\alpha(sp_c, s)) = \alpha(sp_c, s')$*.*

**Proof.** The idea of the proof is simple, and it is based on Eqs. (4.1), (4.2), and (4.3) and Definitions 4.2 and 4.3. The seeming complexity comes from the fact that the proof is primarily involved with different substates of $s$ and $s'$; that makes the notation appear heavy. There are two cases depending on whether $a \in A_c$ or not.

a. $a \notin A_c$: Let $a' = a$. We immediately conclude that $prio(a') = prio(a)$. Moreover, note that $sp_c$ permutes only substates with indices in $\tau_c$ and that the value of $w_a$ does not depend on those substates because $a \notin A_c$. Therefore, $w_{a'}(\alpha(sp_c, s)) = w_a(\alpha(sp_c, s)) = w_a(s)$.

   We further need to prove that $\delta_a(\alpha(sp_c, s)) = \alpha(sp_c, s')$. Let $t' = \delta_a(\alpha(sp_c, s))$. Since $a \notin A_c$, then there exists $c' \in AM$ such that $a \in A_{c'}$ and $c' \notin \tau_c$. According to Eq. (4.3), $I_{c'} \cap \tau_c = \emptyset$. Therefore, for all $i' \in I_{c'}$, we have $i' \notin \tau_c$, and by Definition 4.4, $i' = sp_c(i')$. Using Eq. (4.2) and replacing $c$, $\bar{s}$, and $\bar{s}'$ by $c'$, $\alpha(sp_c, s)$, and $t'$, we have:

$$t'_i = \begin{cases} s'_i & i \in I_{c'} \\ (\alpha(sp_c, s))_i = s_{sp_c(i)} & i \notin I_{c'} \end{cases} \tag{4.4}$$

   We will show that $t' = \alpha(sp_c, s')$ by proving that $\forall\, i\ t'_i = (\alpha(sp_c, s'))_i = s'_{sp_c(i)}$. There are three cases:

   (a) $i \in \tau_c$: By Definition 4.4, we have $sp_c(i) \in \tau_c$, and by Eq. (4.3), we have $sp_c(i) \notin I_{c'}$; using Eq. (4.1), we conclude that $s_{sp_c(i)} = s'_{sp_c(i)}$. Since $i \in \tau_c$ then $i \notin I_{c'}$, and according to Eq. (4.4), $t'_i = s_{sp_c(i)}$, and hence, $t'_i = s'_{sp_c(i)}$.

   (b) $i \notin \tau_c$ and $i \in I_{c'}$: By Eq. (4.4), we have $t'_i = s'_i$, and by $i \notin \tau_c$ and Definition 4.4, we have $i = sp_c(i)$. Therefore, $t'_i = s'_{sp_c(i)}$.

(c) $i \notin \tau_c$ and $i \notin I_{c'}$: Similarly, we have $t'_i = s_{sp_c(i)} = s_i$. Using Eq. (4.1), we have $s_i = s'_i = s'_{sp_c(i)}$. Thus, $t'_i = s'_{sp_c(i)}$.

b. $a \in A_c$: Let $\mathsf{M}_c$ be the model represented by the replicate operator, $\mathsf{M}_{c,i}$ be its $i$th child ($1 \leq i \leq n_c$), and $A_{c,i}$ be the set of actions of $\mathsf{M}_{c,i}$. $a \in A_c$ means that there exists an $i$ such that $a \in A_{c,i}$. Therefore, there exists an action $a' \in A_{c,j}$ ($j = p_c(i)$) in model $\mathsf{M}_{c,j}$ that corresponds to $a$ in model $\mathsf{M}_{c,i}$. From Definition 4.2, we immediately conclude that $prio(a') = prio(a)$.

To prove $\delta_{a'}(\alpha(sp_c, s)) = \alpha(sp_c, s')$, assume that $u = s_{V_c}$ and $u' = s'_{V_c}$, where $V_c$ is the set of SVs of $\mathsf{M}_c$. In other words, $u$ is the projection of the state $s$ of the overall model on the SVs of the replicate model. Therefore, $\delta_a(s) = s'$ in the context of $\mathsf{M}$ implies $\delta_a(u) = u'$ in the context of $\mathsf{M}_c$, because $a$ does not affect any variable in $V \backslash V_c$. By Definitions 4.2 and 4.3, we have $\delta_{i,a}(u_{V_{c,i}}) = u'_{V_{c,i}}$ in the context of $\mathsf{M}_{c,i}$ and $u_{V \backslash V_{c,i}} = u'_{V \backslash V_{c,i}}$ (where $V_{c,i}$ is the set of SVs of $\mathsf{M}_{c,i}$) because $a \in A_{c,i}$. By using permutation $p_c$ to permute the children of the replicate model, we obtain

$$\delta_{p_c(i),a'}(x_{V_{c,p_c(i)}}) = x'_{V_{c,p_c(i)}} \text{ in the context of } \mathsf{M}_{c,p_c(i)} \text{ and } x_{V \backslash V_{c,p_c(i)}} = x'_{V \backslash V_{c,p_c(i)}}$$

where $x = t_V$, $x' = t'_V$, $t = \alpha(sp_c, s)$, and $t' = \alpha(sp_c, s')$. Equivalently, $\delta_{j,a'}(x_{V_{c,j}}) = x'_{V_{c,j}}$ in the context of $\mathsf{M}_{c,j}$ and $x_{V \backslash V_{c,j}} = x'_{V \backslash V_{c,j}}$. Therefore, using Definitions 4.2 and 4.3, we have $\delta_{a'}(x) = x'$ in the context of $\mathsf{M}_c$. Finally, because $a$ does not affect any variable in $V \backslash V_c$, we have $\delta_{a'}(t) = t'$.

Similarly, assume that $w_a(s) = \lambda$. Therefore, we have $w_{c,a}(u) = \lambda$ where $w_c$ is the weight function of $\mathsf{M}_c$. Hence, based on Definitions 4.2 and 4.3, $w_{c,i,a}(u_{V_{c,i}}) = \lambda$ where $w_{c,i}$ is the weight function of $\mathsf{M}_{c,i}$. By permuting the children of the replicate model using permutation $p_c$, we have $w_{c,p_c(i),a'}(u_{V_{c,p_c(i)}}) = \lambda$, and therefore, $w_{c,j,a'}(x_{V_{c,j}}) = \lambda$. By Definitions 4.2 and 4.3, we will have $w_{c,a'}(x) = \lambda$. In the context of $\mathsf{M}$, that implies

$w_{a'}(t) = \lambda$. ■

**Corollary 4.1** *Consider a model* $\mathsf{M} = (V, V_{\tilde{s}}, V_s, A, s^{\text{ini}}, \delta, w, prio)$*, RC-state permutation* $sp_{RC}$*,* $a \in A$*, and states* $s, s' \in N_0^m$*. If* $\delta_a(s) = s'$*, then there exists an action* $a'$ *such that* $\delta_{a'}(\alpha(sp_{RC}, s)) = \alpha(sp_{RC}, s')$*,* $w_{a'}(\alpha(sp_{RC}, s)) = w_a(s)$*, and* $prio(a') = prio(a)$*.*

**Proof.** Use Lemma 4.2 and induction on $i$ where $sp_{RC} = sp_{c_i} \circ \ldots \circ sp_{c_1}$. ■

**Lemma 4.3** *Consider a model* $\mathsf{M}$ *and two states* $s, s', s'' \in \mathbb{N}_0^m$ *such that* $s \to s'$ *and* $s \rightsquigarrow s''$*. Then, for any RC-state permutation* $sp_{RC}$ *and any c-state permutation* $sp_c$ *(*$c \in RC$*),*

*a.* $s$ *is tangible* $\equiv \alpha(sp_{RC}, s)$ *is tangible* $\equiv \alpha(sp_c, s)$ *is tangible.*

*b.* $\alpha(sp_{RC}, s) \to \alpha(sp_{RC}, s')$ *and* $\alpha(sp_{RC}, s) \rightsquigarrow \alpha(sp_{RC}, s'')$*.*

*c.* $\alpha(sp_c, s) \to \alpha(sp_c, s')$ *and* $\alpha(sp_c, s) \rightsquigarrow \alpha(sp_c, s'')$*.*

**Proof.**

a. Corollary 4.1 implies that for each action $a \in E(s)$ there is an action $a' \in E(\alpha(sp_{RC}, s))$ such that $prio(a) = prio(a')$, and vice versa. That means $s$ is tangible if and only if $\alpha(sp_{RC}, s)$ is tangible. Similarly, Lemma 4.2 implies that $s$ is tangible if and only if $\alpha(sp_c, s)$ is tangible.

b. $\alpha(sp_{RC}, s) \to \alpha(sp_{RC}, s')$ follows from Corollary 4.1 and also implies $\alpha(sp_{RC}, s) \rightsquigarrow \alpha(sp_{RC}, s'')$ using induction.

c. Follows from (b) because any $c$-state permutation is also an $RC$-state permutation.

■

Lemma 4.3 shows that reachability and tangibleness are preserved by $c$-state and $RC$-state permutations. In other words, equivalent states have transitions to states that are themselves equivalent. Now, we need to prove that the rates of those transitions are also the same. The proof is given in the following two lemmas (one for $R$ and one for $R_c$).

**Lemma 4.4** *Consider a model* $\mathsf{M}$ *with state space* $\mathcal{S}$*, transition rate matrix* $\mathbf{R}$*, generator matrix* $\mathbf{Q}$*, the partition* $\mathcal{P}$ *(as defined in Definition 4.7), and initial state* $s^{\text{ini}} \in C_0 \in \mathcal{P}$*. If* $C_0 = \{s^{\text{ini}}\}$*, then for all* $s, s' \in \mathcal{S}$*, and all RC-state permutations* $sp_{RC}$*,*

$$\begin{aligned}
&\alpha(sp_{RC}, s) \in \mathcal{S},\ \alpha(sp_{RC}, s') \in \mathcal{S},\\
&\mathbf{R}(s, s') = \mathbf{R}(\alpha(sp_{RC}, s), \alpha(sp_{RC}, s')),\ \textit{and}\\
&\mathbf{Q}(s, s') = \mathbf{Q}(\alpha(sp_{RC}, s), \alpha(sp_{RC}, s'))
\end{aligned}$$

**Proof.** Since $s \in \mathcal{S}$, then $s^{\text{ini}} \rightsquigarrow s$ and $s$ is tangible, and by Lemma 4.3, $\alpha(sp_{RC}, s)$ is tangible too and $\alpha(sp_{RC}, s^{\text{ini}}) \rightsquigarrow \alpha(sp_{RC}, s)$. By the definition of $\mathcal{P}$, we have $\alpha(sp_{RC}, s^{\text{ini}}) \in C_0$. However, $|C_0| = 1$, and hence, $\alpha(sp_{RC}, s^{\text{ini}}) = s^{\text{ini}}$. Therefore, $s^{\text{ini}} \rightsquigarrow \alpha(sp_{RC}, s)$, which leads to $\alpha(sp_{RC}, s) \in \mathcal{S}$. Similarly, $\alpha(sp_{RC}, s') \in \mathcal{S}$. Finally, by Corollary 4.1, $\mathbf{R}(s, s') = \mathbf{R}(\alpha(sp_{RC}, s), \alpha(sp_{RC}, s'))$, since the weight and priority functions all have the same value for states $s$ and $\alpha(sp_{RC}, s)$. $\mathbf{Q}(s, s') = \mathbf{Q}(\alpha(sp_{RC}, s), \alpha(sp_{RC}, s'))$ follows directly from $\mathbf{Q} = \mathbf{R} - rs(\mathbf{R})$. ■

**Lemma 4.5** *Consider a model* $\mathsf{M}$ *with state space* $\mathcal{S}$*, transition rate matrix* $\mathbf{R}$*, generator matrix* $\mathbf{Q}$*,* $c \in RC$*, the partition* $\mathcal{P}_c$ *(as defined in Definition 4.6), and initial state* $s^{\text{ini}} \in C_0 \in \mathcal{P}_c$*. If* $C_0 = \{s^{\text{ini}}\}$*, then for all* $s, s' \in \mathcal{S}$*, and all c-state permutations* $sp_c$*,*

$$\begin{aligned}
&\alpha(sp_c, s) \in \mathcal{S},\ \alpha(sp_c, s') \in \mathcal{S},\\
&\mathbf{R}(s, s') = \mathbf{R}(\alpha(sp_c, s), \alpha(sp_c, s')),\ \textit{and}\\
&\mathbf{Q}(s, s') = \mathbf{Q}(\alpha(sp_c, s), \alpha(sp_c, s'))
\end{aligned}$$

**Proof.** Observe that if $C_0$ is a singleton set, then the class containing $s^{\text{ini}}$ in partition $\mathcal{P}$ will also be a singleton set. Moreover, every $c$-state permutation is also an $RC$-state permutation. Therefore, by Lemma 4.4, the desired result follows. ■

To define an MRP $M$ on a model $\mathsf{M}$ (as defined in Definition 4.1), we need to specify a reward vector $\mathbf{r}$ and an initial distribution probability vector $\boldsymbol{\pi}^{\text{ini}}$, in addition to the state space $\mathcal{S}$ and generator matrix $\mathbf{Q}$ that we have already specified. Since $\mathsf{M}$ explicitly specifies the initial state $s^{\text{ini}}$, we have:

$$\boldsymbol{\pi}^{\text{ini}}(s) = \begin{cases} 1 & s = s^{\text{ini}} \\ 0 & \text{otherwise} \end{cases}$$

Notice that for any $c$-state permutation $sp_c$, $\alpha(sp_c, s^{\text{ini}}) = s^{\text{ini}}$ because by Definition 4.3, each child of the replicate component $c$ has the same initial state and permuting the children does not change the initial state of the replicate component. Therefore, assuming that $C_0$ is the class of $\mathcal{P}_c$ that contains $s^{\text{ini}}$, we have $|C_0| = 1$. Since $\alpha(sp_c, s^{\text{ini}}) = s^{\text{ini}}$ holds for any $c \in RC$ and any $c$-state permutations $sp_c$, $\alpha(sp_{RC}, s^{\text{ini}}) = s^{\text{ini}}$ also holds for any $RC$-state permutation $sp_{RC}$. Therefore, the class of $\mathcal{P}$ that contains $s^{\text{ini}}$ is a singleton set.

The definition of $\mathsf{M}$ does not restrict $\mathbf{r}$. However, for the MRP $M$ to demonstrate the ordinary lumpability property, we place a condition on $\mathbf{r}$. We call $\mathbf{r}$ *$c$-symmetric* ($c \in RC$) if for any $c$-state permutation $sp_c$, we have $\mathbf{r}(s) = \mathbf{r}(\alpha(sp_c, s))$. If $\mathbf{r}$ is $c$-symmetric for all $c \in RC$ then we call it *$RC$-symmetric*. In other words, $\mathbf{r}$ is $RC$-symmetric if for any $RC$-state permutation $sp_{RC}$, we have $\mathbf{r}(s) = \mathbf{r}(\alpha(sp_{RC}, s))$. At the state level, those definitions mean that the rewards for two equivalent states (with respect to relation $R_c$ or $R$) are the same. In terms of the high-level model, they mean that the value of the reward must not distinguish the replicas of a replicate operator.

Before we state and prove the two main theorems of this section, we need one more lemma, which shows that the application of an arbitrary $c$-state permutation (resp., an $RC$-state permutation) to all the states of a class of $\mathcal{P}_c$ (resp., $\mathcal{P}$) generates all the elements of that class. More formally,

**Lemma 4.6** *(a) For any $c \in RC$, class $C \in \mathcal{P}_c$, and a c-state permutation $sp_c$,*

$$\cup_{s \in C}\{\alpha(sp_c, s)\} = C,$$

*i.e., the range of function $\alpha(sp_c, .) : C \to C$ is equal to $C$. (b) For any class $C \in \mathcal{P}$ and an RC-state permutation $sp_{RC}$,*

$$\cup_{s \in C}\{\alpha(sp_{RC}, s)\} = C,$$

*i.e., the range of function $\alpha(sp_{RC}, .) : C \to C$ is equal to $C$.*

**Proof.** To prove (a), verify, using the definition of $\mathcal{P}_c$, that $\alpha(sp_c, .)$ and $\alpha(sp_c^{-1}, .)$ are both injective functions on $C$. Therefore, $\alpha(sp_c, s)$ is also bijective, i.e., injective and surjective. (b) can be proved in the same way. ■

Finally, we have all the tools to prove that the MRP $M$ that we defined on the model $\mathsf{M}$ is ordinarily and exactly lumpable with respect to all partitions $\mathcal{P}_c$ ($c \in RC$) and also with respect to partition $\mathcal{P}$.

**Theorem 4.3** *Consider an MRP $M = (\mathcal{S}, \mathbf{Q}, \mathbf{r}, \boldsymbol{\pi}^{\text{ini}})$ defined on model $\mathsf{M}$, and partition $\mathcal{P}$ (Definition 4.7). Then,*

*a. $M$ is ordinarily lumpable with respect to $\mathcal{P}$ if $\mathbf{r}$ is RC-symmetric.*

*b. $M$ is exactly lumpable with respect to $\mathcal{P}$.*

**Proof.** Consider $C, C' \in \mathcal{P}$ and two (equivalent) states $s, \hat{s} \in C$. Since $s$ and $\hat{s}$ are equivalent, there exists an $RC$-state permutation $sp_{RC}$ such that $\hat{s} = \alpha(sp_{RC}, s)$.

a. $\mathbf{r}(s) = \mathbf{r}(\hat{s})$ because $\mathbf{r}$ is $RC$-symmetric. Moreover,

$$\begin{aligned}
\mathbf{Q}(s, C') &= \sum_{s' \in C'} \mathbf{Q}(s, s') \\
\text{[by Lemma 4.4]} \quad &= \sum_{s' \in C'} \mathbf{Q}(\alpha(sp_{RC}, s), \alpha(sp_{RC}, s')) \\
&= \sum_{s' \in C'} \mathbf{Q}(\hat{s}, \alpha(sp_{RC}, s')) \\
\text{[by Lemma 4.6]} \quad &= \sum_{s'' \in C'} \mathbf{Q}(\hat{s}, s'') = \mathbf{Q}(\hat{s}, C')
\end{aligned}$$

b. If $s = s^{\text{ini}}$, then $s = s^{\text{ini}} = \hat{s}$ because $C$ is a singleton set, and we have, $\boldsymbol{\pi}^{\text{ini}}(s) = \boldsymbol{\pi}^{\text{ini}}(\hat{s}) = 1$. If $s \neq s^{\text{ini}}$, then $\hat{s} \neq s^{\text{ini}}$ and we have $\boldsymbol{\pi}^{\text{ini}}(s) = \boldsymbol{\pi}^{\text{ini}}(\hat{s}) = 0$. Moreover,

$$\begin{aligned}
\mathbf{Q}(C', s) &= \sum_{s' \in C'} \mathbf{Q}(C', s) \\
\text{[by Lemma 4.4]} \quad &= \sum_{s' \in C'} \mathbf{Q}(\alpha(sp_{RC}, s'), \alpha(sp_{RC}, s)) \\
&= \sum_{s' \in C'} \mathbf{Q}(\alpha(sp_{RC}, s'), \hat{s}) \\
\text{[by Lemma 4.6]} \quad &= \sum_{s'' \in C'} \mathbf{Q}(s'', \hat{s}) = \mathbf{Q}(C', \hat{s})
\end{aligned}$$

■

**Theorem 4.4** *Consider an MRP $M = (\mathcal{S}, \mathbf{Q}, \mathbf{r}, \boldsymbol{\pi}^{\text{ini}})$ defined on model $\mathsf{M}$, $c \in RC$, and partition $\mathcal{P}_c$ (Definition 4.6). Then,*

*a. $M$ is ordinarily lumpable with respect to $\mathcal{P}_c$ if $\mathbf{r}$ is c-symmetric.*

*b. $M$ is exactly lumpable with respect to $\mathcal{P}_c$.*

**Proof.** Similar to Theorem 4.3. ■

Partition $\mathcal{P}_c$ and Theorem 4.4 are interesting from the theoretical point of view; we do not (directly) use them to lump the MRP. Instead, we use Partition $\mathcal{P}$ and Theorem 4.3 to lump the MRP in the rest of the chapter and in our implementation.

## 4.3 Symbolic Generation of the Unlumped State Space $\mathcal{S}$

In order to compute performance measures of a composed model, we need to construct a CTMC representing the behavior of the model. Our main goal is to extend the size of composed models that can be handled on a typical computer system by using the structural properties of a model both to reduce the number of states that need to be considered and to compactly represent the states that need to be considered. With that aim, we chose to use MDD and MD data structures, respectively, to represent the set of states and the set of transitions of the CTMC associated with a composed model, and use the structural characteristics of the model to lump equivalent states.

In this section, we will give a detailed description of our new algorithm for symbolic generation of the unlumped state space of a composed model $\mathsf{M}$. We describe the state-space generation (SSG) algorithm that does not take lumping properties into account, and therefore generates the MDD representation of the unlumped state space $\mathcal{S}$. In the next section, we give an algorithm that uses the lumping results we proved in Section 4.2 to lump $\mathcal{S}$ and constructs the MDD representation of $\widetilde{\mathcal{S}}$.

A symbolic SSG algorithm is similar to a traditional one in the sense that both algorithms start with the initial state of the model and keep firing actions until all reachable states have been explored. The difference is that in a traditional algorithm, each time an action is fired only one state is visited, while in a symbolic algorithm, a (potentially large) *set* of states is visited. In our SSG algorithm, we use MDDs to represent sets of states. In order to design an efficient symbolic algorithm for composed models, we identify key structural properties of a model, and based on those properties we determine the "meaning," with respect to the composed model structure, of each level of the MDD. In particular, when we use an MDD to represent the set of states of a model, we use a vector to represent each state of the model. This vector representation is determined by partition $\Pi$ of the set of SVs. More

formally, for each component $1 \leq c \leq m$, level $c$ of the MDD represents substates of the form $s_c$ (enumerating the values of SVs in $\mathcal{V}_c$). In other words, we define $\mathcal{S}_c$, the set of possible values of a level-$c$ node, such that $|\mathcal{S}_c| = |\{s_c | s \in \mathcal{S}\}|$. That also means that level $c$ corresponds to

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

In order to design an efficient state-space generation algorithm, we consider a restricted class of composed models in which all global actions are of the lowest priority, i.e., they are timed actions. There are no other restrictions on how actions are enabled or change state, i.e., $\delta_a(s)$ can be an arbitrary function on its atomic model's SVs. This generality implies that a distinction between acyclic and cyclic dependencies as discussed in [87] does not apply. There is no restriction on local actions. The slight restriction on global actions we do have has two important implications that enable us to design an efficient SSG algorithm: 1) the elimination of vanishing states can take place locally, i.e., in each atomic component, and on the fly, i.e., without storing intermediate vanishing states, and 2) atomic components that share SVs cannot stop one another from proceeding locally. The latter property gives us the ability to use an approach similar to saturation [25] (in firing local actions) and generate a subset of the state space of an atomic component independently from other atomic components as long as the fired actions are independent from the shared SVs of that component, i.e., the actions are local.

```
SSSE(s^ini)
 1  S' := {s^ini}
 2  U := {s^ini}
 3  while U ≠ ∅
 4      LocalSSE(U)
 5      GlobalSSE(U)
 6      U := U − S'
 7      S' := S' ∪ U
 8  return S'
```

Figure 4.1: Pseudocode for the overall symbolic state-space exploration algorithm

### 4.3.1 The Overall Algorithm

We first describe SSSE (Symbolic State-Space Exploration), the algorithm we employ to generate the unlumped state space $\mathcal{S}$ of a composed model (shown in Figure 4.1). SSSE calls two procedures: LocalSSE, which explores the state space by firing local actions, and GlobalSSE, which does the same by firing global actions.

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

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

care of by on-the-fly elimination of vanishing states. GlobalSSE($\mathcal{U}$) adds to $\mathcal{U}$ the set of states that can be reached from any state in $\mathcal{U}$ by the firing of a single global action followed by a (finite) sequence of immediate (local) action firings.

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

### 4.3.2 Firing Local Actions

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

Suppose that a (tangible) substate $s_c$ can lead to a tangible substate $s'_c$ by a sequence of actions in $A_{c,l}$. Hence, if state $(s_1, \ldots, s_{c-1}, s_c, s_{c+1}, \ldots, s_m)$ is reachable, then state $(s_1, \ldots, s_{c-1}, s'_c, s_{c+1}, \ldots, s_m)$ is also reachable. To implement this local state exploration on the MDD, we perform the "saturation" operation on all nodes $u$ in level $c$: $u[s'_c] := u[s_c] \cup u[s'_c]$ for all possible values of $s_c$ and $s'_c$, where $u$ is a non-terminal node and $u[s_c]$ (called a *child* of node $u$) denotes the node to which the $s_c$th pointer of $u$ points. In that operation, values of $s_j$'s ($j \neq c$) are implicit; all state paths that go through node $u$ constitute all states of the form $(s_1, \ldots, s_{c-1}, s_c, s_{c+1}, \ldots, s_m)$.

Figure 4.2(a) shows LocalSSE, which explores the state space using only local actions $A_{c,l}$ of every atomic component $c$. Therefore, LocalSSE iterates through all nodes $u_c$ of levels that correspond to atomic components in depth-first search (DFS) order. For each node $u_c$ that encodes a set of substates of the form $s_c$, it saturates $u_{\text{sat}}$, the node that is to be the saturated version of $u_c$, by calling Saturate($u_{\text{sat}}$) in line 4. Finally, in lines 5-6,

```
LocalSSE(U)
1  foreach non-root node u visited
       in a DFS order starting from
       from node r, root of U
2      c := level of u in the MDD
3      if c ∈ AM
4          u_sat := copy of u
5          Saturate(u_sat)
6          if u ≠ u_sat
7              replace u with u_sat
```

(a) Main pseudocode

```
Saturate(u)
 1  c := level of u in the MDD
 2  Y := {0, ..., |S_c| − 1}
 3  while Y ≠ ∅
 4      pick s_c from Y
 5      foreach s'_c s.t. B*_c(s_c, s'_c) = 1
 6          u' := u[s_c] ∪ u[s'_c]
 7          if u' ≠ u[s'_c]
 8              u[s'_c] := u'
 9              Y := Y ∪ {s'_c}
10      Y := Y − {s_c}
```

(b) Saturation pseudocode

Figure 4.2: Pseudocodes for the local state-space exploration

$u_c$ is replaced by its saturated version $u_{\text{sat}}$.[4] The reason for iterating through all nodes in DFS order is that implementation issues make it necessary for us to ensure that a node is saturated after all its children have been saturated.

Saturate($u$) (shown in Figure 4.2(b)) fires local actions until no further local action firing can add any substate to the set. Lines 3-10 perform the above mentioned saturation operation on $u$ in a "symbolic" manner, i.e., for each $s'_c$, lines 6-8 add all states of the form $(s_1, \ldots, s_{c-1}, s'_c, s_{c+1}, \ldots, s_m)$ to $\mathcal{U}$. Notice that during the saturation operation, we may need to increase the size of $u$ (i.e., the number of its pointers), since we do not know the final value of $|\mathcal{S}_c|$ in advance. The important point is that due to the locality of the actions, we can expand the set of reachable states of the system only by (local) changes to $u$.

Repetitive computations related to local state exploration might occur, since the same substate may be explored many times for different nodes throughout the execution of SSSE. In order to avoid the extra computations, we need an efficient data structure for each atomic component $c$ that stores the reachability relation among substate indices of that component. More formally, we need to know, for every $s_c$, the set of all substate indices $s'_c$ that we can reach from the substate with index $s_c$ by a (finite) sequence of local action firings. We can

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

determine that by computing the reflexive and transitive closure of a square Boolean-valued matrix denoted by $\mathbf{B}_c$. $\mathbf{B}_c$ is defined on the (reachable tangible) state space of $\mathsf{M}_c$, which means that $\mathbf{B}_c$ has $|\mathcal{S}_c|$ rows and columns. $\mathbf{B}_c(s_c, s'_c)$ = true if and only if, starting from substate $s_c$, there is a sequence of local action firings (in which the first is timed and the others, if any, are immediate) that leads to substate $s'_c$. Otherwise, $\mathbf{B}_c(s_c, s'_c)$ = false. Let $\mathbf{B}_c^*$ be the reflexive and transitive closure of $\mathbf{B}_c$. That means $\mathbf{B}_c^*(s_c, s'_c)$ = true if and only if there is a (possibly empty) sequence of local action firings that takes component $c$ from substate $s_c$ to $s'_c$.

Entries of $\mathbf{B}_c$ are updated as we explore the atomic component state space, and computing the transitive closure from scratch is expensive; it takes $\mathcal{O}(|\mathcal{S}_c|^3)$ time for each update of $\mathbf{B}_c$. That excessively large running time is the reason why we use a simple but rather efficient online algorithm given by Ibaraki and Katoh [59] to maintain $\mathbf{B}_c^*$ as we update $\mathbf{B}_c$. Their algorithm takes at most $\mathcal{O}(|\mathcal{S}_c|^3)$ for at most $\mathcal{O}(|\mathcal{S}_c|^2)$ updates to $\mathbf{B}_c$. Notice that $\mathcal{O}(|\mathcal{S}_c|^2)$ is the maximum possible number of updates.

The pseudocode of InsertElementAndUpdate for computing $\mathbf{B}_c^*$, which is based on the algorithm given in [59], is shown in Figure 4.3. InsertElementAndUpdate$(s_c, s'_c)$ sets $\mathbf{B}_c(s_c, s'_c)$ to true and updates $\mathbf{B}_c^*$ accordingly. One way to understand how InsertElementAndUpdate is called by our algorithm is to assume that accessing $\mathbf{B}_c^*(s_c, s'_c)$ in line 5 of Saturate causes a function call if substate $s_c$ has not already been explored. That function explores substate $s_c$ by firing actions in $A_{c,l}$, computes one row of $\mathbf{B}_c$ as defined above, and calls InsertElementAndUpdate for each element of $\mathbf{B}_c$ that is set to true. Note that only $\mathbf{B}_c^*$ is stored in memory during the running time of SSSE and there is no need to store $\mathbf{B}_c$; this is explained here to make the above discussion easier to understand.

### 4.3.3 Firing Global Actions

Figure 4.4(a) shows GlobalSSE, which explores the state space using only global actions $A_{c,g}$ of every atomic component $c$. GlobalSSE iterates through all atomic components

```
InsertElementAndUpdate(s_c, s'_c)
1  for x := 0 to |S_c| − 1
2      if B*_c(x, s_c) and (not B*_c(x, s'_c))
3          for y := 0 to |S_c| − 1
4              B*_c(x, y) := B*_c(x, y) or B*_c(s'_c, y)
```

Figure 4.3: Pseudocode for computing $\mathbf{B}_c^*$ using Ibaraki and Katoh's algorithm

```
GlobalSSE(U)
1  foreach c ∈ AM
2      allocate arrays u_{0..|L_c|}, i_{0..|L_c|}
3      u_0[i_0] := root of U
4      U' := FireAllGlobals(c, 1, u, i)
5      U := U ∪ U'
```

(a)

```
FireAllGlobals(c, k, u_{0..|L_c|}, i_{0..|L_c|})
 1  U' := ∅
 2  foreach node w of level L_c(k)
 3      u_k := w
 4      if MDDConnector(u_{k−1}[i_{k−1}], u_k) ≠ ∅
 5          if (k < |L_c|)  T' := ∅
 6          for j := 0 to |S_{L_c(k)}| − 1
 7              i_k := j
 8              if (k < |L_c|)
 9                  T' := T' ∪ FireAllGlobals(c, k + 1, u, i)
10              else U' := U' ∪ FireOneGlobal(c, u, i)
11          U' := U' ∪ T'
12  return U'
```

(b)

Figure 4.4: Pseudocode of the global state-space exploration procedure

$c \in AM$. For each one, the recursive procedure FireAllGlobals in line 4 generates the set of tangible states that are reachable from states in $\mathcal{U}$ by firing one action in $A_{c,g}$. Finally, in line 5, the states are added to $\mathcal{U}$. The roles of arrays of nodes $u$ and array of substate indices $i$ are described below. They are allocated in GlobalSSE but initialized and used in the recursive calls of FireAllGlobals.

Consider a global action $a \in A_{c,g}$ of an atomic component $c$. As we mentioned in Section 4.2, $I_c$ is the set of indices of $\mathcal{V}$ sets in which the SVs in $V_c$ are encoded. Due to the order we

chose on $\mathcal{V}$ sets, $c' \leq c$ for all $c' \in I_c$. In order to impose an order on the elements of $I_c$, we use $L_c$ to denote the sequence of elements of $I_c$ sorted in ascending order. In other words, if the $k$th element of a sequence $L$ is denoted by $L(k)$, we have $|L_c| = |I_c|$, $L_c(k) \in I_c$ for all $1 \leq k \leq |L_c|$, and $L_c(k) < L_c(k+1)$ for all $1 \leq k < |L_c|$.

It is important to note that, in terms of changes that need to be applied on the MDD, firing a global action in state-sharing composed models is inherently more difficult than firing a synchronizing action in an action-synchronization model, as discussed in [25]. The reason is that in the latter case, the sets of SVs of atomic submodels are disjoint, and due to the product-form behavior [25], the changes that need to be applied on a node $v$ (in the level corresponding to an atomic model) during saturation depend only on the information present in $v$ and the action $a$ to be fired, regardless of whether $a$ is local or synchronizing. However, in the former case, some SVs are shared among atomic models, so that firing a global action $a$ on a node $v$ requires not only the information in $v$ but also the information in other levels of the MDD. That makes the saturation approach inapplicable to the firing of global actions in state-sharing composed models. One may suggest that we could avoid this problem by assigning the shared SVs along with the non-shared ones to one level of the MDD. That approach will not work, because the shared SVs belong to two or more atomic models in the system (corresponding to two or more levels of the MDD).

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

To illustrate this, consider the example in Figure 4.5, in which $|L_c| = 3$. $u_0$ is an imaginary node such that $u_0[i_0]$ is equal to the root of $\mathcal{U}$[5] (line 3 of GlobalSSE). It is used to avoid case-by-case analysis, and thus to simplify the presentation. $u_k$ is a node in level $L_c(k)$ of the MDD for $k \in \{1, \ldots, |L_c|\}$. The left side of the figure shows all paths of the MDD (before the firing of action $a$) that pass through all $u_k$'s. Let $\hat{\mathcal{U}}$ be the set of all states that these paths represent. Let $U_k$ be the MDD connector between $u_{k-1}[i_{k-1}]$ and $u_k$ where $i_k = g_c(s_{L_c(k)})$. FireOneGlobal, which is called by FireAllGlobals (Figure 4.4(b)) generates another MDD that represents the set of states reachable from $\hat{\mathcal{U}}$ by the firing of all actions $a \in A_{c,g}$, that is, $\hat{\mathcal{U}}' = \{(s'_{V_c}, s_{V \setminus V_c}) | s \in \hat{\mathcal{U}}, \delta_a(s) = s', a \in A_{c,g}\}$. Notice that in order to generate $\hat{\mathcal{U}}'$, we do not need to change the nodes in any of the $U_k$'s ($k \in \{1, \ldots, |L_c|\}$), because action $a$ is independent of the SVs encoded in the levels corresponding to $U_k$'s. Instead, FireOneGlobal, for each $k$, 1) makes a copy of each $U_k$, 2) computes $i'_k = g_c(s'_{L_c(k)})$ and creates a new node $u'_k$ and for each enabled action in $A_{c,g}$, and 3) connects all the new nodes as shown on the right side of Figure 4.5 in order to build $\hat{\mathcal{U}}'$. Because of limited space, the pseudocode of FireOneGlobal is not given.

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

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

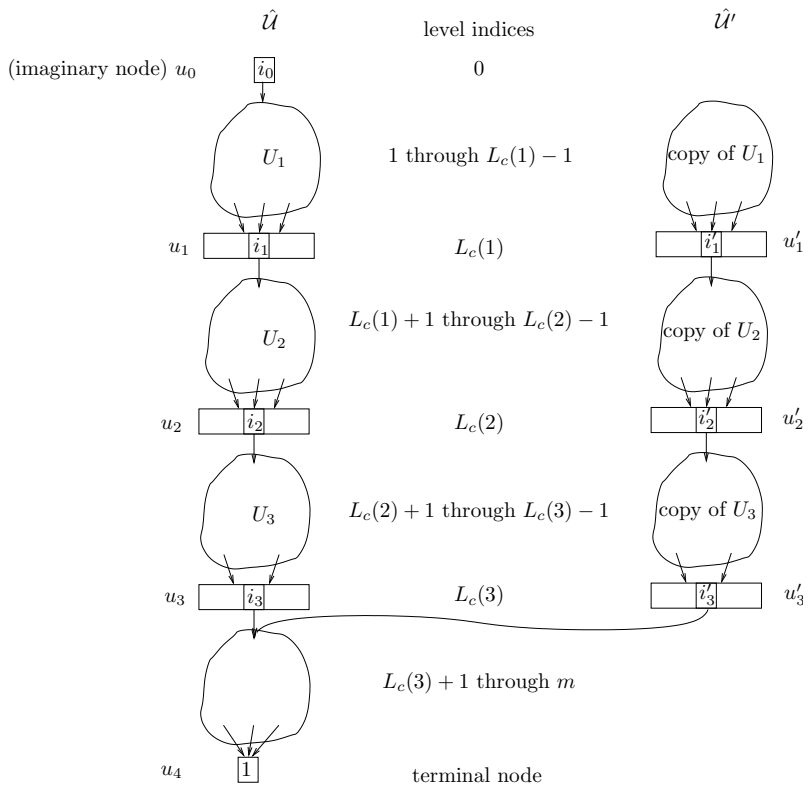



Figure 4.5: Computing the set of next states for global actions using MDD connectors

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

Given $\mathcal{S}$ and a number of small rate matrices (corresponding to each component) that we could build during the SSSE procedure, we could obtain an MD of the unlumped CTMC, and apply known approaches for the MD-based numerical analysis of CTMCs [28]. Part of our goal, however, is to reduce the number of states in the resulting CTMC by lumping it; therefore, as discussed in the next section, we compute $\widetilde{\mathcal{S}}$.

## 4.4 Symbolic Construction of the Lumped State Space $\widetilde{\mathcal{S}}$

In the SSG algorithm described above, we treated the replicate operator as a join operator with identical children, without considering lumping properties. The final step we take is to lump the state space according to the partition induced by the structure of the composed model as discussed in Section 4.2. In fact, we want to compute $\widetilde{\mathcal{S}} = \mathcal{S}/R$, the quotient of the unlumped state space $\mathcal{S}$ with respect to the lumpable partition $R$.

Before we give the details of our approach, we define the extension of the equivalence relations $R$ and $R_c$ ($c \in RC$) to the *potential* state space $\overline{\mathcal{S}} = \times_{c=1}^{m} \mathcal{S}_c \subset \mathbb{N}_0^m$. Observe that $\mathcal{S} \subseteq \overline{\mathcal{S}}$.

**Definition 4.8** *The relations $\overline{R}_c \subseteq \overline{\mathcal{S}} \times \overline{\mathcal{S}}$ ($c \in RC$) and $\overline{R} \subseteq \overline{\mathcal{S}} \times \overline{\mathcal{S}}$ are defined as follows:*

*a.* $(s, s') \in \overline{R}_c \Leftrightarrow \exists\, sp_c \colon s' = \alpha(sp_c, s)$.

*b.* $(s, s') \in \overline{R} \Leftrightarrow \exists\, sp_{RC} \colon s' = \alpha(sp_{RC}, s)$.

As we can see, the only essential difference between the pair of relations $\overline{R}$ and $\overline{R}_c$ and the pair $R$ and $R_c$ is that the former pair are defined on $\overline{\mathcal{S}}$ and the latter pair on $\mathcal{S}$. As for $R$ and $R_c$, we can prove that $\overline{R}$ and $\overline{R}_c$ are equivalence relations, and hence, they induce partitions $\overline{\mathcal{P}}$ and $\overline{\mathcal{P}}_c$ on $\overline{\mathcal{S}}$.

In order to compute $\widetilde{\mathcal{S}}$ from $\mathcal{S}$, we pick a specific element of each equivalence class as the representative of the class. The choice of that representative is arbitrary because all states of the class behave identically. We would like to choose a state whose vector representation has a mathematical property that simplifies the computation of $\widetilde{\mathcal{S}}$. In particular, among a set $C$ of equivalent states, we choose the one whose vector representation is lexicographically the smallest (refer to the definition of $\leq_{lex}$ in Chapter 2.

We can now formally define the notion of the representative state of a class of a given partition $\mathcal{T}$ on a set of states $Y \subseteq \mathbb{N}_0^m$. We define the function $min_{\mathcal{T}}(s) : Y \to Y$, which

gives the representative state of the class of $\mathcal{T}$ to which $s$ belongs, as follows:

$$min_{\mathcal{T}}(s) = s' \Leftrightarrow s \approx_{\mathcal{T}} s' \text{ and } \forall\, s'' \approx_{\mathcal{T}} s \colon s' \leq_{lex} s''$$

In plain English, $s'$ is the smallest of the states equivalent to $s$. We have $min_{\mathcal{T}}(s) = min_{\mathcal{T}}(s')$ for all $s \approx_{\mathcal{T}} s'$. For simplicity, we use $min_c$ and $min$ to denote $min_{\overline{\mathcal{P}}_c}$ and $min_{\overline{\mathcal{P}}}$, respectively ($c \in RC$). Observe that

$$min(s) = s' \Leftrightarrow \forall\, c \in RC \colon min_c(s) = s'. \tag{4.5}$$

Given any state $s$, the corresponding class representative $min(s)$ can be computed by appropriate sorting operations on $s$. Notice that for every equivalence class $C \in \overline{\mathcal{P}}$, there exists only one $s \in C$ (that is, the representative state) such that $min(s) = s$. To compute the MDD representation of $\widetilde{\mathcal{S}}$, we eliminate from the MDD representation of $\mathcal{S}$ all paths (states) that do not satisfy $min(s) = s$. That computation is translated, in terms of MDDs, to the computation of $\mathcal{S} \cap \mathcal{R}$, where $\mathcal{R} = \{s \in \overline{\mathcal{S}} \mid min(s) = s\}$. Hence, the problem of computing $\widetilde{\mathcal{S}}$ is reduced to that of building $\mathcal{R}$ based on the definition of $min$.

However, the definition of $min(s)$ imposes a strict relationship among the various components of the vector representation of $s$, which implies a tight coupling among levels of $\mathcal{R}$ in terms of MDDs. Therefore, the number of nodes of $\mathcal{R}$ grows very quickly in terms of the number of levels involved in the definition of $min(s)$,[6] and this makes the direct computation of $\mathcal{S} \cap \mathcal{R}$ problematic.

To avoid that large memory consumption, we can express the large MDD of $\mathcal{R}$ in terms of a small number of considerably smaller MDDs, and instead of computing $\mathcal{S} \cap \mathcal{R}$ directly, we compute the intersection of $\mathcal{S}$ with a large set expression that is equal to $\mathcal{R}$. As the first step, using Eq. 4.5, we can see that $\mathcal{R} = \cap_{c \in RC} \mathcal{R}_c$, where $\mathcal{R}_c$ is the set of all states $s \in \overline{\mathcal{S}}$ that

[6] In the case of nested replicate operators, the number can be exponential in terms of the cardinality of the inner replicate operators.

satisfy $min_c(s) = s$. That implies $\mathcal{S} \cap \mathcal{R} = (\cdots(\mathcal{S} \cap \mathcal{R}_{c_1}) \cap \cdots \cap \mathcal{R}_{c_r})$, where $RC = \{c_1, \ldots, c_r\}$. Since in general, each of the $\mathcal{R}_c$'s involves tight coupling among far fewer levels than $\mathcal{R}$ does, each $\mathcal{R}_c$ is significantly smaller than $\mathcal{R}$. Hence, computing $(\cdots(\mathcal{S} \cap \mathcal{R}_{c_1}) \cap \cdots \cap \mathcal{R}_{c_r})$ is much faster than computing $\mathcal{S} \cap \mathcal{R}$ directly, because the efficiency we gain by using smaller-sized $\mathcal{R}_c$'s outweighs the extra time we have to spend to compute $r$ intersection operations rather than one.

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

## 4.5 State Transition Rate Matrix Generation and Numerical Analysis

In this section, we describe how to perform an iterative numerical analysis based on an MD representation of $\widetilde{\mathbf{R}}$. Its basic step is a matrix-vector multiplication, which requires consideration of several issues if it is performed with an MD. We start with the generation of an MD from the local transition rate matrices generated during state-space exploration. In Section 4.3 only Boolean matrices $\mathbf{B}_c$ of state transitions are mentioned; however, it is easy to obtain corresponding rates (possibly scaled by probabilities of paths of subsequent immediate transitions) that yield matrices $\mathbf{R}_c$. With the MD representation of the unlumped CTMC at hand, we need to focus on $\widetilde{\mathcal{S}}$ as the set of rows. Matrix entries in those rows will

refer to columns $s'$ whose correspondence to $min(s')$ must be established. Finally, there are cases in which the MD will generate multiple elements that must be added for a single matrix entry in $\widetilde{\mathbf{R}}$. These issues are resolved in the remainder of this section.

### 4.5.1 State Transition Rate Matrix Generation using MDs

Formally, we first derive a generalized Kronecker representation of the rate matrix $\mathbf{R}$ that gives us an unlumped MD in a straightforward manner, and then use the MDD representation of $\widetilde{\mathcal{S}}$ to obtain a projection on the lumped state space. Conceptually, MD generation with the help of a Kronecker representation and MDD projection follows the approach of [28]. However, it differs in important aspects. In particular, the Kronecker representation we derive contains functional transitions [87] that are subsequently resolved to constant values in the MD. The MD that finally results requires additional, specific algorithms to describe the rate matrix of the lumped CTMC. Note that an implementation directly generates an MD based on the local transition rate matrices obtained during state-space exploration.

**A Kronecker representation for R.** A Kronecker structure makes use of the matrix operator Kronecker product $\otimes$ to combine small component matrices into a large matrix. The building blocks of the Kronecker representation are matrices $\mathbf{R}_{a,c}$ that represent the effect of timed action $a$ on atomic component $c$.[7] Let $\gamma_c : \mathcal{S} \rightarrow \times_{v \in V_c} D_v$ be a mapping that provides the state in terms of its SVs for an atomic model $\mathsf{M}_c$ with component index $c \in AM$. Also, let $\eta_c = |range(\gamma_c)|$. In fact, $\mathbf{R}_{a,c} \in \mathbb{R}^{\eta_c \times \eta_c}$, and $\mathbf{R}_{a,c}(s_{V_c}, s'_{V_c})$ is the weight of $a$ at $s_{V_c}$ multiplied by the probability of reaching $s'_{V_c}$ via some sequences of immediate actions in $\mathsf{M}_c$, where $s_{V_c}$ and $s'_{V_c}$ are states of atomic component $c$.

Note that the difficulty in the derivation of $\mathbf{R}_{a,c}$ is not in calculating entries, which is done using the definition of $\mathsf{M}_c$. The difficulty is in finding the set of reachable states of $\mathsf{M}_c$, since sharing state variables with other models causes other models to generate new states

[7] Immediate actions are used only during the on-the-fly eliminations of vanishing states.

as well[8]. This difficulty is overcome by using $\mathcal{S}$ as described below. Specifically, for each timed action $a$ and atomic component $c$, we define $m$ matrices $\mathbf{R}_{a,c}^{c'}, c' \in \{1, \ldots, m\}$, where $\mathbf{R}_{a,c}^{c'}$ denotes the projection of $\mathbf{R}_{a,c}$ on $\mathcal{S}_{c'} \times \mathcal{S}_{c'}$. In fact, $\mathbf{R}_{a,c}^{c'}$ denotes the "effect" of $\mathbf{R}_{a,c}$ on level $c'$ of the MD representation of $\mathbf{R}$. More formally, $\mathbf{R}_{a,c}^{c'} \in \mathbb{R}^{\mathcal{S}_{c'} \times \mathcal{S}_{c'}}$ and

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

where $f_a : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$ is a functional transition that evaluates to $\mathbf{R}_{a,c}(\gamma_c(s), \gamma_c(s'))$ for given states $s, s'$; see [87] for the definition and treatment of Kronecker representations that are generalized with respect to functions as matrix entries. Notice that $\mathbf{R}_{a,c}^{c'}$ is simply an identity matrix if $c' \notin I_c$, where, as defined before, $I_c$ is the set of indices of MDD levels in which the SVs in $V_c$ are encoded.

With those matrices, we obtain a Kronecker representation to describe a state transition rate matrix $\widehat{\mathbf{R}}$. Its basic operation, the Kronecker product $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$, is defined for matrices $\mathbf{A}^{(n \times m)}, \mathbf{B}^{(k \times l)}$, and $\mathbf{C}^{(nk \times ml)}$ as $\mathbf{C}(a_1 k + b_1, a_2 l + b_2) = \mathbf{A}(a_1, a_2) \cdot \mathbf{B}(b_1, b_2)$. We define $\widehat{\mathbf{R}}$ of size $\Pi_{i=1}^{m} |\mathcal{S}_i| \times \Pi_{i=1}^{m} |\mathcal{S}_i|$ as

$$\widehat{\mathbf{R}} = \sum_{M_c \in AM} \sum_{a \in A_c} \otimes_{c'=1}^{m} \mathbf{R}_{a,c}^{c'}, \tag{4.6}$$

where $AM$ is the set of indices of atomic components. We briefly explain why $\mathbf{R}$ is a submatrix of $\widehat{\mathbf{R}}$. Consider an entry $\mathbf{R}((s_1, \ldots, s_m), (s'_1, \ldots, s'_m)) = \lambda$. Since several actions may contribute to $\lambda$, we have $\lambda = \sum_{a \in E(s)} \lambda_a(s, s')$, where $\lambda_a(s, s')$ is $w_a(s)$ possibly multiplied by the probability of a subsequent sequence of immediate actions yielding $s'$. For

[8]Formally, one may consider those new states as a set of initial states that may grow as a result of the firing of other models' global actions.

any term $\lambda_a(s, s') > 0$, we defined $\mathbf{R}_{a,c}^{c'}(s_{c'}, s'_{c'}) = \lambda_{c'} > 0$. Since only $\lambda_c \neq 1$ we have $\prod_{c'=1}^{m} \lambda_{c'} = \lambda_c = f_a = \mathbf{R}_{a,c}(\gamma_c(s), \gamma_c(s'))$. By the definition of Kronecker product, $\otimes_{c'=1}^{m} \mathbf{R}_{a,c}^{c'}$ contributes $\lambda_c = \prod_{c'=1}^{m} \mathbf{R}_{a,c}^{c'}(s_{c'}, s'_{c'})$ to $\widehat{\mathbf{R}}((s_1, \ldots, s_m), (s'_1, \ldots, s'_m))$. $\mathbf{R}$ is a submatrix of $\widehat{\mathbf{R}}$ since $\mathcal{S} \subseteq \times_{c'=1}^{m} \mathcal{S}_{c'}$.

**MD construction for the lumped state-transition rate matrix.** Transformation of a Kronecker representation into an MD is immediate. For each term $\otimes_{c'=1}^{m} \mathbf{R}_{a,c}^{c'}$, we define an MD with one node per level; the node at level $c'$ contains matrix $\mathbf{R}_{a,c}^{c'}$, and its nonzero entries point to the node at level $c' + 1$. In the case of $c' = m$, the nonzero entries formally point to terminal node 1. Since addition is defined for MD, we can sum the resulting MDs of all terms in the two sums in Eq. 4.6. Note that the functional transitions that appear in $\mathbf{R}_{a,c}^{c'}$ can be resolved to constant values in the MD, because sets $\mathcal{V}_{c'}$ are ordered such that the sets that contain shared SVs of an atomic model $\mathsf{M}_c$ all have lower indices than $c$, and thus those sets appear at a higher level of the MD. Hence, if a path through the MD reaches level $c$, the values of all shared SVs are known. Resolving functional transitions into constant values may require the splitting of matrices that were otherwise shared in the MD.

The advantage of an MD over a Kronecker representation is that we can restrict the MD to the $\mathcal{S} \times \mathcal{S}$ submatrix contained in a Kronecker representation. In order to do that, we simply project the set of rows and columns of the MD on $\mathcal{S}$. In the following, we describe how the projection operation transforms the MD representation. Assume that we refine the definition of matrices as $\mathbf{R}_{a,c}^{c'}[(s_1, \ldots, s_{c'-1}), (s'_1, \ldots, s'_{c'-1})] \in \mathbb{R}^{\mathcal{S}_{c'} \times \mathcal{S}_{c'}}$ so that it will depend on the subset of states $(s_1, \ldots, s_{c'-1}), (s'_1, \ldots, s'_{c'-1})$, namely:

$\mathbf{R}_{a,c}^{c'}[(s_1,\ldots,s_{c'-1}),(s'_1,\ldots,s'_{c'-1})](s_{c'},s'_{c'}) =$

$$\begin{cases} 1 & \text{if } c' \neq c \text{ and} \\ & \exists s=(s_1,\ldots,s_m), s'=(s'_1,\ldots,s'_m) \in \mathcal{S} \\ & \text{such that } \mathbf{R}_{a,c}(\gamma_c(s),\gamma_c(s')) \neq 0 \\ \mathbf{R}_{a,c}(\gamma'_c(s_1,\ldots,s_c),\gamma'_c(s'_1,\ldots,s'_c)) & \text{if } c'=c \\ 0 & \text{otherwise} \end{cases}$$

where $\gamma'_c$ is the same as $\gamma_c$ but defined on components $1,\ldots,c$, which is possible since all (shared) SVs of component $c$ appear at components $c' \leq c$ and $a$ is independent of SVs at components $c+1,\ldots,m$. Hence, for any $(s_1,\ldots,s_m) \in \mathcal{S}$ the equality $\gamma'_c(s_1,\ldots,s_c) = \gamma_c(s_1,\ldots,s_m)$ holds. To build an MD data structure out of these matrices, we let an entry in $\mathbf{R}_{a,c}^{c'}[(s_1,\ldots,s_{c'-1}),\ (s'_1,\ldots,s'_{c'-1})](s_{c'},s'_{c'})$ point to matrix $\mathbf{R}_{a,c}^{c'+1}[(s_1,\ldots,s_{c'}),(s'_1,\ldots,s'_{c'})]$ if $c' < m$. Pointers from nonzero entries at level $m$ point to terminal node 1. To keep the definition of those matrices readable, we oversized their dimension as $\mathcal{S}_{c'} \times \mathcal{S}_{c'}$; hence, some rows and columns in the matrices of the MD contain only zero entries and can safely be removed.

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

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

applied to minimize space requirements of the overall structure.

We need to compute the elements of $\widetilde{\mathbf{R}}$, which has its transitions from $\widetilde{\mathcal{S}}$ to $\widetilde{\mathcal{S}}$. The first (incorrect) solution that comes to mind is to follow the approach of [27, 28, 73] and project the rows and columns of the MD we have built so far on $\widetilde{\mathcal{S}}$. The reason why that approach does not work in our case is that not all transitions of $\mathbf{R}$ starting from a representative state end in another representative state. In other words, there are some states that are reachable but not representative of their class. Hence, by projecting both rows and columns on $\widetilde{\mathcal{S}}$, we would remove all transitions from any state in $\widetilde{\mathcal{S}}$ to any state in $\mathcal{S}$. The solution to the problem is to project the rows and columns of the MD to $\widetilde{\mathcal{S}}$ and $\mathcal{S}$, respectively. The resulting MD provides rates of state transitions from $s \in \widetilde{\mathcal{S}}$ to $s' \in \mathcal{S}$. As we said, $s'$ may belong to $\mathcal{S} \backslash \widetilde{\mathcal{S}}$, i.e., $s' \neq min(s')$. In that case, a straightforward option is to sort the vector representation of $s'$ to obtain the representative state $min(s') \in \widetilde{\mathcal{S}}$ of its equivalence class. Performing the sort operation on each access to a transition like that is not efficient because each state $s' \notin \widetilde{\mathcal{S}}$ may need to be sorted many times. We will devise a more efficient approach to this problem below.

A recursive DFS procedure enumerates all matrix entries encoded in the MD as triples $(s, s', \lambda)$, where $\lambda$ results from the product of values found on a path from the root node to a leaf node in the MD. The procedure was inspired by the Act-RwCl algorithm of [20], formulated for matrix diagrams in [74]. DFS gains its efficiency by following paths through all entries of a matrix $\mathbf{R}_{a,c}^{i}$ before returning to level $i-1$. In that way it amortizes the cost of following a path in the MD and increases locality of access to data structures. Note that the construction ensures that all but one of the entries along a path are 1, so in fact no multiplications are required to compute the product of values along a path in the MD. The state $s = (s_1, \ldots, s_m) \in \widetilde{\mathcal{S}}$ must be mapped to the corresponding index value in $\{0, \ldots, |\widetilde{\mathcal{S}} - 1|\}$ to support a matrix-vector multiplication. Other MD approaches perform that mapping by an offset function $\rho$ encoded in an MDD [27, 28, 73]. In our case, we can look up $\rho(s)$ from the MDD of $\widetilde{\mathcal{S}}$ with the help of the offset computation known for MDDs.

Repetitive sorting is avoided if we construct a new "sorting" MDD whose offset function $\rho'$ is modified to fulfill $\rho'(s') = \rho(min(s'))$. That means that states of the same equivalence class will evaluate to the same offset value. To generate the sorting MDD, we can start from an unreduced MDD in the form of a tree for set $\mathcal{S}$. A valid initial way to encode the mapping is to assign $\rho'_m(s_1, \ldots, s_m) = \rho(min(s_1, \ldots, s_m))$ and 0 to all internal values $\rho'_c(s_1, \ldots, s_c), c < m$. In order to allow for sharing, we perform a bottom-up procedure. Let $\mu(s_1, \ldots, s_c) = \min_{s_c}\{\rho'_c(s_1, \ldots, s_c)\}$; then new offset values are $\rho'_{c-1}(s_1, \ldots, s_{c-1}) = \mu(s_1, \ldots, s_c)$ and $\rho'_c(s_1, \ldots, s_c) = \rho'_c(s_1, \ldots, s_c) - \mu(s_1, \ldots, s_c)$. The changes leave $\rho'(s) = \sum_{c=1}^{m} \rho'_c(s_1, \ldots, s_c)$ invariant, but reduce the ranges of numerical values at lower levels of the sorting MDD to allow for sharing. The space used for the sorting MDD depends on the degree of sharing; however, the offset computation for $s' \notin \widetilde{\mathcal{S}}$ can take place at the same cost as for $s \in \widetilde{\mathcal{S}}$. That cost is $O(m)$ for a single state considered individually, because we have to traverse $m$ nodes in the sorting MDD, but since we traverse the MD, the MDD for $\widetilde{\mathcal{S}}$, and the sorting MDD simultaneously and in a DFS order, we profit from locality and let the cost be amortized.

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

However, for many applications, accumulation of entries may be desirable for numerical accuracy but is not a necessity for the algebraic operations performed on the matrix. For instance, a matrix-vector multiplication does not require accumulation, since multiplication

distributes over addition. In the current implementation, we do not perform accumulation of multiple entries to save the overhead in performance.

### 4.5.2 Numerical Analysis

So far, we described how to enumerate all matrix entries of $\widetilde{\mathbf{R}}$ as triples $(\rho(s), \rho(min(s')), \lambda)$ in the order imposed by the DFS procedure on the MD, which implies that we do not observe matrix entries being ordered by rows or by columns. However, it is possible to obtain an order by submatrices, since the top-level node of the MD imposes a block structure on the resulting matrix, which is naturally followed by the DFS enumeration procedure. Following [43], the enumeration of matrix entries suffices to implement matrix-vector multiplication $\mathbf{x} \cdot \widetilde{\mathbf{R}}$, which in turn is essentially what is needed to perform iterative solution methods like the Power method or Jacobi's method for steady-state analysis and uniformization for transient analysis. Some iterative methods require access by submatrices (e.g., Block-SOR, IAD, and Takahashi's method), which also can be efficiently supported. Only efficient access of columns of $\widetilde{\mathbf{R}}$, required for Gauss-Seidel or SOR, remains as an open research problem. The inefficiency results from the poor locality of data accesses, and therefore poor use of the hardware cache, in the algorithm that accesses the MD by columns. In particular, we implemented the column access algorithm given in [28] along with the proposed column caching scheme, and we observed about one order of magnitude slower running times than we did when accessing the elements of the MD in DFS order. As a side remark, we note that we can also use the current enumeration of entries to create an additional, canonical MD [73] and use existing MD multiplication schemes for that canonical MD.

Finally, in order to complete our representation of an MRP, we need some representation of the reward vector $\widetilde{\mathbf{r}}$ in addition to the representation of $\widetilde{\mathcal{S}}$ and $\widetilde{\mathbf{R}}$. Since we have an MDD representation of $\widetilde{\mathcal{S}}$, the simple and space-efficient solution is to use MTMDDs. To build such an MTMDD, we traverse the MDD representation of $\widetilde{\mathcal{S}}$ in DFS order and for each state $s \in \widetilde{\mathcal{S}}$ we compute the value of the reward in that state, i.e., $\widetilde{\mathbf{r}}(s)$, possibly by evaluating

a high-level reward specification given along with the model specification. The DFS order allows us to build the MTMDD in a bottom-up manner while eliminating redundant nodes, which leads to the space efficiency of the representation. In the actual implementation, that process is repeated for each of the reward specifications defined by the modeler.

**Related work** We have described a method to compute an MDD and MD for a lumped CTMC of composed models that share state variables. Existing results of Ciardo and Miner [25, 27–29] for MDD and MD generation of composed models that share actions are related and have been used here; the concept of a local transitive closure has been discussed in different contexts [22, 25, 29]. The novelty of our approach is in the encoding of a different composition operation into symbolic data structures and the treatment of general next-state and weight functions for global actions. The dynamic generation of state spaces for atomic components has been developed independently of the recent result in [26], which is nevertheless conceptually closely related. Note that the encoding of matrix entries we selected in the MD implies that the weight of a state transition results from a product of values of which only one is some number other than 1, so that no real multiplications are required. That is an advantage in terms of efficiency. Furthermore, we exploit the fact that replicated atomic models can share the same matrices in the state space. The definition of a sorting MDD has not been considered before, and it illustrates how an arbitrary function could be encoded as an offset function of an MDD. In summary, the consideration of a different composition operation and its combination with lumping techniques required us to develop several innovations that go beyond existing work.

## 4.6 Performance Results

As stated in the introduction, the goal of our work was to create CTMC generation algorithms that simultaneously exploit the symmetries in models to reduce the number of states

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

### 4.6.1 Implementation in Möbius

In order to test the efficiency of the developed algorithms, we implemented them within Möbius [38, 48]. We have completed the implementation of the MDD-based state space (SS) generation, the lumping algorithm, and the MD-based generation of the lumped CTMC for composed models that consist of an arbitrary number of replicate and join operators. We also implemented the `AllEdges` iterator to support numerical analysis using the Möbius state-level AFI (Abstract Functional Interface) [43]. The `AllEdges` iterator gives access to the elements of $\widetilde{\mathbf{R}}$ with no specific order. We also implemented the `Column` iterator, which gives access to any given column of $\widetilde{\mathbf{R}}$. As we mentioned before, the time efficiency of the `Column` iterator was about an order of magnitude worse than that of `AllEdges`; therefore, we did not include the `Column` iterator in the Möbius tool. Since the `AllEdges` iterator is supported, all the numerical solvers in Möbius that support that iterator can be used. Finally, we implemented the algorithm that generates MTMDDs for all the rewards defined on the model.

The SSG implementation interacts with the component models using the Möbius model-level AFI [38], thus supporting any atomic model type that Möbius supports, including stochastic activity networks, PEPA (Performance Evaluation Process Algebra), and Buckets and Balls, and accepts composed models generated by the Möbius Replicate-Join composed

model editor. All the code involved in the experiments was compiled using the gcc 3.3 compiler with the -O3 optimization option. All experiments were conducted using an Athlon XP2400 machine with 1.5 GB of main memory.

In developing efficient algorithms, many enhancements are small from a conceptual point of view, but can have a large practical impact. One obvious and effective technique we used was to automatically remove levels of the MD/MDD data structures whose corresponding $\mathcal{V}_c$ sets are empty. In the second example model we describe below, that technique reduced the number of levels by about 50%, which made traversing the MDDs and MDs, and therefore the CTMC solution, faster. The other technique that we used was to convert dynamic data structures into static ones after the data structures have been constructed and have ceased to be modified, e.g., we converted all the linked lists that were created for each of the MD nodes during MD construction to arrays. According to our experiments, that decreased the solution time by about 10 to 15 percent.

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

### 4.6.2 Example Models

**Courier protocol.** We first consider a SAN model of a parallel communication software system [93]. The model is parameterized by the transport window size $TWS$, which limits the number of packets that are simultaneously communicated between the sender and receiver. In order to retain a significant number of actions, we considered a model in which all actions are timed. To form a composed model, we have broken the original model (see Appendix A.2) into 4 atomic models, one for each of the following parts of the model: 1) the receiver's session layer, 2) the receiver's transport layer, 3) the sender's session layer, and 4) the sender's transport layer. Figure 4.6 shows a graphical representation (similar to those from the Möbius tool) of how those atomic models are composed using the join operator and how state variables are shared among them. Thick-lined and thin-lined boxes represent join

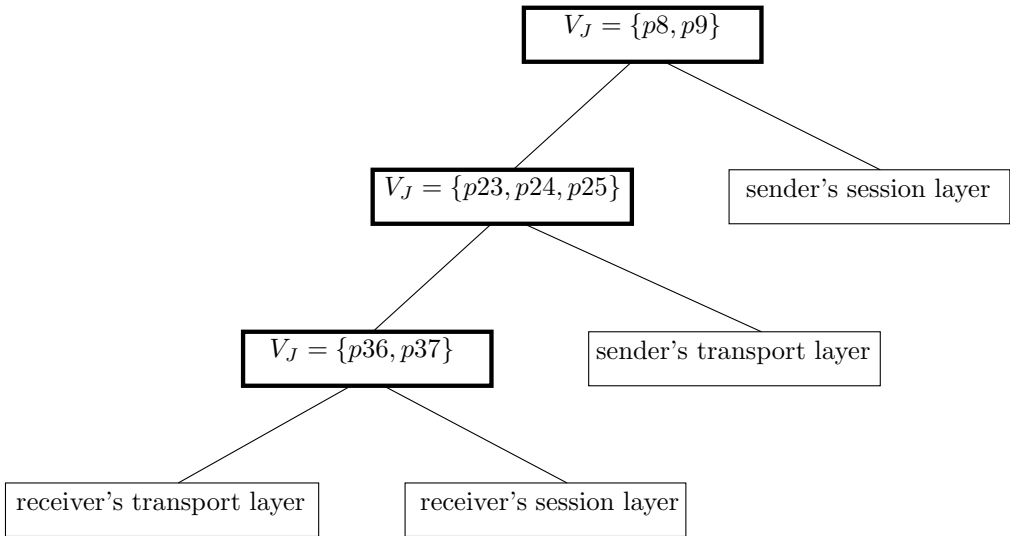



Figure 4.6: Composed model structure of the Courier model

| $TWS$ | state space | | | | SSG time (sec) |
|---|---|---|---|---|---|
| | # of states | final # nodes | mem (KB) final | mem (KB) peak | |
| 3 | 2,381,184 | 23 | 13 | 224 | 1.1 |
| 4 | 9,710,208 | 29 | 31 | 725 | 4.8 |
| 5 | 32,404,608 | 35 | 68 | 1970 | 21.0 |
| 6 | 93,301,632 | 41 | 135 | 4760 | 85.4 |
| 7 | 239,651,712 | 47 | 254 | 10600 | 325.3 |
| 8 | 561,818,880 | 53 | 453 | 21900 | 1020.0 |

Table 4.1: State-space sizes and generation times for the Courier protocol model

operators and atomic models, respectively. Since the replicate operator is not used in the model, lumpability induced by structure is not present, and therefore the lumping algorithm is not applied to the state space produced by the SSG algorithm.

Table 4.1 shows the size of the state space, the state-space generation time, and the solution time (per iteration) for different values of $TWS$. Since each atomic model has a nonempty set of local actions, the atomic models all have some local behavior that makes use of the saturation technique. As we can see, the final size of the MDD representation of the state space does not exceed 500KB.

In Table 4.2, we have compared the performance of transient solutions using two repre-

| $TWS$ | transient solution (sec/iteration) | | slowdown |
|---|---|---|---|
| | MxD | APNN | |
| 3 | 1.26 | 0.89 | 1.42 |
| 4 | 5.54 | 3.76 | 1.47 |
| 5 | 19.15 | 12.76 | 1.50 |

Table 4.2: Solution times (per iteration) for the Courier protocol model

sentations: (1) a Kronecker representation generated by the APNN toolbox [6], and (2) a matrix diagram representation generated by the algorithm given in this chapter. CTMCs represented by both representations have been solved using a state-level AFI-compliant transient solver. The numbers shown in the second and third columns of the table are the times to complete one iteration for each of the representations. For $TWS > 5$ the total size of the probability vectors required for the CTMC solution was too large to fit into the memory of the machine. As we can see, our implementation of iteration on matrix diagrams is at most 1.5 times slower than APNN's iteration on the Kronecker representation, in which an efficient variant of the Act-RwCl algorithm of [20] is used. Notice that the APNN toolbox is not able to exploit structural symmetries in the model in order to lump the underlying CTMC. Therefore, a direct comparison between the APNN toolbox and our algorithm is possible because the Courier model does not have any symmetries to be exploited by our algorithm. However, for the next example model for which lumping is possible, APNN has to operate on the unlumped CTMC, which is dramatically larger than the lumped CTMC on which our algorithm operates.

**Fault-tolerant parallel computer system.** As a second test, we consider a model of a highly redundant fault-tolerant parallel computer system [67]. This model uses both replicate and join operators, and hence provides a more complete test of our algorithms and implementation. The model was first published in [81] where the model has been thoroughly described and specified using the SAN (Stochastic Activity Network) formalism. Therefore, we do not present a full specification of the model in this dissertation and suffice to give a

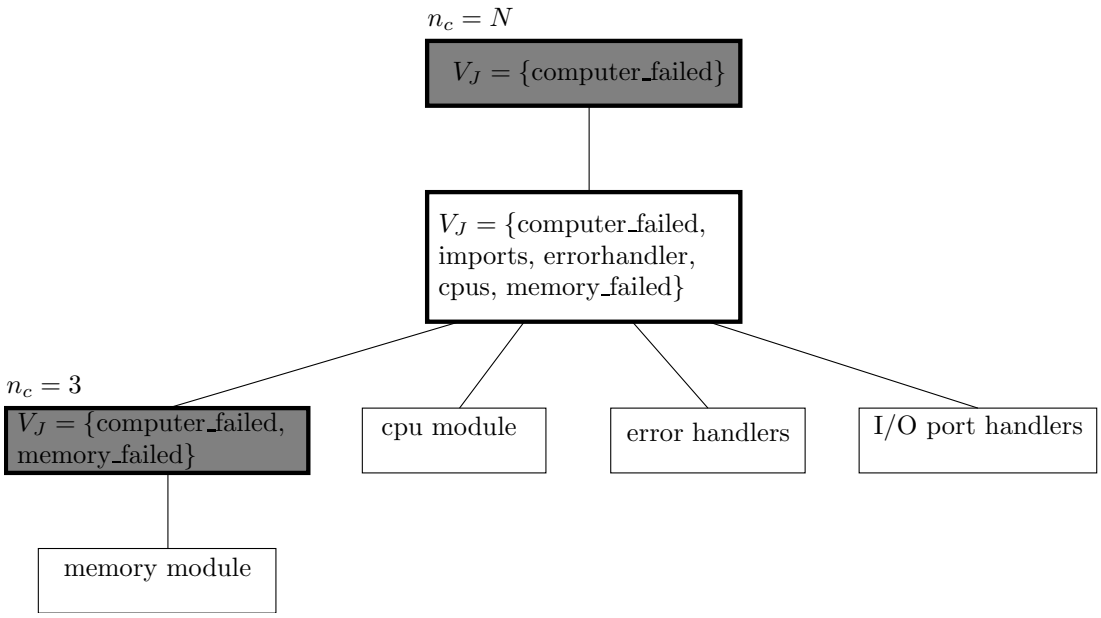



Figure 4.7: Composed model structure of the parallel computer system

high-level description of the model here[9].

We built a composed model for the entire system by first defining atomic models using the SAN formalism [82] to represent the failure of various components in the system. We then used the replicate and join operators to construct the complete composed model shown in Figure 4.7. The thick-lined shaded boxes represent the replicate operator. The leaf nodes of the tree, which are labeled "memory module," "cpu module," "I/O port handlers," and "error handlers," correspond to the atomic models of the reliability of the computer's memory module, its 3 CPU units, its 2 I/O ports, and its error-handling mechanism, respectively. The memory module is replicated 3 times, which equals the number of memory modules in one computer. The replicate component is then joined with the I/O ports model, the CPUs failure model, and the error-handler model to create a join component that models a computer. Finally, the model of one computer is replicated $N$ times to generate the complete composed model of the multiprocessor system.

Table 4.3 shows the number of states in the unlumped and the lumped state spaces, the

[9]The complete Möbius model is also included with the distribution of the tool and can be obtained from `http://www.mobius.uiuc.edu`.

| $N$ | # of unlumped states | # of lumped states | reduction in state-space size | total generation time (sec.) |
|---|---|---|---|---|
| 1 | 414 | 116 | $3.59\times10^{0}$ | 0.027 |
| 2 | 256,932 | 10,114 | $2.54\times10^{1}$ | 1.30 |
| 3 | 124,075,800 | 463,268 | $2.68\times10^{2}$ | 25.1 |
| 4 | 55,039,441,680 | 14,773,967 | $3.72\times10^{3}$ | 200 |
| 5 | 23,549,313,739,104 | 366,912,104 | $6.40\times10^{4}$ | 1310 |
| 6 | 9,908,244,947,184,192 | 7,530,481,244 | $1.31\times10^{6}$ | 5250 |

Table 4.3: Unlumped and lumped state-space sizes and generation times

reduction in the size of the state space due to lumping, and also the total time it takes to generate the MDD representation of the unlumped and the lumped state spaces and the MD representation of the lumped CTMC from the composed model representation. As we can see, lumping reduces the size of the CTMC that must be solved by up to 6 orders of magnitude. Due to the efficient technique we use to compute lumped SS from unlumped SS, the lumping operation takes less than 0.3% of the total time for the example model. It means that the time to generate the lumped SS is essentially equal to the time to generate the unlumped SS. Note that this example is a "worst case" input, in terms of speed, for our state-space exploration algorithm. By "worst case" we mean that none of the atomic components of the model have any local action. The lack of local behavior is caused by the tight coupling that exists among the atomic models of a computer module; in terms of modeling, that coupling is realized by sharing of all the SVs in the join operator of the model. Not having local actions means that the techniques described in Section 4.3 cannot be used to generate any new state based on local behavior of the atomic models. Nevertheless, the generation times reported are reasonable, and show that the times required to generate the lumped CTMCs are small, even for state spaces of extremely large size.

Table 4.4 shows the amount of memory the MDD and MD representations (except for sorting MDD) take. The number of MDD nodes used to represent the state space and the amount of memory taken by the nodes in kilobytes (KB) are given for each form (i.e., lumped and unlumped) of the state space. The peak memory use of the MDD nodes is also given.

For the MD representation, the number of nodes and the memory use of the data structure are shown under the column labeled "MD (final/peak)," since the peak values are equal to the final values for the MD representation.

| $N$ | unlumped SS (MDD) | | lumped CTMC | | | | |
|---|---|---|---|---|---|---|---|
| | # of nodes | mem (KB) | MDD (lumped SS) | | | MD (final/peak) | |
| | | | final # of nodes | mem (KB) final | mem (KB) peak | # of nodes | mem (KB) |
| 1 | 14 | 1.5 | 20 | 2 | 15 | 39 | 5 |
| 2 | 43 | 4.5 | 342 | 45 | 118 | 997 | 160 |
| 3 | 99 | 10.3 | 1496 | 194 | 441 | 4100 | 652 |
| 4 | 167 | 17.3 | 3397 | 424 | 1050 | 9640 | 1550 |
| 5 | 247 | 25.5 | 5726 | 705 | 2050 | 17500 | 2820 |
| 6 | 339 | 34.8 | 8483 | 1040 | 3560 | 27500 | 4440 |

Table 4.4: Space requirements of MDD and MD representations of unlumped and lumped CTMCs

Note that the amount of memory that a lumped SS takes is larger than the amount of memory that the corresponding unlumped SS takes. That happens because from each of the equivalence classes of the state space, we eliminate all except one representative state. That causes the set of states after lumping to be less "structured" than before lumping, and hence the size of the MDD grows after the lumping operation. However, even after lumping, the size of the final MDD is still very small ($< 1.1$ MB) for all considered values of $N$. Since our goal is the numerical solution of the resulting CTMC, in addition to considering the time and space constraints on the CTMC generation, we also have to consider the limitation we have on the size of the solution vector, which grows linearly with the number of states. Therefore, reducing the number of states of the CTMC is crucial, and is a significant advantage of our technique over symbolic techniques that do not support lumping. In that respect, it is important to observe that the number of states in the lumped state space (Table 4.3) does not grow as fast as the number of states in the unlumped one for increasing values of $N$.

Finally, we measured the performance of our implementation in enumerating the elements of an MD-based state-transition rate matrix and compared it, this time, to the performance

| $N$ | # of states | # of transitions | time/iteration (sec) | | slowdown factor |
|---|---|---|---|---|---|
| | | | matrix diagram | sparse matrix | |
| 2 | $1.01\times10^{4}$ | $5.51\times10^{4}$ | $1.39\times10^{-2}$ | $2.83\times10^{-3}$ | 4.91 |
| 3 | $4.63\times10^{5}$ | $3.51\times10^{6}$ | $9.59\times10^{-1}$ | $1.75\times10^{-1}$ | 5.48 |
| 4 | $1.48\times10^{7}$ | $1.43\times10^{8}$ | $4.42\times10^{1}$ | – | – |

Table 4.5: Lumped CTMC characteristics and solution times (per iteration)

of enumeration of a sparse-matrix-based (traditional) state-transition rate matrix. Both state-transition rate matrices correspond to the lumped CTMC. We make the comparisons via the numerical solution of the lumped CTMC. As we are measuring the reliability of the parallel computer system, we use the uniformization method, as implemented in Möbius, for transient solution of the model. Table 4.5 shows the sizes of the lumped CTMCs and the solution times using two different representations (MD representation and sparse-matrix representation) of the lumped CTMC, and the ratio of the two solution times. Remember that the current implementation does not use the accumulation technique described in Section 4.5, and therefore, the number of transitions represented in the MD representation is larger than the number of transitions in the sparse-matrix representation. For the example model, the number of transitions processed in the MD representation is 39% to 42% greater than the number of actual transitions, i.e., the number of transitions in the sparse-matrix representation. Therefore, the slowdown we are experiencing is due to the increased number of transitions to be enumerated and also the greater processing time needed to compute each transition; that is the cost we pay for compact representation. As can be seen in the table, the slowdown we are experiencing is less than 6 for large matrices. It is important to notice that compact representation of a lumped CTMC whose number of states is 3 orders of magnitude smaller than the number of states of its corresponding unlumped CTMC (for $N = 4$) is gained only at the cost of solution times that are about 6 times slower.

The available 1.5 GB of main memory in our machine limits the numerical solution. In particular, the sparse matrix solver causes thrashing of virtual memory for $N = 4$ due to the

| $N$ | # of states | sorting MDD | | |
|---|---|---|---|---|
| | | # of nodes | mem (KB) | gen. time (sec) |
| 2 | $1.01\times10^4$ | $1.50\times10^3$ | $1.56\times10^2$ | 0.066 |
| 3 | $4.63\times10^5$ | $3.47\times10^4$ | $4.00\times10^3$ | 4.81 |
| 4 | $1.48\times10^7$ | $6.91\times10^5$ | $8.38\times10^4$ | 235 |

Table 4.6: Sorting MDD memory requirement and generation time

space needed for the sparse $\widetilde{\mathbf{R}}$ matrix. The MD-based solver causes thrashing for $N = 5$, due to the space needed for the solution vectors. The size of the MD and MDD data structures (except the sorting MDD) is insignificant, relative to the size of the solution vectors. We exercised different implementations of the offset computation, as discussed in Section 4.5. In particular, we employed two methods to compute $\rho(min(s'))$: 1) using the sorting MDD and 2) sorting $s'$ and obtaining $\rho(min(s'))$ from the MDD of $\mathcal{S}_{lumped}$. The latter shows a higher locality in accesses to hardware caches and performed better if the enumeration was forced to proceed in row order. We have not yet come up with a satisfying performance for that enumeration scheme. The former method is superior from a conceptual point of view and performs very well with the DFS (on MD nodes) enumeration procedure. Consequently, we used the first option to obtain the numbers given in Table 4.5.

Table 4.6 gives the size of the sorting MDD and its generation time. Numbers in that table indicate that the size of the sorting MDD is on the order of one solution vector. However, the time to construct that MDD is rather negligible compared to the overall solution time for a CTMC. In fact, there is a trade-off between the space and time complexities of the sorting MDD construction. We claim that by using a more sophisticated algorithm that exploits a modified notion of equality in the generation of an MDD, we can reduce the space complexity of the sorting MDD.

## 4.7 Summary

We described a model composition that is built upon shared state variables by replicate and join operators. The major advantage of state variable sharing in replicate-join composed models is that the replicate operator imposes symmetries in a way that allows an associated CTMC to be lumped. Then, we combined symbolic and model-level lumping techniques to design a symbolic exploration algorithm that generates symbolic structures, namely a multi-valued decision diagram for the state space and a matrix diagram for the state-transition rate matrix of the lumped CTMC, that correspond to the replicate/join composed model. We explained how to obtain the lumped CTMC and how to access elements of its generator matrix $\mathbf{Q}$ without ever explicitly generating $\mathbf{Q}$ in storage. To reach our goal of building an efficiently enumerable MD, we resolved a number of technical issues, such as the development of methods to map state descriptors to indices and scale entries in the case of multiple transitions. The algorithm has been fully implemented and integrated into the Möbius tool.

Possible future work includes further study of the order of components used for the MDDs and MDs. In this chapter, we focused on an order that follows the nesting of operators from top to bottom. This choice allows the symmetries induced by replicate nodes that cause the lumping to impose symmetric patterns on the MDDs as well, such that more nodes are shared in the MDD representation. Clearly other orders are possible, and since the order of components is known to have an impact on the efficiency of the MDD approaches, such orders should be taken into consideration. A second ordering, motivated by the model structure, is to assign low index values to sets of shared variables and high index values to remaining variables of atomic models. Since the atomic models show strong regularities in the case of replicate operators with corresponding potential for sharing, it makes sense to locate them closer to the terminal nodes of an MDD than to its root node.

Our experiments showed that using the sorting MDD along with DFS enumeration (the `AllEdges` iterator) of the elements of the MD improved the time efficiency of enumeration

of $\widetilde{\mathbf{R}}$ by an order of magnitude over the column-oriented MD enumeration. The cost was a rather significant amount of memory (almost as large as the solution vector for the second example model) that the sorting MDD takes. It would be worthwhile to investigate other combinations of data structures and algorithms that can compute $\rho(min(s))$ for $s \in \mathcal{S} \backslash \widetilde{\mathcal{S}}$ with improved, or at least different trade-offs between time and space requirements.

# Chapter 5

# Compositional Lumping of Matrix Diagram Representations of Markovian Models

There are only a few results that address the problem of lumping of Markov chains represented as MDs. In Chapter 4, a model-level lumping technique is proposed for MDs that result from state-sharing compositional models. Like other model-level lumping techniques that can exploit only lumpings that occur due to symmetric composition of components, our technique in Chapter 4 can find only symmetries that occur in MD levels that correspond to identical components of a model. To the best of our knowledge, there is no algorithm that can exploit lumpings in the individual levels of an MD.

In this chapter, we present a new compositional lumping algorithm that is useful for exact and ordinary lumping of Markov chains represented as MDs without knowledge of the modeling formalisms from which the MDs were generated. Our approach relies on local conditions, i.e., conditions on individual levels of the MD. Since our algorithm locally processes MD nodes which often have dramatically smaller sizes than the matrix represented by the MD, it is computationally inexpensive (compared to state-space generation and numerical solution) at the price that it does not necessarily achieve an optimal lumping for the overall CTMC. Our approach is similar to formalism-specific compositional lumping approaches in that they both compute a lumped CTMC by locally applying a lumping algorithm to a part of the model description and replacing that part with its lumped version.

However, the advantage of our algorithm over other formalism-specific ones is that ours is applicable to an MD regardless of the formalism of the model from which the MD has been generated. More specifically, it works for any model formalism for which there is a state-space generation algorithm that generates an MD representation of the underlying

CTMC's state-transition rate matrix [28, 72, 73]. Considering the fact that any matrix represented using Kronecker operators can also be represented using an MD, our compositional lumping algorithm is also applicable to all compositional model formalisms whose underlying CTMCs are specified by Kronecker representations. For example, the compositional lumping algorithms for SANs (Stochastic Automata Networks) [14] and the hierarchical Kronecker representation can be derived from our algorithm.

The important advantage of our compositional technique is that it is complementary to the model-level lumping developed in Chapter 4. In other words, we can apply compositional and model-level lumping techniques simultaneously on a compositional model formalism in which composition is done through sharing state variables. The result is that we will be able to exploit two types of lumping for a composed model that we build using the join and replicate operators of Chapter 4: 1) lumping due to the symmetries present among the various components of the composed model induced by the replicate operator, and 2) lumping present in each individual component. That will enable us to generate potentially smaller MD representations of CTMCs than would be possible using either of the techniques individually. Moreover, either of the two techniques or their combination is complementary to the state-level lumping algorithm developed in Chapter 3 in the sense that any partially lumped CTMC that is computed by the model-level and compositional lumping algorithms can still be fed to the state-level algorithm to obtain the smallest possible lumped CTMC. Remember from Chapter 3 that the state-level algorithm is applicable if the number of states of the input is not prohibitive; that is, if it is less than 30 million states using 1.5 GB RAM.

As we will see in Section 5.4, our algorithm significantly reduces the space and time requirements of MD-based numerical solution algorithms while incurring a negligible time overhead. Our work is related to [49] in that we argue on the level of a block structured matrix to observe lumpability, but unlike the work in [49], it is not limited to Kronecker matrices and stochastic automata networks. It is also related to [14] in that we have a local condition but do not separate local and synchronized actions as was done for the automata

theoretic approach in [14].

The rest of the chapter is organized as follows. We begin, in Section 5.1, with a slight extension of our notation of the MDs and also explain why we can, without loss of generality, focus on three-level MDs instead of MDs with an arbitrary number of levels. Then, in Section 5.2, we present the main results of the chapter, two theorems in which lumpability conditions on the matrices of a single level of an MD are proved to be sufficient to prove that the entire MD is lumpable. Then, in Section 5.3, we give an efficient algorithm that, based on the theorems in Section 5.2, compositionally lumps a CTMC represented by an MD. To demonstrate the applicability of our approach and how it can facilitate the analysis of large models, we use it in Section 5.4 to study the model of a tandem multiprocessor system. We finally summarize in Section 5.5.

## 5.1 Preliminaries

In Definition 2.2, we used the term "lumpable" in reference to MRPs. We also mentioned that if the reward and initial probability vectors satisfy the definition's conditions, we can also speak of a "lumpable" CTMC, its $\mathbf{R}$, or its $\mathbf{Q}$ matrix. In this chapter, we deal with lumping of the MD representations of $\mathbf{R}$ matrices. Therefore, it also makes sense to talk about "lumpability" of MDs. Moreover, since we will define a set of conditions on individual nodes of the MD that leads to the lumpability of an MD (and the MRP we have defined on it), we will also talk about "lumpability" of MD nodes.

### 5.1.1 Extension of MD Notation

In our compositional lumping algorithm for MD representation of CTMCs, we assume that all matrix nodes at the same level of the MD are of size $|\mathcal{S}_c| \times |\mathcal{S}_c|$. At first glance, this assumption seems too restrictive; for example, it is not satisfied when MD nodes' dimensions change due to the projection operation (the "Submat" procedure in [72]) that restricts the

elements of the MD to the reachable state space of a given model.

The apparent limitation is at the notation level and not at a conceptual level. We encounter it because we decided not to adopt a heavyweight notation. Nevertheless, the limitation can be overcome by defining two sets $S_{n_c}, S'_{n_c} \subseteq \mathcal{S}_c$ that specify the set of row and column indices on which the node $R_{n_c}(n_c \in N_c)$ is well-defined. Using that convention, when we enumerate the elements of a given MD (for example, during numerical analysis) we ignore all rows in $\mathcal{S}_c \backslash S_{n_c}$ and columns in $\mathcal{S}_c \backslash S'_{n_c}$ when we access node $R_{n_c}$. However, when we look at $R_{n_c}$ individually (for example, in our compositional lumping algorithm), we assume $r_{n_c,n_{c+1}}(s_c, s'_c) = 0$ for all possible values of $n_{c+1} \in N_{c+1}$, and if $s_c \in \mathcal{S}_c \backslash S_{n_c}$ or $s'_c \in \mathcal{S}_c \backslash S'_{n_c}$.

### 5.1.2 3-Level MDs vs. Arbitrary MDs

In the following paragraphs, we describe how, by merging adjacent levels of an MD in bottom-up and top-down manners, we can consider in our arguments, without loss of generality and in order to avoid an overwhelming notation, MDs with 3 levels instead of MDs with an arbitrary number of levels.

Recall how we merged adjacent levels of an $m$-level MD from the bottom up in Section 2.3.2. Much as levels can be merged in a bottom-up manner, we can reduce the number of levels by merging in a top-down manner, i.e., starting at the top level. For example, we can merge levels 1 and 2 and obtain an $(m-1)$-level MD. The resulting MD has a new root

node $\bar{R}_1$ such that $\bar{s}_1 = (s_1, s_2)$, $\bar{s}'_1 = (s'_1, s'_2)$, and

$$\begin{aligned}
\bar{R}_1(\bar{s}_1, \bar{s}'_1) &= \sum_{n_2 \in N_2} r_{1,n_2}(s_1, s'_1) R_{n_2} \\
&= \sum_{n_2 \in N_2} r_{1,n_2}(s_1, s'_1) \sum_{n_3 \in N_3} r_{n_2,n_3}(s_2, s'_2) R_{n_3} \\
&= \sum_{n_3 \in N_3} \left[ \sum_{n_2 \in N_2} r_{1,n_2}(s_1, s'_1)\, r_{n_2,n_3}(s_2, s'_2) \right] R_{n_3} \\
&= \sum_{n_3 \in N_3} \bar{r}_{1,n_3}(\bar{s}_1, \bar{s}'_1) R_{n_3}
\end{aligned}$$

where $\bar{r}_{1,n_3}(\bar{s}_1, \bar{s}'_1) = \sum_{n_2 \in N_2} r_{1,n_2}(s_1, s'_1) r_{n_2,n_3}(s_2, s'_2)$. By iterating that procedure $c-1$ times we can merge the first levels 1 through $c$.

In our compositional lumping algorithm for MDs, we need to test an arbitrary level $c$ for lumpability conditions local to that level. Therefore, it is natural to merge all levels $1, \ldots, c-1$ into a single level and all levels $c+1, \ldots, m$ into another single level and obtain a 3-level MD when $1 < c < m$. The cases where $c = 1$ (and $c = m$) need special attention. We first add an artificial level 0 (or $m+1$) to the MD that has a single node of size $1 \times 1$ consisting of entry 1, and establish appropriate pointers from level 0 to 1 (or $m$ to $m+1$) such that the new MD represents the same overall matrix. Then, we perform the merging operations mentioned above. Eventually, the merging leads us to consider, for most of our discussion and without loss of generality, an MD of 3 levels and to focus on level 2 for local lumping purposes instead of considering an MD of $m$ levels and focusing on level $c$. Once again, note that the sole purpose of our merging argument is to simplify the notation and understanding of the material. As we will see, when we finally present the compositional lumping algorithm, we do not perform any merging operation.

## 5.2 Compositional Lumping of Matrix Diagrams

As we mentioned earlier, our focus is on describing a lumping algorithm that can be applied directly to an MD representation of a CTMC. We will do so by applying lumping to matrices of each level of the MD and replacing them with possibly smaller ones. The algorithm is eventually applied to all levels of the MD.

In this section, we describe the concepts based on which the compositional lumping works. In particular, we will describe two sets of sufficient lumpability conditions (one for ordinary and one for exact lumping) on the nodes of one level of the MD, and prove that they satisfy lumpability conditions on the overall MD.

### 5.2.1 Augmentation of MDs to MRPs

In Section 2.3, we showed how an MD can represent the state transition rate matrix of a CTMC. To discuss lumping conditions according to Theorem 2.1, we need to augment the CTMC to an MRP by specifying the rewards and the initial probability distribution. In particular, to discuss local lumping conditions, the reward vector and probability distribution have to be decomposable. More specifically, $\mathbf{r}$ is an arbitrary reward vector on $\mathcal{S}$, and when we discuss ordinary lumping, we restrict its representation to $\mathbf{r}(s) = g(f_1(s_1), f_2(s_2), f_3(s_3))$ (i.e., a function built upon functions of substates at each level), in which $s = (s_1, s_2, s_3)$, $g : \mathbb{R}^3 \rightarrow \mathbb{R}$, $f_1 : \mathcal{S}_1 \rightarrow \mathbb{R}$, $f_2 : \mathcal{S}_2 \rightarrow \mathbb{R}$, and $f_3 : \mathcal{S}_3 \rightarrow \mathbb{R}$. Likewise, $\boldsymbol{\pi}^{\text{ini}}$ is also an arbitrary initial probability distribution on $\mathcal{S}$, and when we discuss exact lumping, we restrict its representation to $\boldsymbol{\pi}^{\text{ini}}(s) = g_\pi(f_{\pi,1}(s_1), f_{\pi,2}(s_2), f_{\pi,3}(s_3))$, in which $g_\pi : \mathbb{R}^3 \rightarrow [0, 1]$, $f_{\pi,1} : \mathcal{S}_1 \rightarrow \mathbb{R}$, $f_{\pi,2} : \mathcal{S}_2 \rightarrow \mathbb{R}$, and $f_{\pi,3} : \mathcal{S}_3 \rightarrow \mathbb{R}$.

Notice that the "restriction" we mentioned above is on the way we *represent* $\mathbf{r}$ or $\boldsymbol{\pi}^{\text{ini}}$. The sole purpose of that representation is to simplify the expression of the lumping conditions on those vectors. In fact, that representation does not limit the expressiveness of $\mathbf{r}$ or $\boldsymbol{\pi}^{\text{ini}}$. In fact, the representation is as expressive as the MTMDD data structure, and therefore can

be used to represent arbitrary vectors. In our implementations, we are using MTMDDs to represent those vectors.

### 5.2.2 Local Equivalence Relations $\approx_{lo}$ and $\approx_{le}$

In the rest of this section, we first present two local equivalence relations $\approx_{lo}$ and $\approx_{le}$ on the state space of level 2 of an MD. We then prove that lumping level 2 of the MD with respect to each of those relations results in an MD that is lumped with respect to (global) equivalence relations $\approx_{go}$ or $\approx_{ge}$, which are constructed upon $\approx_{lo}$ and $\approx_{le}$.

**Definition 5.1** *Local equivalences $\approx_{lo}$ and $\approx_{le}$ (and corresponding partition $\mathcal{P}_{lo}$ and $\mathcal{P}_{le}$) are defined on $\mathcal{S}_2$ as follows:*

$$s_2 \approx_{lo} \hat{s}_2 \quad \textit{if} \quad f_2(s_2) = f_2(\hat{s}_2) \tag{5.1}$$

$$\textit{and} \quad \forall\, n_2 \in N_2, C_2 \in \mathcal{P}_{lo} : \mathbf{R}_{n_2}(s_2, C_2) = \mathbf{R}_{n_2}(\hat{s}_2, C_2) \tag{5.2}$$

$$s_2 \approx_{le} \hat{s}_2 \quad \textit{if} \quad f_{\pi,2}(s_2) = f_{\pi,2}(\hat{s}_2) \tag{5.3}$$

$$\textit{and} \quad \forall\, n_2 \in N_2 : \mathbf{R}_{n_2}(s_2, \mathcal{S}_2) = \mathbf{R}_{n_2}(\hat{s}_2, \mathcal{S}_2) \tag{5.4}$$

$$\textit{and} \quad \forall\, n_2 \in N_2, C_2 \in \mathcal{P}_{le} : \mathbf{R}_{n_2}(C_2, s_2) = \mathbf{R}_{n_2}(C_2, \hat{s}_2) \tag{5.5}$$

■

In $\approx_{lo}$ and $\approx_{le}$, the letters $l$, $o$, and $e$ stand for local, ordinary, and exact, respectively. It is important to observe that equivalence relations $\approx_{lo}$ and $\approx_{le}$ are not unique. In other words, there can be more than one equivalence relation that satisfies each set of the sufficient conditions in the definition. In particular, the definition does not specify whether partitions $\mathcal{P}_{lo}$ and $\mathcal{P}_{le}$ are the coarsest possible.

Note that the symbol "=" in Eqs. (5.2), (5.4), and (5.5) means equality of matrices (all elements equal), and that the matrices on both sides of each equality are at most of size $|\mathcal{S}_3| \times |\mathcal{S}_3|$. The following proposition states when the equalities in (5.2), (5.4), and (5.5) will hold in terms of matrix elements.

**Proposition 5.1** *In Definition 5.1,*

*a. Eq. (5.2) is equivalent to*

$$\forall\, n_2 \in N_2, C_2 \in \mathcal{P}_{lo}, s_3, s_3' \in \mathcal{S}_3 :$$
$$\sum_{s_2' \in C_2} \sum_{n_3 \in N_3} r_{n_2,n_3}(s_2, s_2')\mathbf{R}_{n_3}(s_3, s_3') = \sum_{s_2' \in C_2} \sum_{n_3 \in N_3} r_{n_2,n_3}(\hat{s}_2, s_2')\mathbf{R}_{n_3}(s_3, s_3')$$

*b. Eq. (5.4) is equivalent to*

$$\forall\, n_2 \in N_2, s_3, s_3' \in \mathcal{S}_3 :$$
$$\sum_{s_2' \in \mathcal{S}_2} \sum_{n_3 \in N_3} r_{n_2,n_3}(s_2, s_2')\mathbf{R}_{n_3}(s_3, s_3') = \sum_{s_2' \in \mathcal{S}_2} \sum_{n_3 \in N_3} r_{n_2,n_3}(\hat{s}_2, s_2')\mathbf{R}_{n_3}(s_3, s_3')$$

*c. Eq. (5.5) is equivalent to*

$$\forall\, n_2 \in N_2, C_2 \in \mathcal{P}_{le}, s_3, s_3' \in \mathcal{S}_3 :$$
$$\sum_{s_2' \in C_2} \sum_{n_3 \in N_3} r_{n_2,n_3}(s_2', s_2)\mathbf{R}_{n_3}(s_3, s_3') = \sum_{s_2' \in C_2} \sum_{n_3 \in N_3} r_{n_2,n_3}(s_2', \hat{s}_2)\mathbf{R}_{n_3}(s_3, s_3')$$

■

### 5.2.3 Global Equivalence Relations $\approx_{go}$ and $\approx_{ge}$

In Definition 5.2, we "extend" $\approx_{lo}$ and $\approx_{go}$ to define another pair of equivalence relations $\approx_{go}$ and $\approx_{ge}$ that imply a partition on $\mathcal{S}$, the state space of the overall MD.

**Definition 5.2** *Global equivalences $\approx_{go}$ and $\approx_{ge}$ (and corresponding partitions $\mathcal{P}_{go}$ and $\mathcal{P}_{ge}$) are defined on $\mathcal{S}$ for states $s = (s_1, s_2, s_3), \hat{s} = (\hat{s}_1, \hat{s}_2, \hat{s}_3) \in \mathcal{S}$ as*

*a) $s \approx_{go} \hat{s}$ if $s_1 = \hat{s}_1, s_2 \approx_{lo} \hat{s}_2$, and $s_3 = \hat{s}_3$,*

*b) $s \approx_{ge} \hat{s}$ if $s_1 = \hat{s}_1, s_2 \approx_{le} \hat{s}_2$, and $s_3 = \hat{s}_3$.* ■

Notice that for each class $C \in \mathcal{P}_{go}$ (resp., $C \in \mathcal{P}_{ge}$) there is a corresponding class $C_2 \in \mathcal{P}_{lo}$ ($C_2 \in \mathcal{P}_{go}$) such that $C_2 = \{s_2|\ (s_1, s_2, s_3) \in C\}$. On the other hand, given $s_1$ and $s_3$ and

a class $C_2 \in \mathcal{P}_{lo}$ ($C_2 \in \mathcal{P}_{le}$), there is a corresponding class $C \in \mathcal{P}_{go}$ ($C \in \mathcal{P}_{ge}$) such that $C = \{(s_1, s_2, s_3) |\ s_2 \in C_2\}$.

In Theorems 5.1 and 5.2, we prove that the $\approx_{go}$ and $\approx_{ge}$ equivalence relations satisfy the ordinary and exact lumpability conditions (based on Theorem 2.1) on an MD, respectively. That implies that the set of conditions on the matrices and reward vector of level 2 of a 3-level MD given by $\approx_{lo}$ and extended to the complete MD by $\approx_{go}$ implies ordinary lumpability on the MD. Similarly, the set of conditions on the matrices and initial probability distribution vector of level 2 of a 3-level MD given by $\approx_{le}$ and extended to the complete MD by $\approx_{ge}$ implies exact lumpability on the MD.

Note that equivalence relations $\approx_{go}$ and $\approx_{ge}$ are not unique either, because they are defined based on (non-unique) equivalence relations $\approx_{lo}$ and $\approx_{le}$, respectively. In particular, the definition does not specify whether $\approx_{go}$ and $\approx_{ge}$ are the coarsest possible partitions on $\mathcal{S}$. However, Theorems 5.1 and 5.2 hold for any equivalence relation that satisfies the conditions in Definitions 5.1 and 5.2.

**Theorem 5.1** $(\mathcal{S}, \mathbf{Q}, \mathbf{r}, \boldsymbol{\pi}^{\text{ini}})$ *is ordinarily lumpable with respect to partition* $\mathcal{P}_{go}$ *(and its corresponding equivalence relation* $\approx_{go}$*).*

**Proof.** Based on Theorem 2.1, we need to prove:

$$\forall\, s \approx_{go} \hat{s}\ : g(f_1(s_1), f_2(s_2), f_3(s_3)) = g(f_1(\hat{s}_1), f_2(\hat{s}_2), f_3(\hat{s}_3)) \quad \text{and} \tag{5.6}$$

$$\forall\, s \approx_{go} \hat{s}\ \forall\, C \in \mathcal{P} : \mathbf{R}(s, C) = \mathbf{R}(\hat{s}, C) \tag{5.7}$$

Consider two states $s = (s_1, s_2, s_3) \approx_{go} \hat{s} = (\hat{s}_1, \hat{s}_2, \hat{s}_3)$. By Definition 5.2, we have $s_1 = \hat{s}_1$ and $s_3 = \hat{s}_3$, and by Definition 5.1, we know that $f_2(s_2) = f_2(\hat{s}_2)$. Therefore, (5.6) holds.

To prove (5.7), we transform the left-hand side of the equality in a number of steps to its right-hand side, as follows:

$$\begin{aligned}
\mathbf{R}(s, C) &= \sum_{s' \in C} \mathbf{R}(s, s') \\
&= \sum_{s' \in C} \sum_{n_2 \in N_2} r_{1,n_2}(s_1, s'_1) \sum_{n_3 \in N_3} r_{n_2,n_3}(s_2, s'_2) \mathbf{R}_{n_3}(s_3, s'_3) \\
\text{[by Definition 5.2]} \quad &= \sum_{s'_2 \in C_2} \sum_{n_2 \in N_2} r_{1,n_2}(s_1, s'_1) \sum_{n_3 \in N_3} r_{n_2,n_3}(s_2, s'_2) \mathbf{R}_{n_3}(s_3, s'_3) \\
\text{[moving the sums]} \quad &= \sum_{n_2 \in N_2} r_{1,n_2}(s_1, s'_1) \underbrace{\sum_{s'_2 \in C_2} \sum_{n_3 \in N_3} r_{n_2,n_3}(s_2, s'_2) \mathbf{R}_{n_3}(s_3, s'_3)}_{A} \\
[A = B \text{ by Prop. (5.1)}] \quad &= \sum_{n_2 \in N_2} r_{1,n_2}(s_1, s'_1) \underbrace{\sum_{s'_2 \in C_2} \sum_{n_3 \in N_3} r_{n_2,n_3}(\hat{s}_2, s'_2) \mathbf{R}_{n_3}(s_3, s'_3)}_{B} \\
\text{[reverse argument]} \quad &= \sum_{s' \in C} \mathbf{R}(\hat{s}, s') = \mathbf{R}(\hat{s}, C)
\end{aligned}$$

■

**Theorem 5.2** $(\mathcal{S}, \mathbf{Q}, \mathbf{r}, \boldsymbol{\pi}^{\text{ini}})$ *is exactly lumpable with respect to partition* $\mathcal{P}_{ge}$ *(and its corresponding equivalence relation* $\approx_{ge}$*).*

**Proof.** Based on Theorem 2.1, we need to prove:

$$\forall\, s \approx_{ge} \hat{s}: \quad g_\pi(f_{\pi,1}(s_1), f_{\pi,2}(s_2), f_{\pi,3}(s_3)) = g_\pi(f_{\pi,1}(\hat{s}_1), f_{\pi,2}(\hat{s}_2), f_{\pi,3}(\hat{s}_3)) \tag{5.8}$$

$$\text{and} \qquad \mathbf{R}(s, \mathcal{S}) = \mathbf{R}(\hat{s}, \mathcal{S}) \tag{5.9}$$

$$\text{and} \qquad \forall\, C \in \mathcal{P} : \mathbf{R}(C, s) = \mathbf{R}(C, \hat{s}) \tag{5.10}$$

Eqs. (5.8) and (5.10) can respectively be proved much like (5.6) and (5.7) in Theorem 5.1. To prove Eq. (5.9), we use the same approach that we used in the proof of Theorem 5.1, but replace $C$ with $\mathcal{S}$. In particular,

$$\mathbf{R}(s, \mathcal{S}) = \sum_{s' \in \mathcal{S}} \mathbf{R}(s, s')$$

$$
\begin{aligned}
& = \sum_{(s'_1,s'_2,s'_3)\in\mathcal{S}} \sum_{n_2\in N_2} r_{1,n_2}(s_1,s'_1) \sum_{n_3\in N_3} r_{n_2,n_3}(s_2,s'_2)\mathbf{R}_{n_3}(s_3,s'_3) \\
\text{[moving the sums]} \quad & = \sum_{s'_1\in\mathcal{S}_1}\sum_{n_2\in N_2} r_{1,n_2}(s_1,s'_1) \sum_{s'_3\in\mathcal{S}_3} \underbrace{\sum_{s'_2\in\mathcal{S}_2}\sum_{n_3\in N_3} r_{n_2,n_3}(s_2,s'_2)\mathbf{R}_{n_3}(s_3,s'_3)}_{A} \\
[A = B \text{ by Prop. 5.1}] \quad & = \sum_{s'_1\in\mathcal{S}_1}\sum_{n_2\in N_2} r_{1,n_2}(\hat{s}_1,s'_1) \sum_{s'_3\in\mathcal{S}_3} \underbrace{\sum_{s'_2\in\mathcal{S}_2}\sum_{n_3\in N_3} r_{n_2,n_3}(\hat{s}_2,s'_2)\mathbf{R}_{n_3}(\hat{s}_3,s'_3)}_{B} \\
\text{[reverse argument]} \quad & = \sum_{s'\in\mathcal{S}} \mathbf{R}(\hat{s},s') = \mathbf{R}(\hat{s},\mathcal{S})
\end{aligned}
$$

■

Theorems 5.1 and 5.2 ensure the lumpability of an MRP given a set of sufficient conditions on the lumpability of the matrices of a specific level of the corresponding MD. The next natural question is how to efficiently compute a locally lumpable partition $\mathcal{P}_{lo}$ or $\mathcal{P}_{le}$ that satisfies the conditions in those theorems.[1] It is answered in the following section.

## 5.3 Compositional Lumping Algorithm for Matrix Diagrams

We now know, by Theorems 5.1 and 5.2, that given an equivalence relation $\approx_{lo}$ ($\approx_{le}$) on the states of one level, MRP $M$ is ordinarily (exactly) lumpable with respect to $\approx_{go}$ ($\approx_{ge}$), derived from $\approx_{lo}$ ($\approx_{le}$). Therefore, the problem of lumping $M$ is reduced to computing $\approx_{lo}$ ($\approx_{le}$). This section describes an algorithm to compute these relations. Given an MD with its associated reward and initial probability distribution vectors, we run the algorithm for all levels of the MD, once for each level, to obtain a lumped MRP.

The algorithm we develop is based on the lumping algorithm presented in Chapter 3, where we gave an efficient algorithm based on the partition refinement approach for computing the optimal (i.e., coarsest) equivalence relation that gives an ordinary or exact lumping on an MRP. In this chapter, we use a different definition for the $K$ function, and hence, a

[1] Notice that $\mathcal{P}_{lo}$ and $\mathcal{P}_{le}$ are not unique partitions.

different data type $T$ than Eq. (3.4). That enables us to compute a partition on $\mathcal{S}_2$ that satisfies Definition 5.1 rather than a partition on $\mathcal{S}$ that satisfies Definition 2.2. It is essential to note that we do not apply the algorithm to $\mathbf{R}$ (the overall state-transition rate matrix). Instead, we apply it to the individual nodes of the MD, which are often very smaller than $\mathbf{R}$.

### 5.3.1 Computing $\approx_{lo}$ and $\approx_{le}$

In Chapter 3, we used the ComputeCoarsestPartition procedure (Figure 3.2) to compute the coarsest partition that satisfies the conditions in Definition 2.2. To use that procedure to compute equivalence relations $\approx_{lo}$ and $\approx_{le}$ that satisfy Definition 5.1, we need to make a number of changes in the way we use it. Those changes can be recognized by comparing the conditions of Definition 5.1 with Definition 2.2.

Observe the differences and similarities of Eq. (5.2) and Eq. (2.2) for ordinary lumping and Eq. (5.5) and Eq. (2.4) for exact lumping. First of all, Eqs. (5.2) and (5.5) hold for matrices $\mathbf{R}_{n_2}$, whereas in Eq. (2.2) and Eq. (2.4) there is only one matrix $\mathbf{Q}$ involved. Therefore, we need to call ComputeCoarsestPartition$(\mathbf{R}_{n_2}, \mathcal{S}_2, \mathcal{P}_2^{\text{ini}})$ for each $n_2 \in N_2$, instead of calling ComputeCoarsestPartition$(\mathbf{Q}, \mathcal{S}, \mathcal{P}^{\text{ini}})$. We will explain later how to compute $\mathcal{P}_2^{\text{ini}}$, the initial partition of $\mathcal{S}_2$.

Second, we need to find the appropriate choices of function $K$ that satisfy Eqs. (5.2) and (5.5). Again, comparing Eq. (5.2) with Eq. (2.2) leads us to the first obvious choice of $K$ as $K(\mathbf{R}_{n_2}, s_2, \Psi_2) = \mathbf{R}_{n_2}(s_2, \Psi_2)$ where $\Psi_2 \subseteq \mathcal{S}_2$. Similarly, for exact lumping, the comparison of Eq. (5.5) with Eq. (2.4) leads us to $K(\mathbf{R}_{n_2}, s_2, \Psi_2) = \mathbf{R}_{n_2}(\Psi_2, s_2)$ to satisfy Eq. (5.5).

Therefore, data type $T$ is the set of matrices of size at most $|\mathcal{S}_3| \times |\mathcal{S}_3|$. Considering the fact that level 3 of the MD is built through merging of at most $m-1$ levels of the original MD, the computation of function $K$ and equality testing for $T$ are prohibitively time-consuming for levels $c < m$. Therefore, we do not follow that approach.

The other choice is for $K$ to compute a formal sum represented as a set of (coefficient,

node index) pairs. Note that for any node $R_{n_2}$

$$\begin{aligned}
\mathbf{R}_{n_2}(s_2, \Psi_2) &= \sum_{s_2' \in \Psi_2} \sum_{n_3 \in N_3} r_{n_2,n_3}(s_2, s_2') \cdot \mathbf{R}_{n_3} \\
&= \sum_{n_3 \in N_3} \underbrace{\left( \sum_{s_2' \in \Psi_2} r_{n_2,n_3}(s_2, s_2') \right)}_{= r_{n_2,n_3}(s_2, \Psi_2)} \cdot \mathbf{R}_{n_3}, \text{ and} \\
\mathbf{R}_{n_2}(\Psi_2, s_2) &= \sum_{s_2' \in \Psi_2} \sum_{n_3 \in N_3} r_{n_2,n_3}(s_2', s_2) \cdot \mathbf{R}_{n_3} \\
&= \sum_{n_3 \in N_3} \underbrace{\left( \sum_{s_2' \in \Psi_2} r_{n_2,n_3}(s_2', s_2) \right)}_{= r_{n_2,n_3}(\Psi_2, s_2)} \cdot \mathbf{R}_{n_3},
\end{aligned}$$

which implies

**Lemma 5.1** *For all $s_2, \hat{s}_2 \in \mathcal{S}_2$, all $\Psi_2 \subseteq \mathcal{S}_2$, and all $n_2 \in N_2$,*

*a.* $\mathbf{R}_{n_2}(s_2, \Psi_2) = \mathbf{R}_{n_2}(\hat{s}_2, \Psi_2)$ *if* $\forall\, n_3 \in N_3 : r_{n_2,n_3}(s_2, \Psi_2) = r_{n_2,n_3}(\hat{s}_2, \Psi_2)$

*b.* $\mathbf{R}_{n_2}(\Psi_2, s_2) = \mathbf{R}_{n_2}(\Psi_2, \hat{s}_2)$ *if* $\forall\, n_3 \in N_3 : r_{n_2,n_3}(\Psi_2, s_2) = r_{n_2,n_3}(\Psi_2, \hat{s}_2)$.

The conditions are only sufficient since 1) a weighted sum of matrices may be equal even if the individual terms differ, and 2) $\mathbf{R}_{n_i} = \mathbf{R}_{n_i'} \Leftrightarrow n_i = n_i'$ does not necessarily hold for an arbitrary MD. Canonical MDs [73] are a particular subclass of MDs in which the expression is true; $\mathbf{R}_{n_i}$ is uniquely represented by $n_i$. Nevertheless, efficiency of MDs is based on sharing of equal nodes, so one can expect that having two different nodes $R_{n_i}$ and $R_{n_i'}$ that represent the same matrix $\mathbf{R}_{n_i} = \mathbf{R}_{n_i'}$ is uncommon in MDs.

Based on the above observation, we can localize the comparison at level 2 and set

$$\begin{aligned}
&K(R_{n_2}, s_2, \Psi_2) \qquad (5.11) \\
&\quad = \begin{cases}
\{(n_3, r_{n_2,n_3}(s_2, \Psi_2))|\ n_3 \in N_3, r_{n_2,n_3}(s_2, \Psi_2) \neq 0\} & \text{for ordinary lumping} \\
\{(n_3, r_{n_2,n_3}(\Psi_2, s_2))|\ n_3 \in N_3, r_{n_2,n_3}(\Psi_2, s_2) \neq 0\} & \text{for exact lumping,}
\end{cases}
\end{aligned}$$

which is a set representation of the formal sum $\sum_{n_3 \in N_3} r_{n_2,n_3}(s_2, \Psi_2) \cdot R_{n_3}$ or its counterpart $\sum_{n_3 \in N_3} r_{n_2,n_3}(\Psi_2, s_2) \cdot R_{n_3}$ with references to nodes $R_{n_3}$, and not to matrices $\mathbf{R}_{n_3}$. Two formal sums are equal if their corresponding sets are equal. That means that the algorithm is applied locally only at nodes in $N_2$ of size $|\mathcal{S}_2| \times |\mathcal{S}_2|$ and not at matrices of size at most $|\mathcal{S}_2 \times \mathcal{S}_3| \times |\mathcal{S}_2 \times \mathcal{S}_3|$.

Using local lumpability conditions for a level that are only sufficient (instead of both sufficient and necessary) leads to an improved time complexity for the algorithm, but also prevents the algorithm from generating the coarsest possible lumpable partition for that level. That means there is a trade-off between time complexity and coarseness of the computed partition when the algorithm is used on an $m$-level MD. For level $m$, the sufficient condition is also necessary, because level $m$ is a set of simple real-valued matrices, and therefore the coarsest possible partition is computed. For levels $c < m$, the algorithm uses sufficient conditions, and consequently does not necessarily compute the coarsest possible partition for that level. That means that our algorithm does not necessarily generate the smallest possible lumped MD. However, its time complexity is smaller than the time complexity of an algorithm that would be based on both sufficient and necessary conditions, that is, comparison of weighted sums of matrices $\mathbf{R}_{n_3}$. Due to that trade-off, one may investigate the particular level $c$ at which comparison of a weighted sum of matrices is prohibitive. Due to the dramatic growth of the matrices, we expect that level to be very close to $m$.

So far we have been describing the theorems and the changes we needed to make in the procedures of Chapter 3 for the case of a 3-level MD. Now that we have all the pieces and are ready to present the compositional lumping algorithm, we switch to the $m$-level MD case. We have so far shown how to compute a partition that satisfies Eqs. (5.1) and (5.4) for one instance of $n_c \in N_c\,(1 \leq c \leq m)$ (or one instance of $n_2 \in N_2$ in the 3-level MD view). The last step in computing $\mathcal{P}_{lo}$ and $\mathcal{P}_{le}$ is to find a partition that satisfies those equations for all nodes in $N_c$. We do so in ComputeCoarsestPartitionForLevel (Figure 5.1) by a fixed-point iteration. More specifically, we apply the ComputeCoarsestPartition

```
ComputeCoarsestPartitionForLevel(P_c^ini, c)
 1  P_c := P_c^ini
 2  repeat
 3     P'_c := P_c
 4     for each n_c ∈ N_c
 5        P'_c := ComputeCoarsestPartition(R_{n_c}, S_c, P'_c)
 6  until P_c = P'_c
 7  return P_c
```

Figure 5.1: Computing $\approx_{lo}$ and $\approx_{le}$ for level $c$

```
CompositionalLump
 1  for c := 1 to m
 2     Compute P_c^ini
 3     P_c = ComputeCoarsestPartitionForLevel(P_c^ini, c)
 4     for each n_c ∈ N_c
 5        R̃_{n_c} := ComputeLumpedNode(R_{n_c}, S_c, P_c)
 6        Replace R_{n_c} with R̃_{n_c} in MD
 7     Compute lumped version of rewards
       and initial probabilities at level i
```

Figure 5.2: Compositional lumping algorithm for MD

algorithm repeatedly to all nodes in level $c$ until they are all lumpable with respect to the same partition $\mathcal{P}$.

### 5.3.2 Overall Algorithm

The final step of our algorithm for lumping an MD is to use ComputeCoarsestPartitionForLevel to compute lumpable partitions for each level starting from an initial partition (lines 2-3 of Figure 5.2), lump every node with respect to the partition corresponding to its level (line 5), replace each node with its lumped version (line 6), and finally, compute $\widetilde{\boldsymbol{\pi}}^{\text{ini}}$ and $\widetilde{\mathbf{r}}$ (line 7).

Line 2 computes the initial partition for level $c$ based on whether we are computing ordinary or exact lumping. For ordinary lumping, Eq. (5.1) determines how we should

compute $\mathcal{P}_c^{\text{ini}}$: it is the coarsest partition on $\mathcal{S}_c$ such that:

$$\forall\, C \in \mathcal{P}_c^{\text{ini}}, s_c, s_c' \in C : f_c(s_c) = f_c(s_c').$$

For exact lumping, Eqs. (5.3) and (5.4) determine how we should compute $\mathcal{P}_c^{\text{ini}}$. Using Lemma 5.1, Eq. (5.4) holds if $r_{n_c,n_{c+1}}(s_c, \mathcal{S}_c) = r_{n_c,n_{c+1}}(s_c', \mathcal{S}_c)$. Hence, for exact lumping, $\mathcal{P}_c^{\text{ini}}$ is the coarsest partition on $\mathcal{S}_c$ such that

$$\begin{aligned}\forall\, C \in \mathcal{P}_c^{\text{ini}}, s_c, s_c' \in C, n_c \in N_c, n_{c+1} \in N_{c+1} :& \\ f_{\pi,c}(s_c) = f_{\pi,c}(s_c') \text{ and } r_{n_c,n_{c+1}}(s_c, \mathcal{S}_c) &= r_{n_c,n_{c+1}}(s_c', \mathcal{S}_c).\end{aligned}$$

For both cases, $\mathcal{P}_c^{\text{ini}}$ can be computed, in terms of the implementation, using a binary search tree that clusters the substates that belong to the same class of $\mathcal{P}_c^{\text{ini}}$.

ComputeLumpedNode (Figure 5.3) is the same as ComputeLumpedCTMC (Figures 3.7(a) and 3.8(a)) except that it computes the lumped version of an MD node instead of a CTMC. In line 5 of Figures 5.3(a) and 5.3(b), we have used the version of function $K$ that is extended to the $m$-level MD case. Based on Eq. (5.11), we have:

$$\begin{aligned}&K(R_{n_c}, s_c, \Psi_c) \qquad (5.12)\\ &= \begin{cases} \{(n_{c+1}, r_{n_c,n_{c+1}}(s_c, \Psi_c))|\ n_{c+1} \in N_{c+1}, r_{n_c,n_{c+1}}(s_c, \Psi_c) \neq 0\} & \text{ordinary lumping} \\ \{(n_{c+1}, r_{n_c,n_{c+1}}(\Psi_c, s_c))|\ n_{c+1} \in N_{c+1}, r_{n_c,n_{c+1}}(\Psi_c, s_c) \neq 0\} & \text{exact lumping} \end{cases}\end{aligned}$$

where $s_c \in \mathcal{S}_c$, $\Psi_c \subseteq \mathcal{S}_c$, and $n_c \in N_c$. Line 5 assigns to $\widetilde{R}(\tilde{i}, \tilde{j})$ the value of function $K$, which according to Eq. (5.12) is a set of (node reference, rate) pairs. The result is that for ordinary lumping and for any pair $(n_{c+1}, r_{n_c,n_{c+1}}(s_c, C_{\tilde{j}})) \in K(R_{n_c}, s_c, C_{\tilde{j}})$, $r_{n_c,n_{c+1}}(s_c, C_{\tilde{j}})$ is assigned to $\widetilde{r}_{n_c,n_{c+1}}(\tilde{i}, \tilde{j})$. Similarly, for exact lumping for any pair $(n_{c+1}, r_{n_c,n_{c+1}}(C_{\tilde{i}}, s_c)) \in K(R_{n_c}, s_c, C_{\tilde{i}})$, $r_{n_c,n_{c+1}}(C_{\tilde{i}}, s_c)$ is assigned to $\widetilde{r}_{n_c,n_{c+1}}(\tilde{i}, \tilde{j})$.

```
ComputeLumpedNode(R_{n_c}, S_c, P_c)
1  for each ĩ ∈ {1, ..., |P_c|}
2      R̃(ĩ, ĩ) := 0
3  for each ĩ, j̃ ∈ {1, ..., |P_c|}, ĩ ≠ j̃
4      s_c := arbitrary element of C_ĩ ∈ P_c
5      R̃(ĩ, j̃) = K(R_{n_c}, s_c, C_j̃)
6  return R̃
```

(a) Ordinary lumping

```
ComputeLumpedNode(R_{n_c}, S_c, P_c)
1  for each ĩ ∈ {1, ..., |P_c|}
2      R̃(ĩ, ĩ) := 0
3  for each ĩ, j̃ ∈ {1, ..., |P_c|}, ĩ ≠ j̃
4      s_c := arbitrary element of C_j̃ ∈ P_c
5      R̃(ĩ, j̃) = K(R_{n_c}, s_c, C_ĩ)
6  return R̃
```

(b) Exact lumping

Figure 5.3: ComputeLumpedNode's pseudocode for ordinary and exact lumping

## 5.4 Performance Results

In the previous two sections, we presented the theory on which our algorithm is based, and then described how we designed our compositional lumping algorithm by modifying a partition refinement algorithm for state-level lumping. In this section, we briefly explain our implementation of the algorithm, describe its integration into the symbolic state-space generator (symbolic SSG) algorithm of Chapter 4, and illustrate its use through an example model of a tandem multiprocessor system with load-balancing and failure and repair operations. The results show that compositional lumping of MDs is indeed practical and enables us to numerically solve large Markovian models with much smaller time and space requirements than would be necessary using only symbolic data structures or lumping techniques.

### 5.4.1 Implementation in Möbius

To implement the MD-based compositional lumping algorithm described in the last section, we start with the symbolic SSG algorithm of Chapter 4, which generates the MD representa-

tion of the state transition rate matrix of a high-level Markovian model. Our implementation of the lumping algorithm provides a number of functions (like the procedures we described in Section 5.3) that operate on the MD of the original CTMC and generate the lumped MD, which represents the lumped CTMC. As we mentioned in Chapter 4, our symbolic SSG algorithm works only for models composed through sharing of state variables. However, that does not mean that our implementation of the compositional lumping algorithm is only applicable to those types of models. In fact, since the compositional lumping algorithm we described does not make any assumption about the high-level model formalism that generated the MD, it can be applied to any MD, regardless of the high-level model from which it was generated.

The implementation is based on Möbius version 1.6.0 and was compiled with the gcc 3.3 compiler with the -O3 optimization option. All experiments were conducted using an Athlon XP2400 machine with 1.5 GB of main memory running Linux.

### 5.4.2 Tandem Multiprocessor System

We consider a tandem multiprocessor system that consists of two subsystems: MSMQ and hypercube. Each subsystem has a number of servers, an input pool of jobs that are waiting to be serviced, and an output pool of jobs that have already been serviced by a server in the subsystem. Each subsystem takes jobs from its input pool, processes them using its servers, and passes them to its output pool. The interaction between the two subsystems is done through each subsystem sharing its output pool with the input pool of the other. In other words, jobs in one subsystem enter the input pool of the other upon completion of service. The system is closed in the sense that there is always a constant number $J$ of jobs in it.

The first subsystem is an MSMQ (Multi-Server Multi-Queue) polling-based queuing system with 3 identical servers and 4 identical queues, as described in [68]. The complete SAN (Stochastic Activity Network) [84] specification of the subsystem's model is given in Appendix A.3. Figure 5.4 shows a high-level view of the cyclic arrangement of the queues and

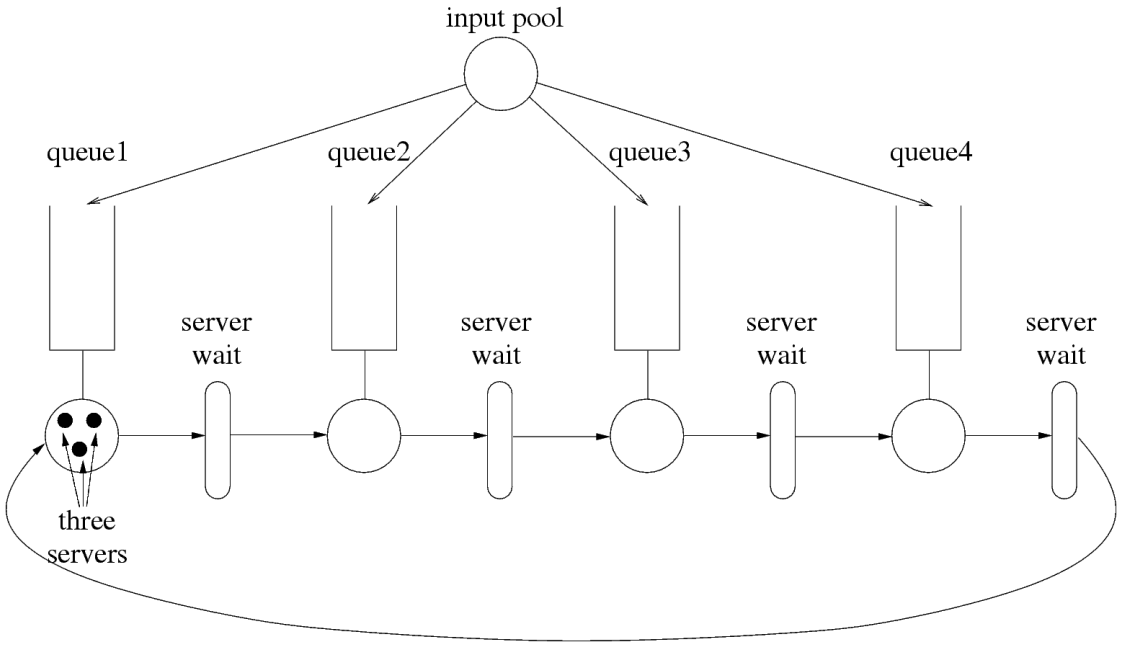



Figure 5.4: MSMQ subsystem

also how each of the 3 servers moves from one queue to the next after a waiting time that is an exponentially distributed random variable with a constant rate. Upon entering a queue, a server polls the queue. If there is no job waiting for service, it goes to the next queue after some waiting time. Otherwise, it gives service to one job in the queue and waits to be transferred to the next one. The MSMQ subsystem distributes the jobs from its input pool to each of the 4 queues with equal probability. After being served, each job is transferred to the hypercube subsystem's input pool. More information on MSMQ systems can be found in [68].

The second subsystem is the hypercube model (Figure 3.9) described in Section 3.4, except that in Section 3.4, we superimposed the input and output pools of the system, but we keep them separate here. The complete SAN specification of the model is given in Section A.1. To attain maximum symmetry, we set all the service rates and all of the repair rates of the processor to the same value.

We used the Möbius tool [38, 48] to specify the model of each of the subsystems using the SAN formalism. Then, we composed the models by sharing their input and output pools

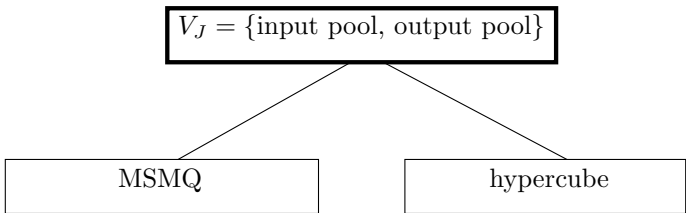



Figure 5.5: Composed model structure of the tandem multiprocessor model

via the Join operator in the Rep/Join composed editor (Figure 5.5). Our implementation of the symbolic SSG automatically partitions the set of places of the complete model and assigns each class of the partition to one level of the MD as follows:

- level 1: shared state variables (places in the SAN formalism) of the two submodels, i.e., input and output pools
- level 2: state variables of the hypercube submodel minus those in level 1
- level 3: state variables of the MSMQ submodel minus those in level 1.

### 5.4.3 Performance Results

Tables 5.1(a) and 5.1(b) show information about the MD representation of the original (i.e., unlumped) and lumped CTMC of the tandem multiprocessor system for different values of $J$, respectively. More specifically, Table 5.1(a) shows the state-space size for each level and for the complete model, the number of MD nodes in each level, the symbolic state-space generation time, and the memory requirement for the original MD. Table 5.1(b) shows the lumped state-space size for each level and for the complete model, the state-space reduction we gain from the compositional lumping algorithm for levels 2 and 3 and for the complete model, the compositional lumping algorithm running time, and the memory requirement for the lumped MD. Table 5.1(b) does not show the number of MD nodes in each level because the number of nodes in each level of the lumped and unlumped MD is the same; the reason is that the compositional lumping algorithm replaces each MD node with a possibly smaller

(a) Unlumped MD

| $J$ | unlumped SS sizes | | | | # of MD nodes | | | generation time (s) | MD memory usage |
|---|---|---|---|---|---|---|---|---|---|
| | overall | $\|\mathcal{S}_1\|$ | $\|\mathcal{S}_2\|$ | $\|\mathcal{S}_3\|$ | $\|N_1\|$ | $\|N_2\|$ | $\|N_3\|$ | | |
| 1 | 22100 | 2 | 650 | 160 | 1 | 3 | 3 | 0.05 | 53.9KB |
| 2 | 197600 | 3 | 3575 | 700 | 1 | 5 | 4 | 0.8 | 421KB |
| 3 | 1236300 | 4 | 14300 | 2220 | 1 | 7 | 5 | 12.1 | 2230KB |

(b) Lumped MD

| $J$ | lumped SS sizes | | | | reduction in SS | | | lumping time (s) | MD memory usage |
|---|---|---|---|---|---|---|---|---|---|
| | overall | $\|\widetilde{\mathcal{S}}_1\|$ | $\|\widetilde{\mathcal{S}}_2\|$ | $\|\widetilde{\mathcal{S}}_3\|$ | overall | $l_2$ | $l_3$ | | |
| 1 | 395 | 2 | 30 | 40 | 55.9 | 21.7 | 4 | 0.04 | 4.7KB |
| 2 | 4075 | 3 | 178 | 175 | 48.4 | 20.4 | 4 | 0.26 | 36KB |
| 3 | 28090 | 4 | 803 | 555 | 44 | 17.8 | 4 | 1.8 | 201KB |

Table 5.1: Specifications of MD representation of tandem system's CTMC

one and does not create or delete any node. In Table 5.1(b), $l_2$ and $l_3$ refer to levels 2 and 3 of the MD. The state-space reduction for level 1 is not shown because it is always 1.

We observe in Table 5.1(b) that the compositional lumping algorithm reduces the state-space size of the overall model to roughly 1/40 to 1/50 of its original value. The equivalently behaving sets of entities, that is, 1) the four queues of the MSMQ subsystem, 2) processors $A$ and $A'$ in the hypercube (Figure 3.9), and 3) the other 6 processors in the hypercube, are the source of the lumpability found by our compositional lumping algorithm.

As mentioned in Section 5.3, our algorithm does not necessarily generate the smallest possible lumped CTMC (or its MD representation) because it is applied locally at each level of the MD and does not have a global view of the CTMC represented by the MD. In other words, the resulting lumped CTMC could possibly be lumped to a smaller CTMC by a state-level lumping algorithm that has a flat (i.e., global) view of the CTMC. Nevertheless, we verified that our compositional algorithm generates the smallest lumped CTMC possible for the example considered. We did that by running the state-level lumping algorithm we developed in Chapter 3 on the CTMC that resulted from the compositional algorithm; the output was the same as the input CTMC.

Generally, the reduction in state-space size has two major effects on the efficiency of

iterative numerical solution algorithms that compute measures of CTMCs: it reduces both the space and time requirements for such algorithms. Reduction in the size of the state space affects space requirements in two ways. First, it makes the MD representation of the CTMC smaller. In our example, the memory requirement for the MD has been reduced by around an order of magnitude for all values of $J$. Second, and more important, it reduces the size of the solution vector, the infamous bottleneck of iterative numerical solution algorithms, proportionately to the overall state-space reduction; in our example, the vector was reduced to no more than 1/40 its original size. Therefore, the advantage of using our compositional lumping algorithm is that we can solve larger models than would be possible using only symbolic techniques; for our example, we solved models that are one to two orders of magnitude larger. The reduction in the size of the state space also results in a roughly proportionate reduction in the amount of time spent for each iteration of the numerical solution algorithm. It is important to realize that all the benefits in terms of time and space requirements described above are achieved through an efficient algorithm in an amount of time that is negligible compared to the time needed for numerical analysis and that is, for our example, considerably less than the time needed for state-space generation.

Note that the degree of reduction in state-space size due to compositional lumping obviously depends on the model and its compositional structure. In the worst case, none of the levels of the MD satisfy the lumpability conditions for any non-trivial partition (partitions with more than one class), so that our lumping algorithm cannot reduce the size of the state space. For example, suppose we decompose the model of the cube-connected processors into 8 submodels. Then, the corresponding MD will have 8 more levels. In that case, our algorithm will not find any non-trivial lumpable partition for any of the 8 levels, because the algorithm is applied at each level separately, and the symmetry is no longer found in any of the 8 levels. Nevertheless, our algorithm finds non-trivial lumpable partitions if a set of levels of the MD, whose corresponding submodel(s) together have more lumping potential, are grouped into one level. In fact, for our example, we grouped all equivalently behaving

processors in the same submodel, and therefore, we could gain significant reduction in the state spaces of levels 2 and 3 and the overall state space.

## 5.5 Summary

In this chapter, we presented a compositional algorithm that lumps an MRP represented as an MD (for the CTMC representation) that is augmented with reward and initial probability distribution vectors. The lumping is performed by reducing the nodes of each level separately from other levels. Recently, many authors have used MDs to represent very large matrices of Markovian models specified in a variety of modeling formalisms. Unlike previous compositional lumping algorithms that were formalism-dependent, our algorithm is applicable on any MD, and thus on any formalism that uses MDs or Kronecker representation[2] for its CTMC representation.

Specifically, we defined two equivalence relations (one for ordinary and one for exact lumping) on the state space of a given level of an MD that is augmented to a Markov reward process (MRP) by assignment of rewards and initial probabilities to states. We then proved that reducing the nodes of that level with respect to each relation yields an MD representing a lumped CTMC. We also designed and implemented an algorithm that computes the equivalence relations for each level and reduces a given MD with respect to those relations. Finally, we showed by an experimental study that the algorithm can in fact take advantage of the existing lumpability in each level and reduce the size of the MD by roughly one to two orders of magnitude for our example model. The state-space reduction makes iterative numerical solution algorithms require roughly one to two orders of magnitude less space (mostly taken by the solution vector) and less time per iteration. The important point to note is that the state-space reduction is achieved in an amount of time that is much smaller than the amount of time needed for the generation or numerical solution of

[2]As we mentioned before, any matrix expression using Kronecker operators can be represented as an MD.

the unlumped MD.

# Chapter 6

# Conclusion

In this final chapter, we identify the main contributions of this dissertation and the extent of their implications, and explain the promising directions in which further ideas can be investigated.

## 6.1 High-level View

The goal of this dissertation is to extend, improve and combine existing solutions of the state-space explosion problem and develop new algorithms and tools to make possible the construction and solution of very large CTMCs generated from high-level Markovian models. Our new techniques attack the state space explosion problem on two fronts: largeness avoidance and largeness tolerance. On the former front, we focused on various types of lumping techniques and designed efficient algorithms that exploit lumping potential at different levels and generate a lumped CTMC that is potentially smaller than the original one. By numerically solving the lumped CTMC, we obtain exactly the same results that we would obtain from the original CTMC. Using lumping techniques, we could reduce both the space and time complexity of iterative numerical solution algorithms. On the latter front, we utilized MDD and MD data structures to very compactly represent large state spaces and state transition rate matrices of compositional models that were built through sharing of state variables.

Our most important contribution is that, with some minor restrictions, all the lumping and symbolic techniques that we have described throughout the dissertation were orthogo-

nal. More specifically, we can apply our compositional and model-level lumping techniques simultaneously on a compositional model formalism and always keep the state space and state transition rate matrix representations symbolic. The result is that we will be able to exploit two types of lumpings for a composed model: 1) lumping due to the symmetries present among the various components of the composed model induced by the replicate operator, and 2) lumping present in each individual component. That will enable us to generate potentially smaller MD representations of CTMCs than would be possible using either of the techniques individually. Moreover, the combination above is again orthogonal to our state-level lumping algorithm in the sense that any partially lumped CTMC that is computed by the model-level and compositional lumping algorithms can still be fed to the state-level algorithm to obtain the smallest possible lumped CTMC.

## 6.2 Contributions

In performability and availability modeling, computing high-level reward measures is the final goal of the analysis of any model, including Markovian models. That is the reason why we introduced MRPs in Chapter 2, which bundle the underlying CTMC of a model with a rate reward measure and an initial probability distribution vector. We defined the ordinary and exact lumpability of MRPs with respect to a partition and pointed out that a lumped MRP can be used to compute exactly the same reward measures that could be computed from the original MRP.

Then, we introduced in Chapter 3 our new state-level lumping algorithm, which, given an MRP, can compute the coarsest possible partition with respect to which the MRP is ordinarily or exactly lumpable. In other words, it generates the smallest possible lumped MRP from which the desired reward can be computed exactly. We proved that the running time of our algorithm was $\mathcal{O}(m \lg n)$, where $m$ is the number of nonzero entries of the generator matrix and $n$ is the number of states, whereas the previously fastest algorithm for

the problem had a running time of $\mathcal{O}(mn)$. We also proved that, prior to our new algorithm, we could not achieve a running time better than $\mathcal{O}(m \lg^2 n)$, even using the state-of-the-art techniques and ideas in the literature. The key to the running time improvement was an elegant use of the splay tree data structure as subclass trees to perform the split operation. That made our algorithm the fastest known to date.

The implications of the algorithm are far-reaching. Although we discussed only optimal lumping of MRPs (that are based on CTMCs), the algorithm can straightforwardly be extended to computation of the coarsest lumping in DTMCs (discrete-time Markov chains), the coarsest bisimulation in probabilistic models such as WSCCS (Weighted Synchronous Calculus of Communicating Systems) and PCCS (Probabilistic CCS) and other Markovian models such as PEPA (Performance Evaluation Process Algebra) and TIPP (Timed Processes and Performance Evaluation), and the coarsest bisimilarity on general weighted automata.

We implemented two variants of our state-level lumping algorithm; one used splay trees and the other used red-black trees to represent subclass trees. Our experiments showed that, in practice and for virtually all cases, the red-black variant is less than 10% faster than the splay tree variant, although the latter is theoretically faster. We explained that observation with the very small size of the subclass trees and the relatively high constant factor of splay tree operations. We also showed that both variants are always faster than the implementation of the fastest previously known algorithm by up to three orders of magnitude, depending on the size of the model.

The limitation of any state-level lumping algorithm, including ours, is that it views the MRP at the level of CTMC states. In other words, it does not exploit any compositional properties of the model that generated that MRP. It also requires the original MRP to be constructed in some representation. For very large MRPs, the space and, more importantly, time requirements become prohibitively large. Hence, we needed other lumping techniques that directly generate a lumped MRP (but not necessarily the smallest possible one) without first generating the original MRP. That motivated the work of Chapters 4 and 5. On the

other hand, the strength of the state-level lumping over all other lumping techniques is that it always generates the smallest possible lumped MRP. Therefore, the best time to use it is after we have exhausted all other lumping techniques. At that point, the size of the partially lumped MRP is possibly much smaller than the original MRP, and we can apply the state-level lumping algorithm to get the fully lumped MRP.

In Chapter 4, we explained a new approach that combines two already existing techniques, namely, the symbolic representation of state transition rate matrices (largeness tolerance) and model-level lumping (largeness avoidance). The first technique decreased the space requirements of numerical solution algorithms by using matrix diagrams and multi-valued decision diagrams to represent state transition rate matrices and state spaces, respectively. Hence, it enabled us to assign most of the available memory to iteration vector(s) that are so far not amenable to compact representation. Therefore, we could analyze models that were one to two orders of magnitude larger than models that were solvable without symbolic representation. The technique had been previously utilized for compositional models that were based on action synchronization. We extended it for state-sharing compositional models that were built using join and replicate operators.

The second technique exploits the symmetry induced by the replicate operator and directly generates a lumped MRP. That symmetry exists due to the equivalently behaving components of the model. The technique has been known for a long time; however, it could be applied only when the state transition rate matrix was represented using a sparse matrix representation, placing an upper limit on the size of the CTMC that could be analyzed. We overcame the challenge by developing 1) a very efficient symbolic algorithm that generated the state space of the lumped MRP from the state space of the original MRP, both of which were represented as MDDs, and 2) a new data structure, which we call the "sorting MDD," that efficiently maps a vector representation of a (non-representative) state of the original CTMC to the index of the corresponding representative state.

We implemented the algorithm mentioned above and integrated it into the Möbius mod-

eling tool. The implementation automatically exploited the symmetry and generated all the necessary symbolic data structures used for the enumeration of the lumped CTMC, including the MD representation of the lumped CTMC and the sorting MDD. Our implementation extended the size of CTMCs that were analyzable by the Möbius tool by one to two orders of magnitude depending on the model.

The last piece of work, which we presented in Chapter 5, dealt with the exploitation of lumping potential *inside* each component of a model, as opposed to lumping that exists because of the symmetry *among* components. More specifically, we designed a compositional algorithm that lumped an MRP represented as an MD (for the CTMC representation), which was augmented with reward and initial probability distribution vectors. The lumping was performed by reducing the nodes of each level separately from other levels. The implication of the work is apparent from the observation that recently many authors have used MDs to represent very large matrices of Markovian models specified in a variety of modeling formalisms. Unlike previous compositional lumping algorithms that were formalism-dependent, our algorithm is applicable on any MD, and thus on any formalism that uses MDs or Kronecker representation for its CTMC representation.

Our lumping algorithm is compositional, which means that it performs the state-level lumping algorithm on a number of often drastically smaller matrices (MD nodes) than the original state transition rate matrix. That reduces both the space and time requirements of the lumping algorithm, at the cost of not being able to compute the optimal lumping, which can be compensated for with a single application of the state-level lumping algorithm on the resulting partially lumped MRP. Our experiments showed that the reduction can be gained in an amount of time that is much smaller than the amount of time needed for the generation or numerical solution of the unlumped MD.

Our compositional lumping algorithm was made possible through the definition of two equivalence relations (one for ordinary and one for exact lumping) on the state space of a given level of an MD. We then proved that reducing the nodes of that level with respect to

each relation yields an MD representing a lumped CTMC.

We also implemented the compositional lumping algorithm and integrated it with our symbolic SSG implementation from Chapter 4. The final implementation could in fact exploit lumping both at the level of each component and also among the equivalently behaving components specified by the replicate operator. Moreover, it represented the lumped MRP using symbolic data structures.

Taken as a whole, the combination of the state-level, model-level, and compositional lumping algorithms and the symbolic data structures that we have developed in this dissertation suggests that they can be effective on a large class of Markov models used in performance, dependability, and performability analysis.

## 6.3 Future Work

There are very specific, and in some cases challenging, ways to extend the work that we have discussed in this dissertation. We explain them in this section in the order in which their related work was presented.

In Chapter 3, we designed a state-level lumping algorithm with $\mathcal{O}(m \log n)$ time complexity, which is the fastest known algorithm. We also proved a lower bound of $\mathcal{O}(m + n \lg n)$ on the running time of any state-level lumping algorithm. We can see that there is a noticeable gap between the two time complexities. It means that we do not know whether there is any algorithm faster than ours or whether our algorithm is the fastest possible. It is an open question, and a difficult one, to figure out how hard the state-level lumping problem is.

In Chapter 4, we designed the "sorting MDD," a data structure that, at least for our example, improved by an order of magnitude the time complexity of the enumeration of the elements of the lumped state transition rate matrix. The sorting MDD computes $\rho(min(s'))$, that is, the representative state index corresponding to a given (non-representative) state. Its shortcoming was that, at least for our example model, it was fairly large compared to

other symbolic representations. It would be worthwhile to investigate other combinations of data structures and algorithms that can do the same computation with improved, or at least different, trade-offs between time and space requirements.

As with any technique that uses symbolic data structures like MDDs and MDs, the order of the levels, and in our case the partitioning of the state variables of a model into levels, can considerably affect the space complexity of the data structure and the time complexity of algorithms that are applied on it. In general, it is an NP-hard problem to compute the order (of the levels) that results in the smallest MDD or MD representation. However, there is a possibility that there are heuristic or exact rules that use the compositional properties of the model to improve the ordering of the levels.

# Appendix A

# Model Specifications

We expect that interested researchers who read this dissertation may like to reproduce some of the performance studies described at the end of Chapters 3, 4, and 5 for two reasons: 1) to gain a better understanding of the techniques by performing further experiments, and 2) to be able to extend upon the work presented in the dissertation. For that reason, we provide below the complete specifications of all the models that we used in those chapters.

## A.1 Hypercube Multi-processor Model

In Chapter 3, we used the model of a hypercube multiprocessor to study the performance of the two variants of our state-level lumping algorithm and Buchholz's algorithm [19]. The model has been specified in the SAN (Stochastic Activity Network) [84] formalism using the Möbius modeling tool [38, 48]. The model consists of a single atomic model[1], so we do not need a composed model in order to specify the complete model.

Figure A.1 illustrates the graphical representation of the SAN model generated by the documentation feature of Möbius. As in all SAN atomic models, (blue[2]) circles represent places, thick (blue) rounded rectangles represent timed activities (the SAN equivalent of timed actions), thin (blue) bars represent instantaneous transitions, and finally, (red) triangles represent input gates, which are SAN-specific elements that specify additional enabling conditions for timed or instantaneous activities. Notice how the input and the output pools

[1] In the Möbius tool terminology, an atomic model is a basic model that the modeler decides not to decompose into smaller models.

[2] It can be proved that you have either a color monitor or a color printer!

are superimposed and modeled by a single place called `input_output_model`. Processors 1 and 8 in Figure A.1 correspond to processors $A$ and $A'$ in Figure 3.9(a). All activities in the model are either instantaneous or timed with an exponentially distributed firing time.

Table A.1 shows the list of the global variables (Möbius's term for parameters) defined for the model. Table A.2 gives the list of places of the model and their initial markings. Table A.3 shows the list of exponentially distributed activities of the model, their rates, and their roles in the model. All the timed activities have empty activation and reactivation predicates. Table A.4 lists all the instantaneous activities and their meanings. Finally, Table A.5 shows the list of input gates and their predicates and functions. In all the tables related to the hypercube models, $i$ and $j$ represent processor indices ($1 \leq i, j \leq 8$) such that processors $i$ and $j$ are neighbors.

## A.2 Courier Protocol Model

The Courier protocol model is a GSPN model of a parallel communication software system that first appeared in [93]. As we mentioned in Chapter 4, we have adapted it from [93] by 1) specifying it in the SAN formalism so that it is usable in the Möbius tool and 2) converting all actions to timed ones to retain a significant number of actions. The description of the model and the role of each place and activity can be found in [93]. In this section, we will only give the SAN representation of the 4 atomic models into which we have decomposed the overall model and also the graphical representation of the composed model.

Figure A.2 illustrates the atomic models of the session layers of both the sender and the receiver. Figure A.3 shows the atomic models of the transport layer of both the sender and the receiver. Finally, Figure 4.6 shows how the atomic models are combined into a composed model by sharing of state variables (or places for SAN atomic models).

Table A.6 shows the list of places of the atomic models and their initial markings. `N`, `TBS`, and `TWS` are global variables defined for the model. For all our experiments, we have

TBS= 1 and N=TWS. See [93] for the meanings of these global variables.

## A.3 MSMQ Polling-based Queuing System

The MSMQ (Multi-Server Multi-Queue) polling-based queuing system that we used in Chapter 5 is one that is derived from [68]. In our case, it has 3 identical servers and 4 identical queues, but the model can be easily extended to have an arbitrary number of servers or queues. Figure A.4 shows the graphical SAN (Stochastic Activity Network) representation of the model. The queues are arranged in a cyclic manner and each of the 3 servers move from one queue to the next after a waiting time that is an exponentially distributed random variable with a constant rate. Upon entering a queue, a server polls the queue. If there is no job waiting for service, it goes to the next queue after some waiting time. Otherwise, it gives service to one job in the queue and waits to be transferred to the next one. The MSMQ subsystem distributes the jobs from its input pool to each of the 4 queues with equal probability.

Tables A.7, A.8, A.9, A.10, and A.11 show the list of global variables (parameters), places, timed activities, instantaneous activities, and input gates of the MSMQ model, respectively. In all tables, $1 \leq i \leq 4$ is the index of one of the four queues.

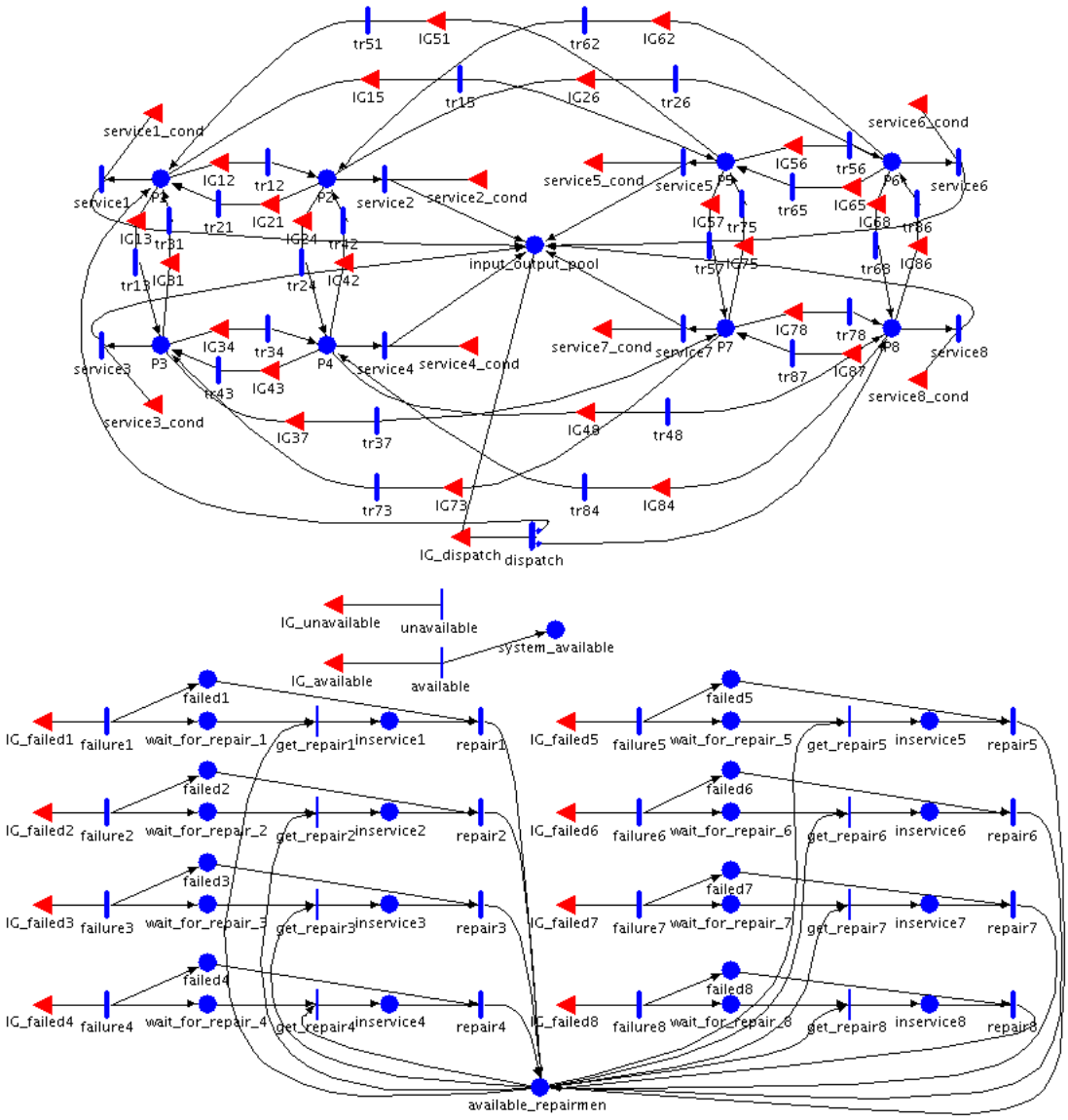



Figure A.1: The model of the hypercube multiprocessor system

| global variable name | description | comments |
|---|---|---|
| `dispatch_rate` | The rate at which jobs are transferred from `input_output_pool` to processors 1 and 8 | Does not affect the degree of lumpability |
| `failure_rate` | The rate of (independent) failure of any processor | Does not affect the degree of lumpability |
| `initial_repairmen` | The number of available repair facilities | Set to 1 for all experiments |
| $J$ | The number of jobs in the system | |
| `proc_capacity` | The capacity of queues of processors 1 and 8 | |
| `repair_rate`$i$ | The rate at which a repair facility repairs failed processor $i$ | Different repair rates for different processors decrease the degree of lumpability |
| `service_rate`$i$ | The rate at which processor $i$ finishes the process of a job in its queue | Different service rates for different processors decrease the degree of lumpability |
| `transfer_diff` | The threshold of the difference between the numbers of jobs in neighboring processors' queues for the load-balancing scheme to trigger between the two processors | Set to 1 for all experiments |
| `transfer_rate` | The rate of transferring a job from one processor's queue to a neighboring processor's queue due to load-balancing | |

Table A.1: List of global variables of the hypercube model and their meanings

| place name | initial marking | description |
|---|---|---|
| available_repairmen | initial_repairmen | The number of repair facilities not busy with repairing a processor |
| failed$i$ | 0 | 1 if processor $i$ is down and 0 otherwise |
| input_output_pool | $J$ | The common input and output pool of jobs |
| inservice$i$ | 0 | 1 if processor $i$ is being repaired and 0 otherwise |
| P$i$ | 0 | The number of jobs in processor $i$'s queue |
| system_available | 1 | 1 if the system is considered available and 0 otherwise |
| wait_for_repair$i$ | 0 | 1 if processor $i$ is failed and is waiting for a repair facility to become available |

Table A.2: List of places of the hypercube model, their initial markings, and their meanings

| activity name | rate | description |
|---|---|---|
| dispatch | dispatch_rate | Gets a job from the input/output pool and dispatches it to processor $k \in \{1, 8\}$ with probability $\frac{\texttt{proc\_capacity}-\texttt{P}k}{2\times\texttt{proc\_capacity}-\texttt{P1}-\texttt{P8}}$ |
| failure$i$ | failure_rate | Causes processor $i$ to fail |
| repair$i$ | repair_rate$i$ | Repairs processor $i$ |
| service$i$ | service_rate$i$ | One job from processor $i$'s queue gets service |
| tr$ij$ | transfer_rate | One job is transferred from processor $i$'s queue to processor $j$'s queue for the purpose of load-balancing |

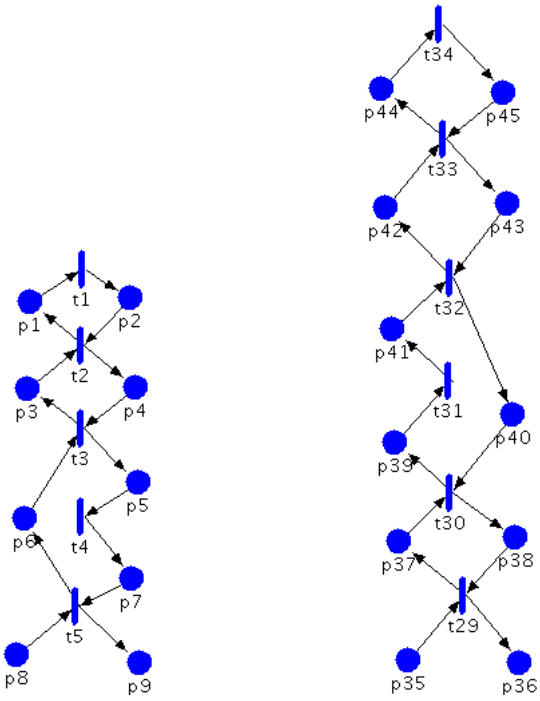
Table A.3: List of timed activities of the hypercube model, their rates, and their meanings

| activity name | description |
|---|---|
| available | Sets system_available (from 0) to 1 if the system is considered available |
| get_repair$i$ | Fires when a repair facility becomes available and chooses failed processor $i$ to repair |
| unavailable | Sets system_available (from 1) to 0 if the system is considered unavailable |

Table A.4: List of instantaneous activities of the hypercube model and their meanings

| **input gate name** | **predicate** | **function** |
|---|---|---|
| IG$ij$ | (P$i$ - P$j$ > transfer_diff) or (failed$i$ = 1 and P$i$ > 0) | P$i$ := P$i$ − 1 |
| IG_available | if system_available = 1<br>  return false<br>int failed_servers = 0<br>for $i$ = 1 to 8<br>  if failed$i$ = 1 and inservice$i$ = 1<br>  failed_servers++<br>if failed_servers >= 2<br>  return false<br>else return true | empty |
| IG_dispatch | input_output_pool > 0 and (P1 < proc_capacity or P8 < proc_capacity) | input_output_pool := input_output_pool −1 |
| IG_failed$i$ | failed$i$ = 0 and system_available= 1 | empty |
| IG_unavailable | if system_available = 0<br>  return false<br>int failed_servers = 0<br>for $i$ = 1 to 8<br>  if failed$i$ = 1 and inservice$i$ = 1<br>  failed_servers++<br>if failed_servers >= 2<br>  return true<br>else return false | empty |
| service$i$_cond | failed$i$ = 0 and system_available= 1 | empty |

Table A.5: List of input gates of the hypercube model, their predicates, and their functions

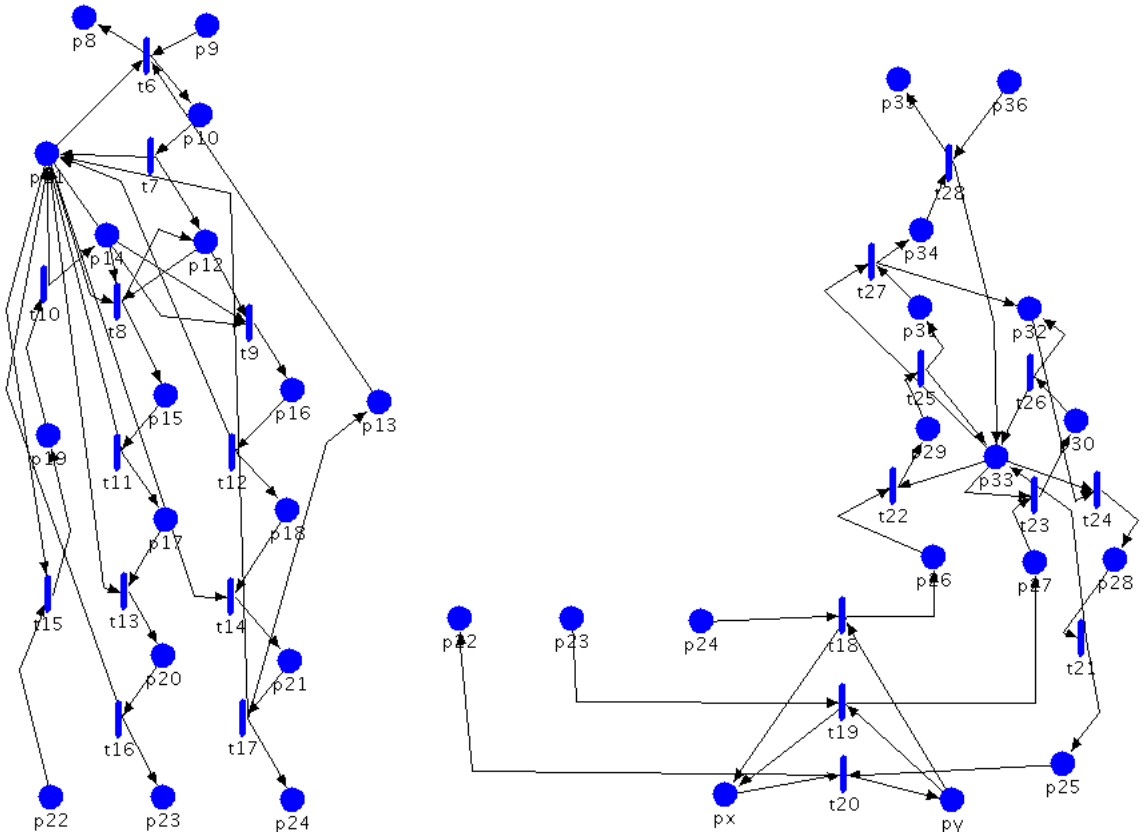



(a) Sender's session layer (b) Receiver's session layer

Figure A.2: Atomic models of the session layers of the Courier protocol model

| activity name | initial marking |
|---|---|
| P2, P4, P5, P7, P9, P10, P12, P15 through P32, P34, P35, P37, P39, P41, P42, P44, and Px | 0 |
| P1, P3, P6, P8, P11, P33, P36, P38, P40, P43, and P45 | 1 |
| P13 | TBS |
| P14 | TWS |
| Py | N |

Table A.6: List of places of the Courier protocol model and their initial markings




(a) Sender's transport layer (b) Receiver's transport layer

Figure A.3: Atomic models of the transport layers of the Courier protocol model

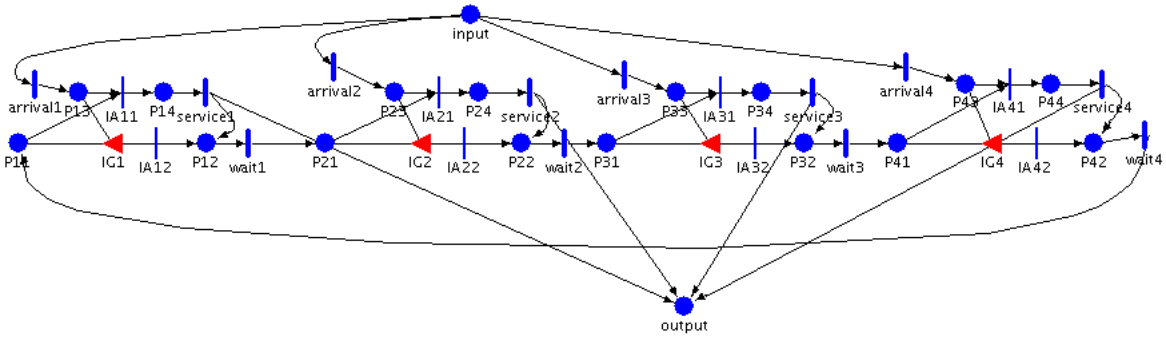



Figure A.4: SAN representation of the MSMQ model

| global variable name | description | comments |
|---|---|---|
| arrival_rate | The rate at which the job arrives from the input pool to each of the queues | Equal value for all queues |
| $J$ | The number of jobs in the system | |
| no_of_servers | The number of servers | Set to 3 for all experiments |
| service_rate | The rate at which a server processes a job in a queue | Value is independent of the job, the server, or the queue |
| wait_rate | The inverse of the expected amount of time that a server waits before polling the next queue | Equal for any consecutive queues |

Table A.7: List of global variables of the MSMQ model and their meanings

| place name | initial marking | description |
|---|---|---|
| input | $J$ | The number of jobs in the input pool |
| P$i$1 | 0 | The number of servers available to queue $i$ |
| P$i$2 | P12 = no_of_servers, P$i$2 = 0 for $i \neq 1$ | The number of servers waiting to poll the next queue |
| P$i$3 | 0 | The number of jobs waiting for servers |
| P$i$4 | 0 | The number of jobs that have acquired servers and waiting for the service to be finished |
| output | 0 | The number of jobs in the output pool |

Table A.8: List of places of the MSMQ model, their initial markings, and their meanings

| activity name | rate | description |
|---|---|---|
| arrival$i$ | arrival_rate | A job arrives from the input pool to queue $i$ |
| service$i$ | service_rate | One server completes the service of a job |
| wait$i$ | wait_rate | A server waits in queue $i$ before going to queue $i+1$ (or 1 if $i = 4$) |

Table A.9: List of timed activities of the MSMQ model, their rates, and their meanings

| activity name | description |
|---|---|
| IA$i$1 | A waiting job acquires a server in queue $i$ |
| IA$i$2 | A server proceeds to wait to move to the queue $i+1$ because there is no job in queue $i$ waiting for server |

Table A.10: List of instantaneous activities of the MSMQ model and their meanings

| input gate name | predicate | function |
|---|---|---|
| IG$i$ | P$i$1 $>$ 0 and P$i$3 $==$ 0 | P$i$1 := P$i$1 - 1 |

Table A.11: List of input gates of the MSMQ model, their predicates, and their functions

# References


[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing, 1974.

[2] M. Ajmone Marsan, G. Balbo, and G. Conte. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 2:93–122, 1984.

[3] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In *Proc. of Int. Conf. on Computer Aided Verification (CAV)*, volume 1102 of *LNCS*, pages 269–276, 1996.

[4] C. Baier, B. R. Haverkort, J.-P. Katoen, and H. Hermanns. On the logical specification of performability properties. In *International Colloquium on Automata, Languages and Programming*, volume 1853 of *LNCS*, pages 780–792, 2000.

[5] C. Baier and M. Z. Kwiatkowska. Domain equations for probabilistic processes. *Mathematical Structures in Computer Science*, 10(6):665–717, 2000.

[6] F. Bause, P. Buchholz, and P. Kemper. A toolbox for functional and quantitative analysis of DEDS. In *Proc. 10th Int. Conf. Modelling Techniques and Tools for Computer Performance Evaluation*, volume 1469 of *LNCS*, pages 356–359. Springer, 1998.

[7] A. Benoit, L. Brenner, P. Fernandes, and B. Plateau. Aggregation of stochastic automata networks with replicas. In *Proc. 4th Int. Conf. on Numerical Solution of Markov Chains (NSMC '03)*, pages 145–166, September 2003.

[8] A. Benoit, L. Brenner, P. Fernandes, and B. Plateau. Aggregation of stochastic automata networks with replicas. In *Proc. 4th Int. Conference on Numerical Solution of Markov Chains (NSMC '03)*, pages 145–166, September 2003.

[9] M. Bernardo and R. Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202:1–54, 1998.

[10] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, Aug. 1986.

[11] P. Buchholz. Numerical solution methods based on structured descriptions of Markovian models. In *Computer Performance Evaluation*, pages 251–267. Elsevier Science Publishers B.V. (North-Holland), 1991.

[12] P. Buchholz. Exact and ordinary lumpability in finite Markov chains. *Journal of Applied Probability*, 31:59–74, 1994.

[13] P. Buchholz. Markovian process algebra: Composition and equivalence. In *Proc. 2nd Workshop on Process Algebras and Performance Modelling*, volume 27(4) of *Arbeitsberichte des IMMD*, pages 11–30, 1994.

[14] P. Buchholz. Equivalence relations for stochastic automata networks. In W. J. Stewart, editor, *Computation with Markov Chains*, pages 197–216. Kluwer Int. Publishers, 1995.

[15] P. Buchholz. Hierarchical Markovian models: Symmetries and reduction. *Performance Evaluation*, 22(1):93–110, February 1995.

[16] P. Buchholz. Lumpability and nearly-lumpability in hierarchical queueing networks. In *Proc. IEEE Int. Computer Performance and Dependability Symposium (IPDS'95)*, pages 82–91. IEEE CS Press, 1995.

[17] P. Buchholz. *A framework for the hierarchical analysis of discrete event dynamic systems (Habilitations thesis).* Universität Dortmund, February 1996.

[18] P. Buchholz. Exact performance equivalence: An equivalence relation for stochastic automata. *Theoretical Computer Science*, 215(1/2):263–287, 1999.

[19] P. Buchholz. Efficient computation of equivalent and reduced representations for stochastic automata. *International Journal of Computer Systems Science & Engineering*, 15(2):93–103, 2000.

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

[21] P. Buchholz and P. Kemper. Numerical analysis of stochastic marked graphs. In *Proc. 6th Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, pages 32–41, October 1995.

[22] P. Buchholz and P. Kemper. Efficient computation and representation of large reachability sets for composed automata. *Discrete Event Dynamic Systems: Theory and Applications*, 12(3):265–286, 2002.

[23] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[24] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed colored nets and symmetric modeling applications. *IEEE Trans. on Computers*, 42(11):1343–1360, November 1993.

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

[26] G. Ciardo, R. M. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Proc. of TACAS 2003*, volume 2619 of *LNCS*, pages 379–393. Springer, 2003.

[27] G. Ciardo and A. S. Miner. Storage alternatives for large structured state spaces. In *Proc. 9th Int. Conf. Modelling Techniques and Tools for Computer Performance Evaluation*, volume 1245 of *LNCS*, pages 44–57. Springer, 1997.

[28] G. Ciardo and A. S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In *Proc. 8th Int. Workshop Petri Nets and Performance Models*, pages 22–31, 1999.

[29] G. Ciardo and A. S. Miner. Efficient reachability set generation and storage using decision diagrams. In *Proc. 20th Int. Conf. Application and Theory of Petri Nets*, volume 1639 of *LNCS*, pages 6–25. Springer, 1999.

[30] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[32] P. J. Courtois. *Decomposability*. Academic Press, New York, 1977.

[33] P.-J. Courtois and P. Semal. Computable bounds for conditional steady-state probabilities in large Markov chains and queueing models. *IEEE Journal on Selected Areas in Communications*, SAC-4(6):926–937, September 1986.

[34] D. Daly, P. Buchholz, and W. H. Sanders. An approach for bounding reward measures in Markov models using aggregation. Submitted for publication.

[35] P.R. D'Argenio, H. Hermanns, J-P. Katoen, and J. Klaren. MoDeST: A modelling language for stochastic timed systems. In *Proc. of Process Algebra and Probabilistic Methods (PAPM-PROBMIV '01)*, volume 2165 of *LNCS*, pages 87–102. Springer, 2001.

[36] E. de Souza e Silva and H. R. Gail. Calculating availability and performability measures of repairable computer systems. *Journal of the ACM*, 36:171–193, January 1989.

[37] E. de Souza e Silva and H. R. Gail. Calculating transient distributions of cumulative reward. In *Proceedings of SIGMETRICS/Performance-95*, pages 231–240, Ottawa, Canada, May 1995.

[38] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius framework and its implementation. *IEEE Trans. on Soft. Eng.*, 28(10):956–969, October 2002.

[39] D. D. Deavours and W. H. Sanders. An efficient disk-based tool for solving large Markov models. *Performance Evaluation*, 33:67–84, 1998.

[40] D. D. Deavours and W. H. Sanders. 'On-the-fly' solution techniques for stochastic Petri nets and extensions. *IEEE Trans. on Soft. Eng.*, 24(10):889–902, October 1998.

[41] C. Delamare, Y. Gardan, and P. Moreaux. Performance evaluation with asynchronously decomposable SWN: Implementation and case study. In *Proc. 10th Int. Workshop on Petri Nets and Performance Models (PNPM '03)*, pages 20–29, September 2003.

[42] S. Derisavi, H. Hermanns, and W. H. Sanders. Optimal state-space lumping in Markov chains. *Information Processing Letters*, 87(6):309–315, September 2003.

[43] S. Derisavi, P. Kemper, W. H. Sanders, and T. Courtney. The Möbius state-level abstract functional interface. In *Proc. of the 12th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS 2002)*, pages 31–50, 2002.

[44] S. Donatelli. Superposed stochastic automata: A class of stochastic Petri nets amenable to parallel solution. In *Proc. 4th Int. Workshop on Petri Nets and Performance Models (PNPM'91)*, pages 54–63, 1991.

[45] S. Donatelli. Superposed generalized stochastic Petri nets: Definition and efficient solution. In *Proc. 15th Int. Conf. on Applications and Theory of Petri Nets*, volume 815 of *LNCS*, pages 258–277. Springer-Verlag, 1994.

[46] J. C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Sci. Comput. Programming*, 13(2-3):219–236, 1990.

[47] S. Gilmore, J. Hillston, and M. Ribaudo. An efficient algorithm for aggregating PEPA models. *IEEE Transactions on Software Engineering*, 27(5):449–464, May 2001.

[48] PERFORM Performability Engineering Research Group. Möbius: Model-based environment for validation of system reliability, availability, security, and performance. Online Manual. `http://www.perform.csl.uiuc.edu/mobius/manual/MobiusManual.pdf`, University of Illinois at Urbana-Champaign, 2005.

[49] O. Gusak, T. Dayar, and J.-M. Fourneau. Lumpable continuous-time stochastic automata networks. *European Journal of Operational Research*, 148:436–451, 2003.

[50] B. R. Haverkort and K. S. Trivedi. Specification techniques for Markov reward models. *Discrete Event Dynamic Systems: Theory and Applications*, 3(2/3):219–247, July 1993.

[51] H. Hermanns. *Interactive Markov Chains and the Quest for Quantified Quality*, volume 2428 of *LNCS*. Springer, 2002.

[52] H. Hermanns and M. Rettelbach. Syntax, semantics, equivalences, and axioms for mtipp. In *Proc. 2nd Workshop on Process Algebras and Performance Modelling*, volume 27(4) of *Arbeitsberichte des IMMD*, pages 71–87, 1994.

[53] H. Hermanns and M. Ribaudo. Exploiting symmetries in stochastic process algebras. In *Proc. of 12th European Simulation Multiconference (ESM)*, pages 763–770, 1998.

[54] H. Hermanns and M. Siegle. Bisimulation algorithms for stochastic process algebras and their BDD-based implementation. In J.-P. Katoen, editor, *Proc. of 5th Int. AMAST Workshop on Real-Time and Probabilistic Systems (ARTS'99)*, volume 1601 of *LNCS*, pages 144–264. Springer, 1999.

[55] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[56] J. E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196, New York, 1971. Academic Press.

[57] R. A. Howard. *Dynamic Probabilistic Systems, Volume II: Semi-Markov and Decision Processes*. Wiley, New York, 1971.

[58] D. T. Huynh and L. Tian. On some equivalence relations for probabilistic processes. *Fundamenta Informaticae*, 17:211–234, 1992.

[59] T. Ibaraki and N. Katoh. On-line computation of transitive closures of graphs. *Information Processing Letters*, 16:95–97, 1983.

[60] W. D. Obal II. *Measure-Adaptive State-Space Construction Methods*. PhD thesis, University of Arizona, 1998.

[61] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 228–240, 1983.

[62] J. G. Kemeney and J. L. Snell. *Finite Markov Chains*. D. Van Nostrand Company, Inc., 1960.

[63] P. Kemper. Numerical analysis of superposed GSPNs. In *Proc. Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, pages 52–61, Durham, N. Carolina, USA, 1995.

[64] Y.-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proc. 29th Conf. on Design Automation*, pages 608–613, Los Alamitos, CA, USA, June 1992. IEEE CS Press.

[65] V. V. Lam, P. Buchholz, and W. Sanders. A structured path-based approach for computing transient rewards of large CTMCs. In *Proc. of 1st Int. Conf. on Quantitative Evaluation of Systems (QEST)*, pages 136–145, Enschede, The Netherlands, September 2004.

[66] K. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1991.

[67] D. Lee, J. Abraham, D. Rennels, and G. Gilley. A numerical technique for the evaluation of large, closed fault-tolerant systems. In *Proc. 2nd Int. Conference on Dependable Computing for Critical Applications 2 (DCCA-2)*, pages 95–114, 1992.

[68] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling With Generalized Stochastic Petri Nets*. John Wiley & Sons, 1995.

[69] J. F. Meyer, A. Movaghar, and W. H. Sanders. Stochastic activity networks: Structure, behavior, and application. In *Proc. International Workshop on Timed Petri Nets*, pages 106–115, Torino, Italy, July 1985.

[70] R. Milner. *Communication and Concurrency*. Prentice Hall, London, 1989.

[71] A. Miner and D. Parker. Symbolic representations and analysis of large probabilistic systems. In *Validation of Stochastic Systems: A Guide to Current Research*, volume 2925 of *LNCS*. Springer, 2004.

[72] A. S. Miner. *Data Structures for the Analysis of Large Structured Markov Models*. PhD thesis, The College of William & Mary, Virginia, 2000.

[73] A. S. Miner. Efficient solution of GSPNs using canonical matrix diagrams. In *Proc. PNPM'01: 9th International Workshop on Petri Nets and Performance Models*, pages 101–110, Aachen, Germany, September 2001.

[74] A. S. Miner. Computing response time distributions using stochastic Petri nets and matrix diagrams. In *Proc. 10th Int. Workshop on Petri Nets and Performance Models (PNPM)*, pages 10–19, 2003.

[75] D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library*. Addison-Wesley, 2001.

[76] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[77] B. Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. 1985 SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 147–153, 1985.

[78] B. Plateau and K. Atif. Stochastic automata network for modeling parallel systems. *IEEE Trans. on Soft. Eng.*, 17(10):1093–1108, Oct. 1991.

[79] M. A. Qureshi. *Construction and Solution of Markov Reward Models*. PhD thesis, University of Arizona, 1996.

[80] W. H. Sanders. *Construction and solution of performability models based on stochastic activity networks*. PhD thesis, The University of Michigan, 1988.

[81] W. H. Sanders and L. M. Malhis. Dependability evaluation using composed SAN-based reward models. *Journal of Parallel and Distributed Computing*, 15(3):238–254, July 1992.

[82] W. H. Sanders and J. F. Meyer. Reduced base model construction methods for stochastic activity networks. *IEEE Journal on Selected Areas in Communications*, 9(1):25–36, Jan. 1991.

[83] W. H. Sanders and J. F. Meyer. A unified approach for specifying measures of performance, dependability, and performability. In A. Avizienis, J. Kopetz, and J. Laprie, editors, *Dependable Computing for Critical Applications*, volume 4 of *Dependable Computing and Fault-Tolerant Systems*, pages 215–237. Heidelberg: Springer-Verlag, 1991.

[84] W. H. Sanders and J. F. Meyer. Stochastic activity networks: Formal definitions and concepts. In *Lectures on Formal Methods and Performance Analysis, First EEF/Euro Summer School on Trends in Computer Science*, volume 2090 of *LNCS*, pages 315–343, Berg en Dal, The Netherlands, July 2000. Springer.

[85] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[86] A. Srinivasan, T. Kam, S. Malik, and R. E. Brayton. Algorithms for discrete function manipulation. In *Proc. of the Int'l Conf. on CAD (ICCAD'90)*, pages 92–95, 1990.

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

[88] C. Tofts. A synchronous calculus of relative frequency. In *CONCUR '90: Proceedings on Theories of Concurrency: Unification and Extension*, volume 458 of *LNCS*, pages 467–480. Springer-Verlag Inc., 1990.

[89] C. Tofts. Processes with probabilities, priorities and time. *Formal Aspects of Computer Science (FACS)*, 6(5):536–564, 1994.

[90] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. John Wiley & Sons, New York, second edition, 2002.

[91] M. A. Weiss. *Data Structures and Algorithm Analysis in C++*. Pearson Education, second edition, 1994.

[92] E. W. Weisstein. Bell number. From Mathworld–A Wolfram Web Resource. `http://mathworld.wolfram.com/BellNumber.html`, 2005.

[93] C. M. Woodside and Y. Li. Performance Petri net analysis of communications protocol software by delay-equivalent aggregation. In *Proc. of the 4th Int. Workshop on Petri Nets and Performance Models*, pages 64–73, 1991.

# Author's Biography

Salem Derisavi was born in Kuwait on January 11, 1978. He moved to Iran in 1990 just before the first Gulf War. Salem received the Bachelors degree in computer engineering from Sharif University of Technology, Tehran, in 1999. He then moved to Champaign, Illinois, to pursue graduate study in stochastic modeling. He completed a Masters of Science degree in computer science from the University of Illinois in 2003. Following the completion of his Ph.D., Salem will begin work for the department of systems and computer engineering of Carleton University, Canada, as a postdoctoral fellow doing research on autonomic systems.