THE MÖBIUS STATE-LEVEL ABSTRACT FUNCTIONAL INTERFACE

BY

SALEM DERISAVI

B.S., Sharif University of Technology, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2003

Urbana, Illinois

Abstract

A key advantage of the Möbius modeling environment is the ease with which one can incorporate new modeling formalisms, model composition and connection methods, and model solution methods. We present a new state-level abstract functional interface (AFI) for Möbius that allows numerical solution methods to communicate with Möbius state-level models via the abstraction of a labeled transition system. This abstraction and its corresponding implementation yield an important separation of concerns: It is possible to treat separately the problem of representing large labeled transition systems, like generator matrices of continuous-time Markov chains, and the problem of analyzing these systems. For example, any numerical solver (e.g., Jacobi, SOR, or uniformization) that accesses a model through the Möbius state-level AFI can operate on a variety of state-space representations, including "on-the-fly," disk-based, sparse-matrix, Kronecker, and matrix-diagram representations, without requiring that the solver implementation be changed to match the state-space representation. This abstraction thus avoids redundant implementations of solvers and state-generation techniques, eases research cooperation, and simplifies comparisons of approaches as well as benchmarking. In addition to providing a formal definition of the Möbius state-level AFI, we illustrate its use on two state-space representations (a sparse matrix and a Kronecker representation) and on several numerical solvers for steady-state and transient analysis. With the help of this implementation and two example models, we demonstrate that the AFI provides the benefits of transparency while introducing only minor slowdowns in solution speed.

To my parents, who are symbols of perseverance and unconditional love.

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

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

Acknowledgments

I would like to thank my advisor, Professor William H. Sanders, for his valuable technical advice and support throughout my studies. I would also like to thank Professor Peter Kemper, from the University of Dortmund, for all of the valuable advice and input he gave me in the preparation of this thesis. A lot of thanks goes to Ms. Jenny Applequist for the time she spends correcting my endless writing mistakes, not only in this thesis but also in all my published works, and teaching me how to improve my writing skills.

I would also like to thank those members of the Möbius group whom I have worked with: Amy Christensen, who was always so willing to help; Graham Clark, the most skilled PhD soccer player I have ever seen; Tod Courtney, the solver of technical problems that everybody else was unable to deal with; David Daly, the epitome of organization; Dan Deavours, the senior member when I joined the group; Jay Doyle, the most smiley member; Vinh Lam, who was always more than willing to chat with me when I needed a break; and Patrick Webster, the smile generator of the group. I would also like to thank other people in the PERFORM group: Michel Cukier, Vishu Gupta, Kaustabh Joshi, Sudha Krishnamurthy, Ryan Lefever, James Lyons, Prashant Pandey, 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.

I would also like to thank all my relatives and friends who supported me throughout my school years. To name a few: Amir Behgooy, Hamid Reza Chitsaz, Yashar Ganjali, Bijan

Ghahreman, Zahra Golshani, Shirin Habib, Hossein Namazai, Ali Reza Namazifard, Roya Rezaii, Bardia Sadri, Hojjat Sharifi, and Reza Ziaei.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Model-based evaluation tools have been developed for many different modeling formalisms and use many different model solution techniques. Möbius [15, 16, 19, 20, 37] is a recent attempt to build a general multi-formalism multi-solution hierarchical modeling framework that permits the integration of a large number of modeling formalisms and model solution techniques. A key step in achieving this multi-paradigm approach is providing an appropriate interface between models expressed in different modeling formalisms, model composition and connection methods, and model solvers (e.g., simulators and state-space generators). We developed such an interface based on the notion of a model-level abstract functional interface (AFI) [15, 23]. The Möbius AFI provides an abstract notion of actions (events), state variables, and properties, and a common set of methods that permits heterogeneous models to interact with one another and with solvers without requiring them to understand details of the formalisms in which the constituent models are expressed.

There has also been a great deal of research in methods for dealing with the state-space explosion problem in state-based models by either avoiding or tolerating large state spaces. These methods have dramatically increased the size of models that can be analyzed. In "largeness avoidance" techniques, certain properties of some representation of the model (ranging from the high-level description of the model to the underlying continuous-time Markov chain (CTMC) itself) are exploited to reduce the size (in number of states and transitions) of the underlying CTMC that needs to be solved to obtain a solution of the model. Examples of this type of technique include detection of symmetries in models, and exploitation of lumping theorems. Approaches with that aim include Stochastic Well-formed Nets (SWNs) [12], stochastic activity networks (SANs) and replicate/join model composition [38], and stochastic process algebras [2, 5, 26, 28], among others. Other researchers have attempted to utilize “largeness tolerance” techniques, such as those that use variations of decision diagrams; such variations appear in the context of stochastic models as multi-terminal binary decision diagrams (MTBDDs) [1, 27, 29, 35], probabilistic decision diagrams [4, 11], and matrix diagrams [13]. These methods are based on the idea of sharing isomorphic substructures to save space and gain efficiency. Kronecker representations also allow representation of large transition rate matrices; different variants exist to reflect a modular [8] or hierarchical [6, 7] structure, or to allow matrix entries to be functions [24, 39]. In addition, on-the-fly generation [18] and disk-based methods [17, 33] make it possible to avoid the storage of a large state-level model by generating required matrix entries as needed, or by storing them on disk rather than in main memory, respectively.

All of the above approaches are interesting candidates for integration into Möbius, even though most of them were developed separately from one another and in the context of single modeling formalisms and/or model solution methods. Interestingly, although there is such a broad spectrum of avoidance and tolerance techniques, the techniques all place very similar requirements on the subsequent numerical model solution methods. In particular, the numerical model solution methods typically involve execution of a sequence of matrix-vector multiplications on some variant of the generator matrix of the resulting CTMC. Methods that require this include the Power method, the Jacobi method, and the Gauss-Seidel method for stationary analysis and uniformization for transient analysis. Clearly, numerical analysis is much richer in theory (for examples, see [3, 39]); however, the mathematical objects that are usually employed are matrix elements, rows, columns and submatrices, and simple algebraic operations thereon. Submatrices are used, for example, in iterative aggregation/disaggregation methods like KMS [34] or Takahashi’s [41] method. Those simple methods, which employ a homogeneous set of basic operations, thus appear most frequently in combination with the techniques discussed above.

The rich variety of techniques for dealing with the state-space explosion problem, and the fact that many numerical solution methods share similar basic operations, have motivated us to develop a state-level, as opposed to the existing model-level, AFI for Möbius [21, 22]. By doing so, we can separate state-space and state transition rate matrix generation and representation issues from issues related to the solution of the resulting state-level models. Creating a state-level AFI also allows us to create and implement numerical solution methods that do not require information about the data structures used to represent a state-level model. The key idea of this approach is to formulate a state-level AFI that allows numerical solution methods to see a model as a set of states and state transitions or, in other words, as a labeled transition system (LTS). The state-level interface we have created supports access to states and transitions in an efficient way via container and iterator classes. We are not the first ones to build an interface that allows one to iterate on an LTS; for example, in the field of protocol verification, the Caesar/Aldebaran tool [25] provides different iterators for this purpose that seem to rely on preprocessor expansion, and in his thesis [32], Knottenbelt gives an abstract C++ representation of states in the form of a non-template class. In contrast, we follow an object-oriented approach that uses templates similar to those used in the C++ standard template library (STL)[36]. Moreover, our proposed interface provides more general methods for accessing the LTS, and therefore enables us to support a larger number of numerical solution methods than previous works allow.

By creating a state-level AFI, we achieve more independence than is possible using the model-level Möbius AFI alone; this has advantages for both tool developers and tool users. In particular, our approach, when used together with the Möbius model-level AFI, avoids redundant reimplementations of the three steps (model specification, state-space and state transition rate matrix generation, and numerical analysis) taken when solving models numerically using state-based methods. That significantly reduces the effort that is necessary to implement, validate, and evaluate new approaches. Furthermore, it allows users to perform direct comparison of alternative approaches, without having to reimplement the work of other researchers; thus, they avoid the risk of being unfair when doing a comparison. Finally, it facilitates cooperation among researchers in developing new solution methods and combining existing ones; e.g., within Möbius, largeness-avoidance techniques based on lumpability can be combined with any state-based analysis method. In short, we achieve a situation in which research results that focus on model reduction, state-space exploration, LTS representation, or analysis can be developed independently but be used with one another. Obviously, tool users profit from this integrated approach, since more state-space generation and model solution methods become available to them. Likewise, we make the Möbius framework more useful to researchers who are creating techniques to avoid or tolerate large state spaces.

The remainder of this thesis is structured as follows: Chapter 2 specifies the requirements a state-level AFI must meet in order to be effective, and also presents, in detail, the state-level AFI that we developed, explaining the motivation behind the choices we made in developing it. In Chapter 3, we describe the implementation of two new state-level objects that use the AFI: an unstructured sparse-matrix representation as it is used in Möbius and a Kronecker representation that is employed in the APNN toolbox. Chapter 4 then analyzes two example models that are frequently considered in the literature for the performance of numerical analysis methods. We performed transient as well as steady-state analysis. Based on the performance results, we show that the overhead induced by the Möbius state-level AFI is small and is outweighed by the advantages it achieves. In Chapter 5, we conclude and mention potential research that can be done in continuation of this work.

Chapter 2

State-level AFI

For a state-level AFI to be effective in practice, it should satisfy a set of requirements. In this chapter, we first describe such requirements and then present what we believe is a state-level AFI that satisfies all those (rather conflicting) requirements in an appropriately balanced manner. Finally, we mention certain combinations of state-level representations and solution methods that our state-level AFI does not support.

2.1 State-level AFI Requirements

The transformation of a model from a high-level, user-oriented representation into a state-level model by a state transition graph generation is a transformation that may be technically complex, but it does not create additional information in the process. Instead, the goal of the transformation process is to create a representation that is as compact as possible, but can efficiently perform the operations needed during numerical solution. In order to design such a representation, we must study the type and amount of state and state transition information that algorithms access and the pattern of the accesses. In the following, we describe the characteristics that a state-level AFI should have and summarize how we have considered each one when designing the Möbius state-level AFI.

Functionality. A state-level AFI must have functionality sufficient to serve a large set of analysis techniques. More specifically, it must be easy to use and must include a sufficiently complete set of functions such that all of the analysis algorithms we are interested in can be written using this common interface. After studying a number of transient and steady-state solvers, we decided to include (among other things) function calls in the interface, so that we could access the elements of the rate matrix in a row-oriented, column-oriented, or arbitrary order. Matrices are sometimes partitioned into submatrices or blocks, so we also added functionality to operate on specific submatrices. More details are given in the next section.

Economy. The effort to support and implement the AFI for a particular state transition graph representation must be minimal, so as not to put an unnecessary burden on an AFI implementor. For state transition graph representations, it should be possible to support the required functionality in a natural, straightforward manner.

Clearly, economy and functionality are in conflict with one another, and a compromise must be reached. In our case, this means that we refrain from defining operations from linear algebra, such as matrix-vector multiplication, in the interface, since that could lead to an endless demand for further operations. We rather follow an approach in which a state-based analysis method reads information via the AFI, but does not transform it using the interface.

Generality. A state-level AFI should be "solution-method neutral," in the sense that it is not tailored toward any particular state transition graph representation or solution method. For example, many sophisticated algorithms rely on additional structural information. Kronecker methods are based on a compositional model description. Most variants of decision diagrams require an order on the variables, and heuristics on the order make use of information present in a model.

Flexibility. A state-level AFI must give implementors the opportunity to find creative optimizations in their implementations. For example, it should allow a developer who implements the interface for a particular state transition graph representation to exploit the special structure that may be present in the underlying state transition graph, in order to optimize the interface implementation. Ideally, all the optimizations that are possible in a traditional “monolithic” implementation should also be applicable to an implementation that uses the developed AFI. In Chapter 3, we will give an example of such possible state-space-specific optimizations when we describe the implementation of the interface for the Kronecker-based state transition graphs.

Performance. The performance of implementations using the interface must be competitive with the monolithic implementation. To achieve this we follow two design goals. First, we provide an AFI that is able to exploit the state-space-specific optimizations in the interface implementation. Second, we attempt to minimize the amount of overhead due to separation of the analysis algorithm and the state transition graph representation. The overhead is mostly caused by non-fully-optimized C++ compilation, extra function calls and assignments, and construction and deconstruction of temporary objects in the stack.

To summarize, we seek an interface that is straightforward to use and implement, is sufficient in functionality to support a wide variety of state transition graph representations and numerical solution methods, and provides good performance. A compromise among these goals is obviously necessary in any particular practical implementation of such an interface. We believe we have achieved an appropriate balance in our state-level AFI definition, which is described in the next section.

2.2 State-level AFI Definition

In this section, we begin by formalizing the notion of a labeled transition system by giving a definition that contains the key elements that specify a state-level, discrete-event system. Then, we briefly review several solution methods for CTMCs, which are a special case of discrete-event systems, to derive the basic operations that a state-level AFI needs to provide so that a wide range of solution methods can be implemented using the AFI. Finally, we show how containers and iterators help us achieve the separation of concerns discussed earlier, and show how our C++ realization of a state-level AFI satisfies the requirements described above.

In particular, with respect to those requirements, we address the flexibility and functionality of the Möbius state-level AFI in Sections 2.2.2 and 2.2.3, and its performance in Chapter 4. We illustrate the generality and economy of the AFI by implementing two conceptually different state-space representations in Chapter 3.

2.2.1 Labeled Transition System Definition

To define an appropriate state-level abstract functional interface for Möbius, we start by defining a labeled transition system (LTS). We define an $LTS = (S, s_0, \delta, L, \mathcal{R}, \mathcal{C})$, where:

- S is a set of states and $s_0 \in S$ is the initial state
- $\delta \subseteq S \times \mathbb{R} \times L \times S$ is the state transition relation, which describes possible transitions from a state $s \in S$ to a state $s' \in S$ with a label $l \in L$ and a real value $\lambda \in \mathbb{R}$
- $\mathcal{R} : S \times \mathbb{N} \rightarrow \mathbb{R}$ is the value of the i^{th} rate reward for each state in S
- $\mathcal{C} : \delta \times \mathbb{N} \rightarrow \mathbb{R}$ is the value of the i^{th} impulse reward for each transition in δ

The label l gives additional information concerning each transition, typically related to the event (in the higher-level model) that performs it. The real value λ can have several different meanings. In the following, it is taken to be the rate of the exponential distribution associated with the transition, because we are primarily interested in the numerical solution of the CTMC derived from a stochastic model. However, one can also consider probabilistic models, for which $\lambda \in [0, 1]$ gives the probability of a transition, or weighted automata, for which $\lambda > 0$ denotes a distance, a reward, or costs of a transition. By integrating both rates and labels in the definition of the LTS, we can use the interface based on it for both numerical solution and non-stochastic model checking. In the latter case, transition time is unimportant, and one may wish to consider the language that is generated by the transitions that may occur in the LTS. $\mathcal{R}$ and $\mathcal{C}$ are functions that define a set of rate and impulse rewards, respectively, for the LTS. They define what a modeler would like to know about the system being studied. Note that we could define δ as a function from $S \times S$ to $\mathbb{R} \times L$; that would be sufficiently descriptive for many models. However, since we wanted to be able to represent more general semantics (e.g., non-determinism) we chose to define δ as shown above.

Since we focus on Markov reward models in this paper, an *LTS* defines 1) a real-valued $(S \times S)$ rate matrix $R(i, j) = \sum_{e \in E} \lambda_e$, where $E \subset \delta$ is the set of transitions of the form $(i, \lambda_e, *, j)$,[1] and 2) a set of reward structures. The generator matrix Q of the associated CTMC is then defined as $Q = R - diag(rowsum(R))$, where the latter term describes a diagonal matrix with row sums of R as diagonal entries. The reward structures associated with the Markov model are determined by $\mathcal{R}$ and $\mathcal{C}$.

2.2.2 Use of Containers and Iterators

The philosophy we took when designing the state-level AFI and its implementation was inspired by the concept of "generic programming" [36] and the associated "containers" and "iterators" constructs in the STL (Standard Template Library) and generic class libraries. The idea was to decouple the implementations of a data structure and an algorithm operating on it, since the two are conceptually different. In other words, these concepts facilitate the implementation of algorithms that operate on data structures that are different and have different implementations, but support the same functionality. This decoupling is achieved through identification of a set of general requirements (called *concepts* in generic programming terminology and realized as member functions) met by a large family of abstract data structures. In our case, the "set of requirements" is a state-level AFI that provides the functionality necessary to implement a large class of solution methods efficiently. The requirements allow us to separate the numerical solution method that operates on a state-level model from the particular data structure that implements the model. This separation makes it easy to develop numerical solution methods and makes them applicable to any state-level

[1]The $*$ symbol means that the label of the transition can be arbitrary.

model that complies with the state-level AFI. Since the implementation of the AFI is separate from, and therefore does not interfere with, the analysis algorithm, we have the flexibility to optimize the internal implementation of the AFI for any particular state-level model (one of the characteristics mentioned in Section 2.1).

The two notions that help achieve this separation are those of "containers" and "iterators." *Containers* are classes (usually template classes) whose purpose is to contain other objects; *objects* are instantiations of classes. *Template* container classes are parameterized classes that can be instantiated so that they can contain objects of any type. A container is a programming abstraction of the notion of a mathematical set. By hiding the implementation of the algorithm for accessing the set elements inside the container class, we give developers both the ability to use a unified interface to access objects inside a container, and the flexibility to optimize the implementation of the container. In the Möbius state-level AFI, a container is used to represent a subset of transitions of an LTS. For example, the elements of a row or a column of a rate matrix constitute a row or column container object.

Iterators are the means by which the objects in a container are accessed. They can be considered "generalized" pointers, which allow a programmer to select particular objects to reference. The following operations are usually defined for iterator classes and implemented in the iterators that we define as part of the Möbius state-level AFI:

- *Navigation operators*, such as ++ and −−, which return iterators for (respectively) the next element and the previous element relative to the element pointed to by the iterator.
- *Dereferencing operators*, which enable us to access the object.
- *Comparison operators*, which define an order on the objects of the container.

2.2.3 State-Level AFI Classes

Before explaining how we use containers to represent sets of transitions, we review several common numerical solution methods to illustrate the access patterns they require from an LTS representation. These patterns will suggest the order in which the transitions of a state-level model should be placed in container objects. We then give a precise programming representation for the transitions contained in containers. This implementation, together with a set of methods returning information about the whole model (e.g., the number of states) along with a number of methods to facilitate computation of reward structures defined on the models, provides the complete state-level AFI we have developed.

Required Operations

We now briefly recall iteration schemes of some simple but frequently employed iterative solution methods, namely the Power, Jacobi, Gauss-Seidel, and Takahashi methods for stationary analysis, and uniformization for transient analysis. This review will help us determine both the type of container classes that a state-level object should provide to a solver, and also the general information the solver needs concerning a model. Table 2.1 summarizes the iteration schemes. More details can be found in, for example, [39].

Notably, all of the iteration schemes we describe, except Takahashi's method, are based on successive vector-matrix multiplications, where matrices P and Q are only minor transformations of the rate matrix R given by an *LTS* as mentioned above. For Takahashi's method, the given description does not specify how the aggregated equation system and the N non-homogeneous block equation systems, one for each block, are solved. If iterative solution methods are applied in those systems, Takahashi's method reveals a vector-matrix multiplication as an essential operation on submatrices, just as it did for matrices. Typical access patterns for matrix-vector multiplications are accesses by rows or by columns. However, a closer look reveals that only Gauss-Seidel requires a sequential computation of entries $\pi^{(k+1)}(i)$; Gauss-Seidel completes computation of $\pi^{(k+1)}(i)$ before continuing with $\pi^{(k+1)}(j)$

Method	Iteration scheme
Power	$\pi^{(k+1)} = \pi^{(k)}P$ where $P = (\frac{1}{\alpha}Q + I)$ and $\alpha \geq 1/(max_{i=1}^{n}\lvert Q_{i,i}\rvert)$
Jacobi	$\pi^{(k+1)} = (1-\omega)\pi^{(k)} + \omega\pi^{(k)}(L+U)D^{-1}$ where $0 < \omega < 2$ is the relaxation factor.
Gauss-Seidel	$\pi^{(k+1)} = (1-\omega)\pi^{(k)} + \omega[(\pi^{(k)}L + \pi^{(k+1)}U)D^{-1}]$ where $0 < \omega < 2$ is the relaxation factor.
Takahashi	1) $\phi_i^{(k)} = \pi_i^{(k)}/(\pi_i^{(k)}e)$ $\quad A_{ij}^{(k)} = \phi_i^{(k)}Q[i,j]e$ 2) aggregated equation system: $\nu^{(k)}A^{(k)} = \nu^{(k)}$ with $\nu^{(k)}e = 1$ 3) block equation systems: for $i = 1, 2, \ldots, N$ $\pi_i^{(k+1)}D[i,i] = \pi_i^{(k+1)}Q[i,i] + \sum_{j<i}\nu_j^{(k)}\phi_j^{(k+1)}Q[j,i]$ $+ \sum_{j>i}\nu_j^{(k)}\phi_j^{(k)}Q[j,i]$ 4) $\pi^{(k+1)} = (\nu_1^{(k)} \cdot \pi_1^{(k+1)}, \ldots, \nu_N^{(k)} \cdot \pi_N^{(k+1)})$
Uniformization	$\pi(t) = \sum_{k=0}^{\infty} e^{-\alpha t}\frac{(\alpha t)^k}{k!}\pi^{(k)}$ where $\pi(t)$ is the transient solution and $\pi^{(k)}$ is obtained as in the Power method.

Notes:

1. Q is the generator matrix of the underlying CTMC.
2. $\pi^{(0)}$ is an appropriately selected initial distribution.
3. $Q = D - (L + U)$, where D is a diagonal matrix and L and U are, respectively, strictly lower and strictly upper triangular matrices.
4. $Q[i,j]$ and $D[i,i]$ are, respectively, blocks of Q and D.
5. e is the column unity vector.
6. A is called the *coupling matrix*.

Table 2.1: Iterative solution methods

for a $j > i$. This means that Gauss-Seidel implies a multiplication that accesses a matrix by columns. All other iteration schemes can be formulated with an access by columns or by rows; in fact, the order in which matrix elements are accessed need not be fixed at all, as long as all nonzero matrix elements are considered. This has been frequently exploited for iterative methods on Kronecker representations, e.g., in [8]. Since access by rows is the same as access by columns on a transposed matrix, we consider it to be part of the interface as well. Decompositional methods like Takahashi's method access submatrices' elements, so the same access pattern used for whole matrices must also be supported for submatrices.

AFI Classes

Motivated by the numerical solution methods, we now define the data structure that represents a transition and the set of functions that comprise the state-level AFI. Different access patterns are made possible through a number of methods that return container objects that, for example, contain elements in a row or column. We also define methods that return information on the number of states of the model (for dimensioning vectors) and on the initial state (for defining an initial distribution, as in uniformization). Accessing elements of the rate matrix through containers hides the enumeration of the states (mapping of the state representation to the row/column index of the state) in the state-level class. The freedom to choose this mapping creates an opportunity to optimize the implementation of the state-level class. Kronecker-based models take particularly great advantage of this property, as described in Section 3.2.

Figure 2.1 shows the interface of the template class used to represent a transition. The template parameters are StateType, RateType, and LabelType, which represent the data types used for states (S), transition rates ($\mathbb{R}$), and transition labels (L), respectively. There are four methods that return the characteristics of a transition. They are row(), col(), rate(), and label(), which are used to access (i.e., read and write), respectively, the starting state and ending state of a transition and a transition rate and label.

```
template <class StateType, class RateType, class LabelType>
class Transition {
 public:
  StateType& row();
  StateType& col();
  RateType& rate();
  LabelType& label();
  RateType& reward(int RewardNumber);
};
```

Figure 2.1: Transition class interface

Each pattern of access to transitions of an *LTS* corresponds to one container class. Therefore, in order for us to provide the numerical solution methods with the different patterns of access they need, the number of methods that return container objects in the AFI must be the same as the number of patterns.

The state-level AFI provides three main container classes that provide access to the whole matrix. They are accessible by the following methods:

- LTSClass::getRow(StateType s, row& r) assigns to r the container object consisting of transitions originating from a given state[2] s.

- LTSClass::getColumn(StateType s, column& c) assigns to c the container object consisting of transitions leading to a given state s.

- LTSClass::getAllEdges(allEdges& e) assigns to e the container object consisting of all transitions. The transitions within the container object are in no particular order.

Each of the methods mentioned above initializes a container object that provides iterator classes that are used to scan through the elements of the container.

To support access to submatrices, the interface contains variants for each of the access patterns we described for the whole matrix; for example, for access by rows, it contains LTS-

[2]When it is clear from the context, we deviate from C++ syntax and remove the *class scope operator* (i.e., ClassName::) from the beginnings of names of class members. For example, we write getRow() instead of LTSClass::getRow().

Class::getSubMatrixByRows(StateType rowstart, StateType rowend, StateType colstart, StateType colend, submatrixbyrow& sm), which assigns to sm the container object consisting of elements of the submatrix specified by the four limiting values for row and column indices. In fact, the set of row (column) indices is partitioned such that indices in one block of the partition are consecutive rows (columns) of the matrix. This means that each block of the row indices partition and each block of the column indices partition uniquely specify a submatrix. The state-level AFI provides supplementary methods for determining the number of blocks of row and column indices partitions. They do so via a trading mechanism, in which the solver specifies a range for the number of blocks of each partition as well as a value it prefers in each of those ranges. The range and the preferred value are separately specified for both the columns and rows of the whole matrix. The state-level object subsequently responds with a specific partition whose number of blocks is within the specified range and corresponds closely to the requested preferred value. If the state-level object is unable to come up with a valid partition, it throws an exception. This mechanism allows the solver to select a level of granularity, since it can determine the total number of states, while the state-level object can choose detailed settings in order to retain efficient access.

Later in this section, we give more details on submatrix, an example container class that LTSClass::getSubmatrix(...) returns. This container provides access to the elements of a submatrix in no specific order. The same ideas used in the definition of submatrix also apply to the definition of container classes that provide other patterns of accesses to the whole matrix and also to the submatrices.

To facilitate the analysis of the LTS, we also need the following methods defined in LTSClass:

- StateType getNumberOfStates(), which returns $|S|$.
- StateType getInitialState(), which returns the index of s_0, the starting state.

In order to have a state-level interface that enables us to compute reward structures for stochastic models, we should also incorporate rate rewards and impulse rewards into the interface. The following methods are defined to allow access to the reward structure:

- LTSClass::getNumberOfRateRewards() and LTSClass::getNumberOfImpulseRewards() return the number of rate rewards defined on the states and the number of impulse rewards defined on the transitions of the LTS, respectively.

- Transition::reward(**int** i) returns the value of the i^{th} impulse reward for a transition.

- LTSClass::getRateReward(**int** i) returns the set of values of the i^{th} rate reward for all the states. The set is provided through a container class that can itself be accessed using its corresponding iterator class.

All of the container classes, their associated iterator classes, the methods returning container objects, and the additional methods mentioned above are encapsulated into an LTSClass class that provides the complete state-level AFI. Note that all of the implementation details of LTSClass are hidden from the solution methods operating on it, and that the only way they can see LTSClass is through its interface, i.e., the state-level AFI.

Example Container Class: submatrix. Figure 2.2 illustrates the interface of the submatrix container class and its corresponding iterator class. submatrix and other container classes are declared as inner classes of the LTSClass class. A container class definition must implement all the functionality described earlier (navigation, dereferencing, and comparison operators) as well as provide a prototype of a particular access method (in this case, access to all elements of a submatrix without any specific order). In order to avoid most of the extra method calls, we have inlined all the method definitions by defining them inside their own classes.

The submatrix class is a container that contains the elements of a submatrix of the rate matrix corresponding to the LTS. In particular,

```
class LTSClass {
  ...
  class submatrix {
  public:
    class iterator {
    public:
      iterator();
      ~iterator();
      typedef iterator self;
      Transition& operator*();
      Transition* operator->();
      self& operator++();
      self& operator--();
      bool end();
      bool operator==(self &it);
      bool operator!=(self &it);
      const self& operator=(self const &it);
    };
    void begin(iterator& it);
  };
  void getSubmatrix(StateType rowstart, StateType rowend,
    StateType colstart, StateType colend, submatrix& sm);
  ...
};
```

Figure 2.2: Container class submatrix and its associated iterator

- submatrix::begin() initializes an iterator such that it corresponds to the first element of the submatrix.
- submatrix::iterator::end() returns true if the iterator is past the last element of the submatrix and false otherwise.
- submatrix::iterator::operator++ (submatrix::iterator::operator−−) moves the iterator to the next (previous) element in the submatrix and returns an iterator for the next (previous) element in the submatrix. These operators, along with submatrix::begin() and submatrix::iterator::end(), make it possible to iterate through all elements of a submatrix.

- submatrix::iterator::operator-> and submatrix::iterator::operator* are dereferencing operators. They make it possible to access the individual components of an element (i.e., a transition object).

- LTSClass::getSubmatrix(...) initializes object sm of class submatrix to the submatrix specified by the range of indices given as parameters.

- submatrix::iterator::operator=, the assignment operator, is used to assign one iterator to another. Notice that the interface should be implemented such that after it1=it2 is executed, it1 and it2 are two *independent* iterators pointing to the same element, i.e., each of them can advance independently of the other. That enables analysis algorithms to have multiple iterators on different parts of the matrix simultaneously.

- Equality (operator==) and inequality (operator!=) operators of the submatrix::iterator class should be implemented such that two iterators are equal if and only if they point to the same element of the matrix.

Much as the submatrix class has been defined to provide access to the elements of submatrices with no specific order, we have similarly defined classes to provide row-oriented and column-oriented access to submatrices. Moreover, row, column, and allEdges, along with their corresponding iterators, have been defined to provide row-oriented and column-oriented access, and access to all transitions (with no specific order) for the whole matrix.

2.3 Evaluation

The state-level AFI defined in this section is clearly good at supporting iterative solution methods that are based on the enumeration of matrix elements. Nevertheless, if we consider a wide range of state-level representations and solution methods, we find certain cases that remain unsupported, such as:

- methods that represent probability distributions with some type of decision diagrams, like MTBDDs [1, 27, 29] or PDGs [4, 11]. These so-called "symbolic" approaches perform an iteration step by multiplying numerical values of subsets of states with subsets of matrix entries instead of single elements. The selection of the considered subsets would make it necessary to reveal the underlying compositional structure of the LTS. However, the "hybrid" approach mentioned in [29] not only has the potential to perform better, but also uses a vector representation. That approach could therefore work with the state-level AFI as it is.

- methods, like the shuffle algorithm by Plateau et al. [24, 39], that are based on a compositional state-space representation and that divide transitions into conjunctions of partial transitions. Unlike the Kronecker approaches employed in Section 3.2, the shuffle algorithm iterates through submodels, so either 1) it needs to reside behind the state-level AFI, while the state-level AFI supports a matrix-vector multiplication, or 2) it needs to be implemented by a solver, in which case the state-level AFI needs to reveal the compositional structure of the LTS.

- methods that use decompositions of a matrix and rely on a specific property, like nearly completely decomposability. So far, a solver can only check whether the derived partition shows the particular property, and has no ability to direct the state-level object in its decisions on how to partition the matrix. There are some heuristic methods to compute NCD partitions [40] that could be implemented to compute partitions for the flat state-level object. The lack of rigorous theoretical results on computing NCD partitions is merely a consequence of the state of the art.

In summary, the basic functionality provided by the state-level AFI is sufficient to allow us to proceed with several solution methods. For specific algorithms, additional functionality may be needed, especially to reveal more information on the structure of the LTS. However, before such extensions can be considered, it is important to ensure that the fundamental approach is applicable in practice and performs sufficiently well. Once that has been established, more elaborate functionality can be built on top of the state-level AFI. In the next chapter, we describe how two different representations of the state transition rate matrix, i.e., “flat” (sparse matrix) and Kronecker representations, are implemented such that they provide state-level AFI.

Chapter 3

Example State-level AFI Implementations

To demonstrate the generality of the AFI developed in Chapter 2, we describe how two conceptually different LTS representations can provide the same state-level AFI. In particular, we have implemented two AFI-compliant state-level classes based on 1) "flat" unstructured LTS representation based on sparse-matrix representation and 2) structured LTS representation amenable to Kronecker representation. To test those representations, we also implemented several solvers that use the state-level AFI to solve models. The set of solvers includes the Jacobi method, SOR, an iterative aggregation/disaggregation method (Takahashi) for stationary analysis, and uniformization for transient analysis. Since all solvers comply with the AFI, we can use any of them with either of the state-level objects to solve the models. In this section, we describe the implementation details of two complete state-level classes.

To make all AFI-compliant numerical solution methods applicable to an AFI-compliant state-level object, we need to implement the complete set of methods described in Section 2.2.3 for the corresponding class. However, for some LTS representations, there may be some access patterns that cannot be implemented efficiently in terms of space or time requirements. In such cases, a mechanism must notify the analysis algorithm that a specific access pattern has not been implemented efficiently for that state-level object. We use the C++ exception-handling mechanism to do that. In particular, if an LTS class does not provide efficient implementation for a particular access pattern X, calling the corresponding getX method raises an exception that is caught by the analysis algorithm. Conceptually, it is a signal to the analysis algorithm that it cannot perform efficiently on this LTS representation.

3.1 Flat State-level Object

In the Möbius modeling tool, the modeler can generate a CTMC from a high-level stochastic model whose transitions' time distributions are all exponential. The CTMC and the associated reward structures are stored in two files that will, in a later phase, be fed into an appropriate numerical solver that solves the CTMC and computes the measures of the model we are interested in. In an attempt to show the generality of the state-level AFI, we wrapped the interface around those two files.

In Möbius, the LTS is stored in a row-oriented format in the file. When read into memory, it is stored in a compressed sparse row format. In order to support both row- and column-oriented access patterns, we had to sacrifice either speed (by converting back and forth between compressed row and column formats) or space (by storing both formats). In order to achieve separation of concerns, it is essential that we be able to dynamically modify the internal data structure of the LTS without changing the interface. In the case of flat LTS representation, we chose to sacrifice speed (and save space), because typical solution methods use only one of the formats during their run-times. Basically, the conversion from one format to another is performed when the solver wants to access the LTS in the format that is not readily available in the state-level object. Notice that this conversion requires only one scan through all the elements of the matrix; that amount of work is constantly proportional to the amount of work needed for a single iteration of any numerical solution method discussed earlier.

We also need to provide methods to access submatrices of the LTS representation (seen as a matrix). It is easy to see how a submatrix with arbitrary dimensions can be extracted from a matrix that is stored in, for example, column-oriented format; we go through each of the appropriate columns one by one, and within each column, we extract the appropriate rows and the corresponding rates. However, we have two options here: 1) to perform the algorithm each time we want to access the elements of a submatrix, or 2) to perform the algorithm only once for all the submatrices needed during the execution of the solution algorithm and change the representation of the LTS from a single sparse matrix to a two-dimensional array whose elements are sparse representations of submatrices.

There are some advantages and disadvantages to each option. By choosing the first one, we can provide both efficient methods for column-oriented access to the whole matrix and methods for submatrix-oriented access, since we keep the sparse column format. The drawback is that accessing submatrices will not be as efficient as possible, especially when the submatrices are small (i.e., the number of blocks of the partitions is large). In that respect, the second option is superior. By providing instant access to the submatrices of a matrix, a 2D array of sparse matrices is the most efficient way to obtain submatrix-oriented access. However, the second option also has two disadvantages. First of all, since the 2D array of submatrices is computed in advance, the partitioning of the matrix also needs to be known in advance. Second, after the representation is transformed, we cannot have efficient column-oriented access to the whole matrix unless we transform the representation back to sparse column format. Since most (if not all) decompositional solution methods fix the partitioning at the beginning of the algorithm and also use only submatrix-oriented access throughout the algorithm, the two disadvantages are not vital; therefore, we chose the second option. Once again, we see how the state-level AFI gives us the freedom to dynamically modify the internal data structure of the state-level object to provide efficient access methods for the solution algorithms based on their pattern of access to the object.

In our implementation, when a solution algorithm asks the flat state-level object for a specific partitioning on columns and rows, the object can conform to the request as long as the number of blocks in each partition (both column indices and row indices partitions) is less than or equal to the number of states. In such a case, the object provides the solution algorithm with submatrices whose sizes differ at most by one.

3.2 Kronecker-based State-level Object

Kronecker representations of CTMCs can result from several modeling formalisms, including generalized stochastic Petri nets (GSPNs). GSPNs are supported by the APNN toolbox [10], which implements many state-based analysis methods using Kronecker representations. In order to achieve an implementation of an LTS interface, we modified SupGSPN, a numerical solver of the APNN toolbox that uses a modular Kronecker representation and improved algorithms from [8]; the improvement is that they avoid binary search, as described in [13]. The state-level AFI imposes a producer-consumer relationship between the LTS object and the solver. The solver consumes the nonzero matrix entries produced by the LTS object. It is possible to relax the coupling between producer and consumer by using a buffer for the produced elements whose capacity is larger than one. To make the matrix-vector multiplication code within SupGSPN be a producer, one changes the pieces that perform multiplications of vector elements with matrix elements into code that writes the matrix elements into the buffer for the consumer. There are at least two ways to arrange the switch of control flow between producer and consumer. One way involves threads; the consumer would be a thread that waits if the buffer is empty and notifies the producer that the buffer should be filled, and the producer would be a thread that fills the buffer until it is full and notifies the consumer that matrix elements in the buffer are ready to be used. This method comes with the overhead of a thread switch for moving the control flow between producer and consumer, but has the potential to be used in a parallel implementation. The second method is to make the consumer call a method of the producer asking it to fill the buffer. This approach implies that the method call also provides an object that encapsulates the state of the producer such that it can avoid costly reinitialization and recomputation by proceeding from the same state from which it returned last time. The state of the producer includes its local variables and the line of code from which the computation will continue. The drawback of this approach is that there must be a method call whenever the buffer needs to be filled. We focus on the latter approach.

In order to implement the allEdges iterator, we modified algorithms *Act-RwCl* and *Act-RwCl*$^{+}$ of [8]. These algorithms are well-suited for the Kronecker representation since they can simply follow the internal structures of the Kronecker terms; however, the resulting sequence of matrix entries will not show an order on row or column indices. To create an iterator, we replaced the multiplication of matrix and vector elements in the last lines, namely 19 and 13, with statements that 1) return the matrix element to the iterator object and 2) ensure that the algorithms proceed to serve a subsequent increment operation right after that line. Algorithms *Act-ClEl*$_2$ and *Act-ClEl*$_2^{+}$ of [8] are used as a basis for the column iterator. They are modified accordingly.

A Kronecker representation of the generator matrix $\mathbf{Q}$ of a CTMC is based in its simplest form on a diagonal matrix $\mathbf{D}$ for the diagonal entries and a sum of terms over all labels $l \in L$. Each term in the sum gives all the rates for all state transitions with that label. Each term is given by a Kronecker product $\otimes$ over the much smaller state-transition matrices of the N components into which the overall model is decomposed. For more details see, for example, [7, 8, 39].

$$\mathbf{Q} = \sum_{l \in L} \omega_l \bigotimes_{i=1}^{N} \mathbf{Q}_l^i + \mathbf{D}$$

In order to implement the submatrix iterators with reasonable efficiency, we support only partitions that contain no more parts than there are states in the first component of the Kronecker representation, i.e., the dimensions of $\mathbf{Q}_l^1$ restrict the granularity of the partition. The advantage of this restriction is that additional effort for the submatrix iterator is restricted to the treatment of the first component $i = 1$ only. That corresponds to the root element of the multi-valued decision diagrams that represent the reachable fraction of the cross-product of component state spaces. If the restriction is prohibitive in any case, one can permute the components to make the one with the largest state space the first one. Alternatively, one can merge components to create one with more states, a procedure known as *grouping*. One can implement the approach in at least two ways. One is to partition the matrices of the first component and modify the Kronecker representation:

$$\mathbf{Q} = \sum_{i,j \in Partition} \sum_{l} \omega_l \mathbf{Q}_l^1[i,j] \otimes \bigotimes_{k=2}^{N} \mathbf{Q}_l^k$$

where $\mathbf{Q}_l^1[i,j](x,y) = \mathbf{Q}_l^1(x,y)$ if x is an element of part i and y is an element of part j, and 0 otherwise. Therefore, $\mathbf{Q}_l^1[i,j]$, and $\mathbf{Q}_l^1$ are of the same dimension. This has been suggested in [31] as a way to achieve a partition of the generator matrix into submatrices that are blocks of columns; that partition turns out to be useful for parallel matrix-vector multiplications. For a partition into more than a few parts, this approach leads to a substantial increase in Kronecker terms. Hence, we follow a different approach that keeps the component matrices as they are. We dynamically restrict the accesses to the multi-valued decision diagrams that perform the projection on the reachable subset of states, in order to consider only states that belong to the required subsets of row and column states.

To avoid permanent creation and destruction of iterator objects and other data structures, the implementation has memory management of its own that recycles memory space. Reusing memory reduces the effort needed to initialize an iterator object. For example, in SOR, a column iterator is needed for each column, but columns are typically accessed in sequential, increasing order, so that if we reuse memory space of the i^{th} iterator, only a partial update of the internals of the $(i+1)^{\text{st}}$ iterator object is necessary upon initialization of the iterator. For the allEdges iterator, we implemented one variant that determines single elements and a second, buffered variant that pulls a set of elements from the Kronecker representation in order to reduce the number of method calls needed to proceed on the Kronecker data structures. Since CMTCs typically have extremely few elements per row or column, other iterators that give an ordered access by columns or rows write all entries into a buffer, so that access to the Kronecker representation takes place only in the iterator begin() method.

In Chapter 4, we consider examples that are formulated as GSPNs. However, the presented approach is rather independent from the modeling formalism used to describe a stochastic discrete event system in Möbius. Any modeling formalism supported by Möbius has to implement a model-level AFI that provides a uniform, notation-independent representation of a discrete event system to an analysis engine. Our presented approach relies only on this model-level AFI to interact with a model in Möbius, and consequently it is completely unaware of the modeling notation used to specify a given model, i.e., it works the same way for stochastic automata networks as for a process algebra like PEPA. The APNN toolbox supports GSPNs, but the applied modular Kronecker representation is the same for model formalisms with synchronizing actions like stochastic automata networks and stochastic process algebras. Again, the fact that we consider GSPNs is not a restriction to our appraoch.

Chapter 4

Performance

In order to be useful, the Möbius state-level AFI must not unacceptably degrade the performance of numerical solvers. Clearly, use of the AFI does not increase the time complexity of numerical solution methods that are based on the explicit enumeration of all transitions in a state-level representation, since in principle one can always implement the AFI using the original numerical solution algorithm and interrupt its enumeration of matrix elements whenever a single entry is considered. The enumeration then continues with a call for the next increment or decrement operator method. This mechanism implies a constant overhead, which is irrelevant in the computation of the order of a numerical solution algorithm. Nevertheless, in practice, constant factors must be sufficiently small.

In this chapter, we evaluate the performance implications of the use of the Möbius state-level AFI for two examples taken from the literature: the Flexible Manufacturing System (FMS) described by Ciardo et al. [14] and a parallel communication protocol (Courier protocol) designed by Woodside and Li [42]. We also compare the efficiency of different methods of accessing the elements of the generator matrix, i.e., the column, allEdges, and submatrix iterators. We consider the two AFI implementations discussed in the previous chapter. The first implementation is based on a sparse-matrix representation of the LTS, and originates from the numerical solver of Möbius and *UltraSAN*. The second AFI implementation uses a Kronecker representation of the LTS and is derived from the SupGSPN numerical solver in the APNN toolbox. Both implementations are evaluated with respect to the existing non-AFI versions of the solvers from which they originated.

We did experiments on different architectures, operating systems, and compilers. For a number of experiments we tried two compiler versions (gcc versions 2.95.2 and 3.2.1) on the same platform. The performance difference between the two compiler versions was no more than 2%, so we decided to perform all the experiments only with version 2.95.2, with which we have had good experiences so far. We observed that with the same compiler version (gcc version 2.95.2) and optimization parameter settings (-O3), the relative performance of two programs varied significantly across platforms. We considered a Sun Enterprise 400MHz, a Sun Ultra60 450MHz running Solaris, and a PIII 1GHz PC running Linux. All the machines had enough RAM to hold all the data needed by the programs. In this chapter, we present the running times on the PIII 1GHz (as reported in [22]) and the Ultra60 450MHz (as reported in [21]) machines. For Solaris machines, we observed that APNN was faster than Kron LTS by about 52%, on average; for the PC running Linux, we observed an inverse relation: the Kron LTS was faster by about 15%, on average. A similar pattern shows itself when we compare Möbius and Flat LTS, i.e., Möbius is faster by about 8% on Solaris machines, and slower by about 5% on Linux machines, on average. We are currently investigating the reasons for that behavior. Our initial hypothesis is that the variations are due to hardware and/or compiler differences across platforms, such as cache size, register bank size, and instruction re-ordering. So far, we conclude that the overhead is overweighted by platform-specific and compiler-specific effects; that it is sufficiently limited to retain the same time complexity; and that the constant factors are almost always less than 2.

4.1 Example Models

In [14], FMS is described to illustrate the benefits of an approximate analysis technique based on decomposition. The model has been used in many papers as a benchmark model for CTMC analysis (e.g., [9, 43]). For simplicity, we consider a variant in which transitions have marking-independent incidence functions and rates. The model is parameterized by the number of parts that circulate in the FMS. The model distinguishes three types of parts, and we assume that the model contains the same number of pieces (N) of each type. For the Kronecker methods we partition the model into three components as in [9].

We also use a GSPN model of a parallel communication software system [42] that has been considered for benchmarking CTMC analysis techniques (for example, see [30]). 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. To obtain a Kronecker representation, we use the same partition into four components that was used in [30].

The dimensions of the CTMCs associated with a number of model configurations are shown in Table 4.1; column $|S|$ shows the number of states, and column $NZ(Q)$ gives the number of off-diagonal nonzero matrix entries in Q. For those model configurations, Jacobi and Gauss-Seidel solvers perform on the Linux and Solaris platforms as shown in Tables 4.2 and 4.3, respectively. In those tables, the first column gives the parameter setting of N or TWS. The other columns refer to results obtained with the original Möbius implementation, the sparse-matrix state-level AFI (Flat LTS) implementation, the APNN toolbox implementation, and the Kronecker state-level AFI (Kron LTS) implementation. For each tool we present the times per iteration, in seconds, spent using the Jacobi and the Gauss-Seidel solution methods. The "slowdown" columns in those tables give the percentage of decrease in speed caused by the overhead of the state-level AFI. For the Möbius and sparse-matrix AFI implementations, the slowdown for the Jacobi solver is computed by subtracting column 2 from column 4 and dividing the result by column 2; likewise, the slowdown for the SOR solver is computed by subtracting column 3 from column 5 and dividing the result by column 3. The same formula is used to compute the slowdown column for comparison between the APNN toolbox and Kron LTS implementations.

Note that on the Linux platform, as mentioned earlier, we often observe a speedup instead

(a) FMS

N	$\|S\|$	$NZ(Q)$
4	35910	237120
5	152712	1111482
6	537768	4205670
7	1639440	13552968
8	4459455	38533968
9	11058190	99075405

(b) Courier protocol

TWS	$\|S\|$	$NZ(Q)$
1	11700	48330
2	84600	410160
3	419400	2281620
4	1632600	9732330
5	5358600	34424280
6	15410250	105345900

Table 4.1: CTMC size of the studied models

of a slowdown when using the state-level AFI, as indicated by the negative values in the "slowdown" columns. The slowdown for the sparse-matrix AFI implementation is always less than or equal to 1% on the Linux platform. Conversely, we often observe slowdown (almost always less than 10%) on the Solaris platform. In solving the models using the Kronecker approach on the Linux platform, we use on average 29% less time for the allEdges iterator (in the Jacobi AFI implementation) and on average 1% less time for the column iterator. Again, inverse observations have been reported on the Solaris platform; in solving models using the Kronecker approach on the Solaris platform, we use on average 57% more time for the allEdges iterator and 47% more for the column iterator.

In the original Möbius and the sparse-matrix AFI implementations, a Gauss-Seidel iteration is on average 20% faster than a Jacobi iteration. The reasons are that 1) computation of $\pi^{(k+1)}$ in each iteration involves only a few accesses to the elements of π and Q, and a few floating-point operations; 2) accessing the memory is much more expensive than a floating-point operation; and 3) in our implementation, the number of memory accesses in the Jacobi method is one more than in the Gauss-Seidel method. That relationship does not hold for Kronecker implementation, which allows the use of more efficient algorithms for enumerating matrix entries in an arbitrary order (the allEdges iterator in the interface) than for an order by columns as required for Gauss-Seidel (the column iterator in the interface) [8]. Therefore, for the APNN toolbox, irrespective of the architecture and operating system, the allEdges iterator remains significantly faster than the column iterator. The results for Jacobi use an implementation of the buffered allEdges iterator with a buffer size of 128 matrix entries. For $N \in \{3, 4, \ldots, 7\}$, we ran a series of experiments with buffer sizes in the range of $2^0, 2^1, \ldots, 2^{14}$. The observed computation times describe a curve that initially decreases sharply, reaches a minimum in an interval that contains 128 matrix entries for all values, and only gradually increases for increasing buffer sizes. Hence, we fixed the buffer size to 128 for the Jacobi allEdges iterator reflected in Tables 4.2 and 4.3.

Table 4.4 shows the performance of our AFI-compliant transient solver on the Linux platform. Each computation time is the average CPU time in seconds taken to perform a single iteration step. In this newly implemented transient solver, uniformization makes use of the allEdges iterator. That is why the running times in Table 4.4 are similar to those observed for the Jacobi method (shown in Table 4.2). The numbers are slightly higher than those for Jacobi because the computation of the transient distribution involves an additional accumulation of vectors.

(a) FMS

N	Möbius		Flat LTS		slowdown %		APNN		Kron LTS		slowdown %	
	JAC	SOR	JAC	SOR	JAC	SOR	JAC	SOR	JAC	SOR	JAC	SOR
4	0.024	0.018	0.023	0.018	-4	0	0.038	0.046	0.029	0.048	-24	4
5	0.112	0.086	0.106	0.086	-5	0	0.178	0.214	0.132	0.223	-26	4
6	0.418	0.323	0.397	0.326	-5	1	0.671	0.809	0.471	0.822	-30	2
7	1.31	1.01	1.24	1.02	-5	1	2.18	2.62	1.48	2.59	-32	-1
8	–[a]	–	–	–	–	–	5.85	7.45	4.18	7.33	-29	-2
9	–	–	–	–	–	–	14.97	19.14	10.73	18.67	-28	-2

(b) Courier protocol

TWS	Möbius		Flat LTS		slowdown %		APNN		Kron LTS		slowdown %	
	JAC	SOR	JAC	SOR	JAC	SOR	JAC	SOR	JAC	SOR	JAC	SOR
1	0.0063	0.0046	0.0058	0.0046	-8	0	0.009	0.017	0.0065	0.017	-28	0
2	0.047	0.037	0.044	0.035	-6	-5	0.096	0.14	0.07	0.134	-27	-4
3	0.257	0.198	0.233	0.195	-9	-2	0.518	0.74	0.37	0.714	-29	-4
4	1.03	0.802	0.954	0.792	-7	-1	2.17	3.05	1.49	2.94	-31	-4
5	–[a]	–	–	–	–	–	7.23	10.51	4.98	10.12	-31	-4
6	–	–	–	–	–	–	21.5	32.01	15.49	30.74	-28	-4

[a]The state space is too large to be explicitly constructed.

Table 4.2: Time per iteration (in seconds) for the studied models on the Linux platform

(a) FMS

N	Möbius		Flat LTS		slowdown %		APNN		Kron LTS		slowdown %	
	JAC	SOR	JAC	SOR	JAC	SOR	JAC	SOR	JAC	SOR	JAC	SOR
4	0.036	0.030	0.037	0.031	3	3	0.033	0.11	0.065	0.17	97	55
5	0.206	0.181	0.220	0.19	7	5	0.14	0.52	0.30	0.82	114	58
6	0.785	0.693	0.844	0.737	8	6	0.77	1.90	1.24	2.81	61	48
7	2.54	2.23	2.72	2.38	7	7	2.45	6.03	3.88	9.24	58	53
8	–[a]	–	–	–	–	–	6.98	17.5	11.0	25.0	58	43
9	–	–	–	–	–	–	17.9	42.1	28.0	65.9	56	57

(b) Courier protocol

TWS	Möbius		Flat LTS		slowdown %		APNN		Kron LTS		slowdown %	
	JAC	SOR	JAC	SOR	JAC	SOR	JAC	SOR	JAC	SOR	JAC	SOR
1	0.0075	0.0059	0.0073	0.0070	-3	19	0.0126	0.0418	0.019	0.060	51	43
2	0.088	0.079	0.088	0.079	0	0	0.0858	0.327	0.143	0.46	67	41
3	0.487	0.423	0.496	0.442	2	4	0.634	1.65	0.831	2.42	31	47
4	2.01	1.76	2.05	1.85	2	5	2.71	7.41	3.56	9.98	31	35
5	–[a]	–	–	–	–	–	9.32	24.5	12.3	34.6	32	41
6	–	–	–	–	–	–	28.6	83.3	36.8	117.0	29	40

[a]The state space is too large to be explicitly constructed.

Table 4.3: Time per iteration (in seconds) for the studied models on the Solaris platform

(a) FMS

N	Flat LTS	Kron LTS
4	0.024	0.030
5	0.116	0.139
6	0.435	0.488
7	1.36	1.53
8	–[a]	4.39
9	–	11.3

(b) Courier protocol

TWS	Flat LTS	Kron LTS
1	0.0056	0.0071
2	0.048	0.073
3	0.257	0.38
4	1.09	1.54
5	–	5.29
6	–	16.2

[a] The state space is too large to be explicitly constructed.

Table 4.4: Time per iteration (in seconds) for the transient solver on the Linux platform

Since we implemented the state-level AFI-compliant transient solver from scratch, it would not be feasible to compare its performance with the corresponding traditional solvers in Möbius and APNN. And since we did not compare the AFI-compliant solver with traditional solvers, we ran the experiments related to the AFI-compliant solver on only one platform, i.e., the Linux platform.

Summary

After collecting and studying the numbers presented above, we conclude that the overhead imposed by the state-level AFI does not change the time complexity of the solvers and that its corresponding constant factor is almost always less than 2. This overhead is so small that it may easily be outbalanced by other implementation- and platform-dependent factors, like the underlying architecture and operating system.

4.2 Comparison of Iterators

In order to measure the efficiency of submatrix access methods in the two state-level classes that we implemented, we could choose either of two approaches: 1) solve the above-mentioned models using Takahashi's method and measure the running time of a single outer iteration, or 2) measure the amount of time it takes to go through all elements of the generator matrix using the submatrix container. The main drawback of the first approach is that each outer iteration of Takahashi's method involves a variable number of inner iterations on diagonal blocks, which makes the running time of a single outer iteration far from meaningful. One advantage of the second approach is that it does not count the time to perform computations specific to a solution method. Another advantage is that we can extend the second approach so that it is used not only to measure the efficiency of the submatrix iterator but also to compare it to other iterators, i.e., col and allEdges, in a fair and insightful way. Recall that in Section 3.1 we described two approaches for providing submatrix access methods. In the first implementation of the flat state-level object, we used the first (inefficient) approach. After comparing the different access methods on the object using the experiments described below, we observed that the approach was too inefficient compared to col and allEdges, so we reimplemented the necessary parts to reflect the second approach. After choosing the second approach for comparing efficiency of iterators, we wrote a simple piece of code that reads all the matrix elements from the state-level object using each type of container class.

Based on our experiments with running our AFI-based implementation of Takahashi's method on Kronecker and flat state-level objects, we determined that the solver can save a considerable amount of time by remembering which submatrices do or do not have nonzero elements in them. The effectiveness of this optimization can be explained by the fact that generator matrices are usually very sparse, which means that a large percentage of the submatrices have only zero elements. In fact, we performed some experiments to determine the ratio of the number of zero submatrices to the total number of submatrices. Our results for the FMS model show that this ratio starts at 0% when the state transition rate matrix is partitioned into 25 submatrices and increases to an average of 98% when it is partitioned into 250,000 submatrices. This very large ratio justifies the use of the optimization. By skipping those zero submatrices, the solver avoids the administrative overhead of setting up an iterator for a submatrix that contains no nonzero elements. Our solver implementation uses a 2D Boolean array to keep track of zero submatrices. We measured the effect of this optimization for a set of partitions of different granularities, i.e., we tried using different numbers of blocks for the partition (while keeping the same number of blocks for column indices and row indices partitions).

Table 4.5 shows the results of the comparison. The numbers reflect the average time in seconds of a single read of all the elements of the state transition rate matrix of the FMS model. We performed the measurements for the two state-level objects, for the col, allEdges, and submatrix iterators, with and without skipping of zero submatrices, and for different numbers of blocks in each of the column indices and row indices partitions. The submatrix iterators are of the allEdges kind, i.e., the elements of each submatrix are iterated through in no specific order. The numbers $10, 20, ..., 500$ under the submatrix iterator header are the numbers of blocks in the partition of the rows and columns of the matrix, e.g., 20 means that the matrix is divided into 400 submatrices.

As we can see for flat state-level objects of large models, as the number of submatrices grows, the time to access all submatrices rises very quickly if we do not use the optimization. However, if zero submatrices are skipped, the overhead of accessing 250,000 rather than 25 submatrices drops below 50%. Because accessing an element has a larger cost in Kronecker representation than in sparse representation, the effect of skipping zero submatrices is observed differently. In the Kronecker case, if zero submatrices are not skipped, we see that there is at most 50% overhead in accessing a large number of submatrices as opposed to a small number of them. If we skip zero submatrices, the overhead is dramatically reduced to under 2%.

We can also use the results in the table to determine how efficient submatrix access methods are relative to the other two methods. For a Kronecker representation, we observe that when we skipped zero submatrices, the largest number of submatrices we tried incurred an average overhead of only 10% compared to the allEdges iterator, which gives, for the whole matrix, the same access pattern that the submatrix iterator does. allEdges is definitely faster than the col iterator, for the reasons mentioned earlier. The results we observed for the flat state-level object reflect some issues that we need to explain. The first issue is the fact that the allEdges iterator is on average 16% slower than the col iterator. Theoretically, one should be able to implement the allEdges iterator as efficiently as col simply by going through all the columns one by one and using the col container for each one. We put that artificial difference into our implementation in order to make the implementation as much as possible like the Möbius implementation of the Jacobi solver, so that we could compare Möbius and Flat LTS as fairly as possible. The second issue is that for large models, it is faster to access the LTS by submatrix than by col or allEdges for any block size. That is because of the optimized and simple data structure of the submatrix container that we described in section 3.1. Again, we could implement both the col and allEdges iterators to perform as well as or better than submatrix (as they do for the Kronecker representation), but in order to compare them fairly (Tables 4.2 and 4.3) to their counterparts in Möbius, we intentionally have not done so.

(a) Flat state-level object

N	col iterator	allEdges iterator	submatrix iterator: Without optimization						With optimization					
			10	20	50	100	200	500	10	20	50	100	200	500
4	0.0028	0.003	0.0018	0.0018	0.003	0.007	0.023	0.13	0.0016	0.0018	0.002	0.0026	0.0044	0.013
5	0.012	0.014	0.0078	0.0082	0.009	0.013	0.029	0.14	0.0078	0.0078	0.0084	0.0088	0.0108	0.02
6	0.041	0.049	0.030	0.030	0.031	0.035	0.051	0.16	0.030	0.030	0.031	0.031	0.033	0.044
7	0.129	0.155	0.096	0.096	0.097	0.101	0.117	0.23	0.096	0.096	0.096	0.097	0.099	0.109

(b) Kronecker state-level object

N	col iterator	allEdges iterator	submatrix iterator: Without optimization						With optimization					
			10	20	50	100	200	500	10	20	50	100	200	500
4	0.042	0.0125	0.014	0.014	0.018	0.029	–[a]	–	0.013	0.014	0.014	0.015	–	–
5	0.188	0.0535	0.057	0.059	0.062	0.075	0.12	–	0.058	0.058	0.059	0.059	0.061	–
6	0.678	0.1925	0.20	0.20	0.21	0.23	0.28	–	0.21	0.21	0.21	0.21	0.21	–
7	2.12	0.589	0.63	0.63	0.64	0.66	0.72	1.07	0.64	0.65	0.65	0.65	0.65	0.67
8	6.02	1.67	1.76	1.77	1.78	1.80	1.86	2.24	1.80	1.80	1.80	1.80	1.81	1.82
9	14.94	4.18	4.47	4.46	4.46	4.47	4.57	5.00	4.55	4.55	4.56	4.56	4.55	4.53

[a] Exceeds the maximum number of states of the first component.

Table 4.5: Comparison of iterators on FMS model on the Linux platform. Numbers are in seconds per iteration.

Chapter 5

Conclusions

In this thesis, we have presented a state-level abstract functional interface for models expressed as labeled transition systems (LTS), and experimentally compared the performance of solvers using our interface with that of standard implementations of solvers. Our interface uses containers and iterators to separate issues related to representation of LTS from issues related to solution of such systems. The use of our interface thus yields an important separation of concerns with significant advantages for research related to state-based analysis methods, as well as for applications that use these methods.

More specifically, we discussed the requirements that a state-level AFI must fulfill to be useful in practice. The presented AFI was designed accordingly, and we described the important design issues involved in implementing the AFI efficiently. In particular, with the help of two examples, we illustrated the usability of our approach and its impact on the performance of different numerical solvers in CTMC analysis. The architecture and compiler determine whether we observe a speedup or slowdown when using the state-level AFI instead of the original implementations of Möbius and the APNN toolbox. We thus conclude that we gain much more from the use of the interface than we lose from the potential minor performance overhead incurred.

It became possible for us to implemented Takahashi's method in our work after we extended (compared to [21]) the AFI to support access to submatrices. Access to submatrices is also useful for parallel numerical solvers. In [31], specific submatrices, namely sets of columns, are required in order to achieve a parallel uniformization for shared memory architectures and Kronecker representation. Therefore, the support of submatrices is useful in broadening the set of sequential and parallel numerical solvers that can work with the AFI.

5.1 Future Work

We are continuing to work on a full integration of different state-space representations in Möbius, based on the new state-level AFI. In addition to implementing known state-space representations, we envision the creation of adaptive state-level AFI objects that modify their internal data structures depending on the usage patterns that are dynamically observed. That way, we could dynamically make use of the space-time trade-off that characterizes different LTS representations.

References

[1] C. Baier, J. P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In *Proc. Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 146–162. Springer, 1999.

[2] M. Bernardo and R. Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202(1-2):1–54, Jul 1998.

[3] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains*. John Wiley & Sons Inc., 1998.

[4] M. Bozga and O. Maler. On the representation of probabilities over structured domains. In *Proc. of Int. Conf. on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 261–273. Springer, 1999.

[5] P. Buchholz. Markovian process algebra: Composition and equivalence. In U. Herzog and M. Rettelbach, editors, *Proc. of the 2nd Work. on Process Algebras and Performance Modelling*, pages 11–30. Arbeitsberichte des IMMD, University of Erlangen, no. 27, 1994.

[6] P. Buchholz. Hierarchical Markovian models: Symmetries and aggregation. *Performance Evaluation*, 22:93–110, 1995.

[7] P. Buchholz. Structured analysis approaches for large Markov chains. *Applied Numerical Mathematics*, 31(4):375–404, 1999.

[8] 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.

[9] P. Buchholz, M. Fischer, and P. Kemper. Distributed steady state analysis using Kronecker algebra. In *Proc. 3rd Int. Workshop on the Numerical Solution of Markov Chains (NSMC'99)*, pages 76–95, Zaragoza, Spain, Sept. 1999.

[10] P. Buchholz, M. Fischer, P. Kemper, and C. Tepper. New features in the APNN toolbox. In P. Kemper, editor, *Tools of Aachen 2001, Int. Multiconference on Measurement, Modelling and Evaluation of Computer-communication Systems*, Tech. report No. 760/2001, pages 62–67. Universität Dortmund, FB Informatik, 2001.

[11] P. Buchholz and P. Kemper. Compact representations of probability distributions in the analysis of superposed GSPNs. In *Proc. of the 9th Int. Workshop on Petri Nets and Perf. Models*, pages 81–90, Aachen, Germany, 2001.

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

[13] G. Ciardo and A. Miner. A data structure for the efficient Kronecker solution of GSPNs. In *Proc. of 8th Int. Workshop on Petri Nets and Performance Models (PNPM'99)*, pages 22–31, 1999.

[14] G. Ciardo and K. S. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993.

[15] G. Clark, T. Courtney, D. Daly, D. D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius modeling tool. In *Proc. of the 9th Int. Workshop on Petri Nets and Performance Models*, pages 241–250, Aachen, Germany, September 2001.

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

[17] D. D. Deavours and W. H. Sanders. An efficient disk-based tool for solving very large Markov models. In *Proceedings of the 9th Int. Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS '97)*, pages 58–71, June 1997.

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

[19] D. D. Deavours and W. H. Sanders. The Möbius execution policy. In *Proc. of 9th Int. Workshop on Petri Nets and Performance Models(PNPM'01)*, pages 135–144, Aachen, Germany, 2001.

[20] D. D. Deavours and W. H. Sanders. Möbius: Framework and atomic models. In *Proc. of 9th Int. Workshop on Petri Nets and Performance Models (PNPM'01)*, pages 251–260, Aachen, Germany, 2001.

[21] 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, London, UK, April 2002.

[22] S. Derisavi, P. Kemper, W. H. Sanders, and T. Courtney. The Möbius state-level abstract functional interface. *Performance Evaluation*, to be published.

[23] J. M. Doyle. Abstract model specification using the Möbius modeling tool. Master's thesis, University of Illinois at Urbana-Champaign, January 2000.

[24] P. Fernandes, B. Plateau, and W. J. Stewart. Efficient descriptor-vector multiplication in stochastic automata networks. *JACM*, 45(3):381–414, 1998.

[25] J. C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In *Proc. of the 8th Conference on Computer-Aided Verification*, volume 1102 of *LNCS*, pages 437–440. Springer, August 1996.

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

[27] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In *Proc. 3rd Int. Workshop on the Numerical Solution of Markov Chains*, pages 188–207, Zaragoza, Spain, 1999.

[28] H. Hermanns and M. Ribaudo. Exploiting symmetries in stochastic process algebras. In *Proc. of ESM'98: 12th European Simulation Multiconference*, pages 763–770, 1998.

[29] J. P. Katoen, M. Kwiatkowska, G. Norman, and D. Parker. Faster and symbolic CTMC model checking. In *Proc. PAPM-PROBMIV'01*, volume 2165 of *LNCS*, pages 23–38. Springer, 2001.

[30] P. Kemper. Numerical analysis of superposed GSPNs. *IEEE Trans. on Software Eng.*, 22(9):615–628, Sep 1996.

[31] P. Kemper. Parallel randomization for large structured Markov chains. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN 2002)*, pages 657–666, Washington DC, US, 2002.

[32] W. J. Knottenbelt. Generalised Markovian analysis of timed transition systems. Master's thesis, University of Cape Town, Cape Town, South Africa, July 1996.

[33] W. J. Knottenbelt and P. G. Harrison. Distributed disk-based solution techniques for large Markov models. In *Proc. of NSMC'99: 3rd International Meeting on the Numerical Solution of Markov Chains*, pages 58–75, Zaragoza, Spain, 1999.

[34] R. Koury, D. F. McAllister, and W. J. Stewart. Methods for computing stationary distributions of nearly-completely-decomposable Markov chains. *SIAM Journal of Algebraic and Discrete Mathematics*, 5(2):164–186, 1984.

[35] K. Lampka and M. Siegle. Symbolic composition within the Möbius framework. In B. Wolfinger and K. Heidtmann, editors, *Leistungs-, Zuverlässigkeits- und Verlässlichkeitsbewertung von Kommunikationsnetzen und verteilten Systemen, 2. MMB Arbeitsgespräch*, pages 63–74. Universität Hamburg, Fachbereich Informatik, Bericht 242, 2002.

[36] 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, Reading, MA, 2001.

[37] W. H. Sanders. Integrated frameworks for multi-level and multi-formalism modeling. In *Proceedings of PNPM'99: 8th Int. Workshop on Petri Nets and Performance Models*, pages 2–9, Zaragoza, Spain, September 1999.

[38] 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, January 1991.

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

[40] W. J. Stewart and W. Wu. Numerical experiments with iteration and aggregation for Markov chains. *ORSA Journal on Computing*, 4(3):336–350, 1992.

[41] Y. Takahashi. A lumping method for numerical calculation of stationary distributions of Markov chains. Technical Report B-18, Department of Information Sciences, Tokyo Institute of Technology, Tokyo, Japan, June 1975.

[42] 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.

[43] P. Ziegler and H. Szczerbicka. A structure based decomposition approach for GSPN. In *Proc. of PNPM'95: 6th Int. Workshop on Petri Nets and Performance Models*, pages 261–270, 1995.