The Möbius State-level Abstract Functional Interface*

Salem Derisavi[1], Peter Kemper[2], William H. Sanders[1], and Tod Courtney[1]

[1] Coordinated Science Laboratory,
Electrical and Computer Engineering Dept.,
and Computer Science Department
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL, U.S.A.
{derisavi,whs,tod}@crhc.uiuc.edu
[2] Informatik IV
Universität Dortmund
D-44221 Dortmund, Germany
kemper@ls4.cs.uni-dortmund.de

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. In this paper, we describe 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 as a set of containers and iterators, yields 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 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 comparison 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 two numerical solvers (Jacobi and SOR). 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.

* This material is based upon work supported in part by the National Science Foundation under Grant No. 9975019, by the Motorola Center for High-Availability System Validation at the University of Illinois (under the umbrella of the Motorola Communications Center), and by the DFG, Collaborative Research Centre 559.

1 Introduction

Model-based evaluation tools have been developed for many different modeling formalisms and use many different model solution techniques. Möbius [14–16,30] 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 separation of concerns between models expressed in different modeling formalisms, model composition and connection methods, and model solvers (e.g., simulators and state-space generators). This is achieved by using the notion of a model-level abstract functional interface (AFI) [14, 19]. 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 solvers without requiring them to understand details of the formalisms in which the constituent models are expressed.

There has 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. For example, there have been many attempts to avoid large state spaces by detecting symmetries in models, and exploiting lumping theorems. Approaches with this aim include stochastic well-formed nets [11], stochastic activity networks (SANs) and replicate/join model composition [31], and stochastic process algebras [2, 4, 22, 24], among others. Attempts to tolerate large state spaces include variations of decision diagrams; they appear in the context of stochastic models as multi-terminal binary decision diagrams (MTBDDs) [1, 23, 25], probabilistic decision diagrams [3, 9], and matrix diagrams [12]. These methods are based on the idea of sharing isomorphic substructures to save space and to gain efficiency. Kronecker representations also allow representation of large state transition systems; different variants exist to reflect a modular [7] or hierarchical structure [5, 6], or to allow matrix entries to be functions [20, 32]. In addition, on-the-fly generation [18] and disk-based methods [17, 27] 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 these 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, these numerical model solution methods typically involve executing a sequence of matrix-vector multiplications on some variant of the generator matrix of the resulting continuous-time Markov chain (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

(see [32], for example), but the abovementioned rather simple methods are the ones that appear most frequently in combination with the techniques discussed above.

This rich variety of techniques to deal 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. By doing so, we can separate state-space 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. Clearly, we are not the first ones to build an interface that allows one to iterate on an LTS; e.g., in the field of protocol verification, the Caesar/Aldebaran tool [21] provides different iterators for this purpose that seem to rely on preprocessor expansion, and in his thesis [28], 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) [29].

By creating a state-level AFI, we achieve a further 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 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 and benchmarking 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, state-space representation, or analysis can be developed independently but can be used with one another. Obviously, tool users profit from this integrated approach, since more state 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 paper is structured as follows: Section 2 specifies the requirements a state-level AFI must meet in order to be effective. In Section 3 we present the state-level AFI in detail, explaining the motivation behind the

choices we made in developing it. In Section 4, 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. Section 5 then analyzes two examples that are frequently considered in the literature for the performance of numerical analysis approaches. In describing the examples, we show that the overhead induced by the Möbius state-level AFI is small and is outweighed by the advantages it achieves.

2 Requirements

The transformation of a model from a high-level, user-oriented representation into a state-level model by a state-space exploration 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. To do that, we must study the type and amount of state-space 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. More details are given in Section 3.

Economy. The effort to support and implement the AFI for a particular state-space representation must be minimal, so as not to put an unnecessary burden on an AFI implementor. For state-space 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-space representation or solution method. For example, many sophisticated algorithms rely on additional structural information. Decompositional methods such as iterative aggregation/disaggregation require a partition of the state space; Kronecker methods are based on a compositional model description; and 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-space representation to exploit the special structure that may be present in the underlying state space, 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 Section 4, 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 spaces.

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-space 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-space 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.

3 State-level AFI Definition

In this section, we will first formalize 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 continuous-time Markov processes, 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 in Section 2. In particular, with respect to the requirements described in the previous section, we address flexibility of the Möbius state-level AFI in Section 3.2, and its functionality and economy in Sections 3.3 and 5. We illustrate the generality of the AFI by implementing two conceptually different state-space representations in Section 4.

3.1 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 $\lambda \in R$
- $\mathcal{R} : S \times \mathbb{N} \rightarrow R$ is the value of the n^{th} rate reward for each state in S
- $\mathcal{C} : \delta \times \mathbb{N} \rightarrow R$ is the the value of the n^{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 exponential rate of the associated 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, where $\lambda \in [0, 1]$ gives the probability of a transition, or weighted automata, where $\lambda > 0$ 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.

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 with $(i, \lambda_e, *, j)$, and 2) a set of n 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 $\mathcal{C}$.

3.2 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" [29] 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 creation of implementations 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 efficiently implement a large class of solution methods. 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 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).

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 on iterator objects and implemented in the iterators that we define as part of the Möbius state-level AFI:

- *Dereferencing*, which enables us to access the object.
- *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.
- *Comparison operators*, which define an order on the objects of the container.

3.3 State-Level AFI Classes

We use containers to represent sets of transitions. Before explaining how we do this, we review several common numerical solution methods to illustrate the access patterns they require from a state-space representation. These patterns will suggest how 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 functions returning information about the whole model (e.g., the number of states) along with a number of functions to facilitate computation of reward structures defined on the models, provides a complete state-level AFI.

Required Operations We now briefly recall iteration schemes of some simple but frequently employed iterative solution methods, namely the Power, Jacobi, and Gauss-Seidel methods for stationary analysis, and uniformization for transient analysis. This review will help us determine both the type of container objects that a state-level object should provide to a solver and also the general information the solver needs concerning a model. Table 1 summarizes the iteration schemes. More details can be found, for example, in [32].

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.
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 Markov process. 2. $\pi^{(0)}$ is an arbitrary 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.	

Table 1. Iterative solution methods

Notably, all four iteration schemes are based on successive vector-matrix multiplications, where matrices P or Q are only minor transformations of the rate matrix R given by an *LTS* as mentioned above. 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 $\pi^{(k+1)}(j)$ for a $j > i$. This means that Gauss-Seidel implies a multiplication that accesses a matrix by columns. All other iteration schemes can also 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 [7]. 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.

AFI Classes Motivated by these 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 functions that return container objects that, for example, contain elements in a row or column. We also define functions that return information

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

Fig. 1. `Transition` class interface

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 4.2.

Figure 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 (R), and transition labels (L), respectively. There are a number of methods that return the characteristics of a transition. They are `row()`, `column()`, `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.

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 discussed in Section 3.3 with the different patterns of access they need, the number of functions that return container objects in the AFI must be the same as the number of patterns.

Three container classes are provided by the state-level AFI. They are accessible by the following methods:

- `LTSClass::getRow(Transition::StateType s, row& r)` assigns to `r` the container consisting of transitions originating from a given state `s`.
- `LTSClass::getColumn(Transition::StateType s, column& c)` assigns to `c` the container consisting of transitions leading to a given state `s`.
- `LTSClass::getAllEdges(allEdges& e)` assigns to `e` the container consisting of all transitions.

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

In Section 3.3 we give more details on `column`, an example container class that `LTSClass::getColumn()` returns. The same ideas apply to the definitions of the container classes that the other methods return.

To facilitate the analysis of the *LTS*, we also need the following functions:

- `Transition::StateType LTSClass::getNumberOfStates()` returns $|S|$.
- `Transition::StateType LTSClass::getInitialState()` returns s_0.

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 are defined to allow access to the reward structure:

- **int** `LTSClass::getNumberOfRewards()` returns the number of reward structures defined on the *LTS*.
- `Transition::RateType Transition::reward(`**int** n`)` returns the value of the n^{th} impulse reward for a transition.
- `Transition::RateType LTSClass::getRateReward(`**int** n`)` returns the set of values of the n^{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. The values are given in the order of the indices of the states, i.e., the first value corresponds to state 0, the second value to state 1, and so on.

All of the container classes, their associated iterator classes, the functions returning container objects, and the additional functions 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: `column` Figure 2 illustrates a container and a particular iterator class interface (`get_column`). A container class definition must implement all the functionality described earlier (dereferencing, navigation operators, and comparison operators) as well as provide a prototype of a particular access method (in this case, access by columns). In order to avoid most of the extra function calls, we have inlined all the member function definitions by defining them inside their own classes.

The `column` object is a container that contains the elements of a column of the rate matrix corresponding to the *LTS*. In particular,

- `column::begin()` initializes an iterator such that it corresponds to the first element of the column.
- `column::iterator::end()` returns true if the iterator is past the last element of the column and false otherwise.
- `column::iterator::operator++` (`column::iterator::operator--`) advances the iterator to point to the next (previous) element in the column and returns an iterator for the next (previous) element in the column. This operator, along with `column::begin()`, `column::iterator::end()`, makes it possible to iterate through all elements of a column of the matrix.

```
class LTSClass {
  ...
  class column {
  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 getColumn(Transition::StateType _col_index, column& c);
  ...
};
```

Fig. 2. Container class `column` and its associated iterator

- `LTSClass::getColumn()` initializes the `c` object of class `column` to the specified column of the matrix.
- `column::iterator::operator=`, the assignment operator, is used to assign one iterator to another. Notice that the interface should be implemented in such a way that just after `it1=it2` is executed, `it1` and `it2` are two *independent* iterators to the same element in the column, i.e., each of them can advance regardless of the others. This will enable the analysis algorithm to have multiple iterators on different parts of the matrix simultaneously.
- Equality and inequality operators (`operator==` and `operator!=` functions of `column::iterator` class) should be implemented such that two iterators are equal if and only if they point to the same element of the matrix.

Similar to the way `column` class has been defined to provide column-oriented access to the matrix, `row` and `allEdges`, along with their corresponding iterators, can be defined to provide row-oriented access and access to all transitions (with no specific order).

3.4 Evaluation

The state-level AFI defined in this section clearly supports iterative solution methods that are based on the enumeration of matrix elements quite well. Nev-

ertheless, if we consider the wide range of specific state-level representations and solution methods, we find certain cases that remain unsupported so far, e.g.,

- methods that represent probability distributions by some type of decision diagrams, like MTBDDs [1,23,25] or PDGs [3,9]. 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 [25] not only has the potential to perform better, but also uses a vector representation. This approach could therefore work with the state-level AFI as it is.
- methods, like the shuffle algorithm by Plateau et al. [20, 32], 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 4.2, 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 is 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. Decompositional methods, e.g., the one described in Chapter 6 of [32], consider block-structured matrices and require iterators for block-access of matrices. This is a minor extension to the set of iterators defined so far. An issue that is even more delicate in theory is that of providing additional information on a useful, effective partition into blocks, e.g., to achieve decomposition that is nearly-completely-decomposable (NCD), a property that is important for the efficiency of decompositional methods.

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 we defined.

4 Example State-Space Implementations

To demonstrate the generality of the AFI developed in Section 3, we describe how two conceptually different state-space representations can provide the same state-level AFI. In particular, we have implemented two AFI-compliant state-level objects: "flat" unstructured state spaces based on sparse-matrix representation, and structured state spaces amenable to Kronecker representation. To test these representations, we also implemented two solvers (using the Jacobi and Gauss-Seidel methods) that use the state-level AFI to solve models. Since both

solvers comply with the AFI, we can use either 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 functions described in Section 3.3 for the object. However, for some state-space representations, there may be some access patterns that cannot be implemented efficiently. For example, if we have a sparse-matrix representation for the state space stored in a row-oriented order, it may be considered inefficient (in terms of space or time) to provide both row- and column-oriented access functions. 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 this. In particular, if a state-space class does not provide efficient implementation for a particular access pattern `X`, calling the corresponding `getX` function raises an exception that is caught by the analysis algorithm. Conceptually, this is a signal to the analysis algorithm that it cannot perform efficiently on this state-space representation.

4.1 Flat State-Space 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 be, in a later phase, fed into an appropriate numerical solver that solves the Markov chain 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 this interface around the two files.

In Möbius, the state space 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 state space without changing the interface. In the case of flat state-space representation, we chose to sacrifice speed (and save space) because typical solution methods use only one of the formats during their run-times. Considering this, the conversion from one format to another is not done if the solver requires row-oriented access, and is done only once if the solver requires column-oriented access. Notice that this conversion needs only one scan through all the elements of the matrix; that is roughly the same amount of work needed for a single iteration of a fixed-point numerical solution method.

4.2 Kronecker-based State-Space 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 of [7]; the improvement is its avoidance of binary search, as described in [12]. In the following, we focus on iterators to support Jacobi and Gauss-Seidel iteration schemes for fixed-point iteration. One observation in [7] is that a Jacobi solver need not impose any kind of order on a matrix-vector multiplication in an iteration step, and that that degree of freedom can be extensively used to optimize enumeration of matrix elements stored in a Kronecker representation. Jacobi solvers match the behavior of the `allEdges` iterator in the LTS interface. The SupGSPN implementation of the originally recursive algorithm of [7] performs iteratively with an internal data structure to store local variables used in the corresponding recursive approach. Therefore, iteration through a Kronecker representation is far from trivial if performance is of importance. For instance, directed acyclic graphs (or multi-value decision diagrams) are used to restrict the matrix vector multiplication on the reachable set of states. In addition to making obvious functional changes, we moved all local variables and local data structures into an iterator object, such that an increment operation can continue iterating through the Kronecker representation at the very position the last increment operation left the function and with the same local variable space. We modified algorithms *Act-RwCl* and *Act-RwCl*$^+$ of [7] to support an `allEdges` iterator by replacing the multiplication of matrix and vector elements in the last lines, namely 19 and 13, by 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 [7] are modified accordingly for a column iterator.

To avoid permanent creation and destruction of iterator objects and other data structures, the implementation has a 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 only a partial update of the internals of the $(i+1)^{\text{th}}$ iterator object is necessary upon creation if we reuse memory space of the i^{th} iterator. For the `allEdges` iterator, we implemented one variant that determines single elements and a second, buffered one that pulls a set of elements from the Kronecker representation in order to reduce the number of function calls. Example results suggest that it would be helpful to employ buffers of around 128 elements to exploit the tradeoff between handling additional data structures and saving function calls. 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()` function. The current implementation mixes C code for accessing Kronecker data structures and C++ code for the LTS and iterator classes.

5 Performance

In order to be useful, the Möbius state-level AFI must not unacceptably degrade the performance of numerical solvers. Clearly, using 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 operation. 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 section, we evaluate the performance implications of the use of the Möbius state-level AFI for two examples taken from the literature. We consider the two AFI implementations discussed in the previous section. The first implementation is based on a flat sparse-matrix state-space representation, and originates from the numerical solver of Möbius and *UltraSAN*. The second AFI implementation uses a Kronecker state-space representation 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 originate.

We did experiments on different architectures and operating systems, and observed that with the same compiler (gcc version 2.95.2) and optimization parameter settings (-O4), the relative performance of two programs varied significantly across platforms. We considered a Sun Enterprise 400MHz, a Sun Ultra60 450MHz, and a PIII 1GHz PC running Linux. All the machines had enough RAM to hold all the data needed by the programs. The given numbers in the following tables present the running times on the Ultra60 machine. For a Sun Enterprise we observed that APNN was faster than KronLTS by 30 to 60% , and for the PC we observed an inverse relation: the KronLTS was faster by about 20%. We are currently investigating the reasons for the behavior in further detail. 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 sufficiently limited to retain the same time complexity, and that the constant factors are almost always less than 2.

5.1 Flexible Manufacturing System Example

Ciardo et al. [13] describe a flexible manufacturing system to illustrate the benefits of an approximate analysis technique based on decomposition. This model has been used in many papers as a benchmark model for CTMC analysis (e.g., [8, 34]). 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 there are the same number of pieces (N) of

each type. For the Kronecker methods we partition the model into three components as in [8]. The dimensions of the associated CTMC are shown in Table 2(a); 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, the implementations of a Jacobi and a Gauss-Seidel solver perform as shown in Table 3(a). The first column gives the parameter setting of N. The other columns refer to results obtained with the original Möbius implementation, the sparse-matrix AFI state-space implementation, the APNN toolbox implementation, and the Kronecker AFI state-space implementation. For each tool we present the times using the Jacobi and the Gauss-Seidel solution methods. Each computation time is the average time for a single iteration step, as described in Section 3.3; that is the CPU time in seconds taken to perform one iteration.

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

(a) FMS

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

(b) Courier protocol

Table 2. State spaces of the studied models

In the original Möbius and the sparse-matrix AFI implementations, a Gauss-Seidel iteration is about 15% faster than a Jacobi iteration. This happens because 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 accesses to memory (specifically, an element of π) in the Jacobi method is one more than in the Gauss-Seidel method. This is different for Kronecker implementations, which allow 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 [7]. The "slowdown" columns in Table 3(a) 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 (SOR) solver is computed by subtracting column two (three) from column four (five) and dividing the result by column two (three). The same formula is used to compute the slowdown column for comparison between the APNN toolbox and Kronecker AFI state-space implementations. As can be seen, the slowdown for the sparse-matrix AFI implementation is almost always less than 10%. In solving the larger models using

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

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

(a) FMS

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.

(b) Courier protocol

Table 3. Time per iteration (in seconds) for the studied models

the Kronecker approach, we use on average 74% more time for the `allEdges` iterator (in the Jacobi AFI implementation) and on average 52% more for the `column` iterator. Nevertheless, the `allEdges` iterator remains significantly faster than the `column` iterator (in the SOR implementation). The results use an implementation of an `allEdges` iterator that uses an internal buffer of 128 matrix entries in order to reduce the number of function calls needed to proceed on the Kronecker data structures. For $N \in \{3, 4, \ldots, 7\}$, we run an experiment series 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 Table 3(a).

5.2 Courier Protocol Software Example

In this subsection, we consider a GSPN model of a parallel communication software system that was originally designed by Woodside and Li [33], and has been

considered for benchmarking CTMC analysis techniques (for example, see [26]). 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 [26]. Table 2(b) shows the number of states in the CTMC and the off-diagonal nonzero entries, much like Table 2(a). Table 3(b) gives the computation times we observed on average for a single iteration step for this model. The selection of columns is the same as in Table 3(a). The results are slightly different from the values we observed for the FMS example. The overhead imposed by the interface is still low in the case of sparse matrices. For the Kronecker approach, the overhead is on average 40% for Jacobi and 41% for SOR. We also exercised different buffer sizes and observed that 128 is always in the range that contains the minimum overhead.

6 Conclusions

In this paper, we have presented a state-level abstract functional interface for models expressed as labeled transition systems, and experimentally evaluated the performance of that interface compared to standard implementations. This interface uses containers and iterators to separate issues related to representing labeled transition systems from issues related to solving such systems. The use of this 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 our experiences and the important design issues involved in implementing this 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 presented results show that for simple LTS representations, the overhead is negligible, and for complex data structures such as are used for Kronecker representations, computation times increase by a constant factor that is almost always less than 2. We thus conclude that we gain much more from the use of the interface than we lose from the minor performance overhead incurred.

Our ongoing work is on a full integration of different state-space representations in Möbius, based on the AFI and the new state-level AFI. In addition to implementing known state-space representation techniques, we envision the creation of adaptive state-level AFI implementations that internally modify themselves or switch over to other implementations 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 labeled transition system representations.

Acknowledgments We thank Graham Clark for his comments and ideas in our discussions on the design of Möbius interfaces.

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)*, pages 146–162. Springer LNCS 1664, 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. M. Bozga and O. Maler. On the representation of probabilities over structured domains. In *Proc. CAV'99*, Springer, LNCS 1633, pages 261–273, 1999.
4. 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.
5. P. Buchholz. Hierarchical Markovian models: Symmetries and aggregation. *Performance Evaluation*, 22:93–110, 1995.
6. P. Buchholz. Structured analysis approaches for large Markov chains. *Applied Numerical Mathematics*, 31(4):375–404, 1999.
7. 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), 2000.
8. 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.
9. 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.
10. P. Buchholz, 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. Universität Dortmund, FB Informatik, 2001.
11. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed coloured nets for symmetric modelling applications. *IEEE Transactions on Computers*, 42(11):1343–1360, Nov 1993.
12. G. Ciardo and A. Miner. A data structure for the efficient Kronecker solution of GSPNs. In *Proc. of PNPM'99: 8th Int. Workshop on Petri Nets and Performance Models*, pages 22–31, 1999.
13. G. Ciardo and K. S. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993.
14. G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster. The Möbius modeling tool. In *Proc. of the 9th Int. Workshop on Petri Nets and Performance Models*, pages 241–250, Aachen, Germany, September 2001.
15. D. Deavours and W. H. Sanders. The Möbius execution policy. In *Proc. of PNPM'01: 9th Int. Workshop on Petri Nets and Performance Models*, pages 135–144, Aachen, Germany, 2001.
16. D. Deavours and W. H. Sanders. Möbius: Framework and atomic models. In *Proc. of PNPM'01: 9th Int. Workshop on Petri Nets and Performance Models*, pages 251–260, Aachen, Germany, 2001.

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. J. M. Doyle. Abstract model specification using the Möbius modeling tool. Master's thesis, University of Illinois at Urbana-Champaign, January 2000.
20. P. Fernandes, B. Plateau, and W. J. Stewart. Efficient descriptor-vector multiplication in stochastic automata networks. *JACM*, 45(3):381–414, 1998.
21. 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, New Brunswick, USA, August 1996.
22. S. Gilmore, J. Hillston, and M. Ribaudo. An efficient algorithm for aggregating PEPA models. *Software Engineering*, 27(5):449–464, 2001.
23. 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.
24. H. Hermanns and M. Ribaudo. Exploiting symmetries in stochastic process algebras. In *Proc. of ESM'98: 12th European Simulation Multiconference*, 1998.
25. J. P. Katoen, M. Kwiatkowska, G. Norman, and D. Parker. Faster and symbolic CTMC model checking. In *Proc. PAPM-PROBMIV'01*, pages 23–38. Springer LNCS 2165, 2001.
26. P. Kemper. Numerical analysis of superposed GSPNs. *IEEE Trans. on Software Eng.*, 22(9):615–628, Sep 1996.
27. W. 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, Zaragoza, Spain*, pages 58–75, 1999.
28. W.J. Knottenbelt. Generalised Markovian analysis of timed transition systems. Master's thesis, University of Cape Town, Cape Town, South Africa, July 1996.
29. 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.
30. 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, Zaragoza, Spain*, pages 2–9, September 1999.
31. 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.
32. W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press, 1994.
33. 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.
34. 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.