Can matrix-layout-independent numerical solvers be efficient?

Implementing the Moebius State-Level Abstract Functional Interface for ZDDs

K. Lampka, S. Harwarth, M. Siegle[*]
University of the Federal Armed Forces
Munich, Germany
{kai.lampka,stefan.harwarth,markus.siegle}@unibw.de

ABSTRACT

Symbolic approaches based on decision diagrams have shown to be well suited for representing very large continuous-time Markov chains (CTMC), as derived from high-level model descriptions. Unfortunately, each type of decision diagram requires its own implementation of the numerical solvers for computing the state probabilities of the CTMC. For this reason, some time ago the idea of separating numerical solution methods from the representation of the CTMC was proposed [12], suggesting the implementation of a so-called state-level abstract functional interface (AFI), which defines classes of iterators for accessing the entries of the CTMC transition rate matrix. In this paper we (a) present an implementation of the AFI for zero-suppressed multi-terminal binary decision diagrams (ZDDs) [18] and (b) empirically investigate the viability of matrix-layout-independent implementations of numerical solvers.

1. INTRODUCTION

High-level description methods for Markovian models, such as stochastic Petri nets, stochastic process algebras, among others, have shown to be powerful formalisms for describing and analyzing concurrent systems. The first step of quantitative analysis of such models is the generation of a continuous-time Markov chain (CTMC), where the interleaving semantics of standard high-level model description methods yields the explicit extraction of all possible execution sequences of system activities. This may lead to an exponential blow-up in the number of system states, commonly known as *state space explosion*, which may prevent or at least hamper the analysis of complex and large systems. Once the CTMC has been generated, steady-state or transient probabilities can be computed using a numerical algorithm which accesses the entries of the transition rate matrix.

Many approaches have been developed in order to cope with state space explosion on the one hand and limited availability of memory and CPU time on the other hand. In the context of numerical solvers, techniques for storing and handling large CTMCs can be grouped into the following four classes:

1. Methods exploiting mass storage and/or distributed hardware (e.g. [17, 15, 22]).
2. Reduction and symmetry exploitation techniques based on state lumping (e.g. [27, 28, 16, 1, 11]).
3. Implicit (or Kronecker-operator-based) representation techniques (e.g. [26, 3, 5, 4, 7]).
4. Methods employing decision diagrams [6, 24, 29, 30].

Decision diagrams (DD) enable both, the efficient exploration of huge CTMCs, and their compact storage. In [18] we introduced partially shared zero-suppressed multi-terminal binary decision diagrams (ZDD for short), and a new method for efficiently deriving a ZDD-based representation of a high-level model's CTMC. This scheme was extended in [19] to the case of high-level performability models, where we also presented a ZDD-based variant of the hybrid solution method which had been previously developed for multi-terminal binary decision diagrams (MTBDDs) by Parker [25]. Altogether, these innovations enable us to efficiently compute performability measures on commodity computers for CTMCs with more than, say, 10^8 states.

Since different techniques employ different data structures for storing the transition rate matrix (also referred to as the "state-level object"), a given iterative solution method usually has to be be re-implemented for each new representation format. Thus, it seems very useful to employ a generic interface, which separates the concerns of matrix representation on the one hand, and implementation of numerical solvers on the other hand. In order to make matrix representation and numerical solvers independent of each other, the authors of [12] presented the idea of a state-level abstract functional interface (AFI). For demonstrating the competitiveness of such an interface, the authors provided an implementation of the AFI within the Moebius modelling framework [9], where the following matrix layouts were employed:

1. Flat explicit storage, where the transition rate matrix is stored in a straight-forward manner, employing the well-known sparse matrix technique.

[*]supported by DFG grants SI 710/2 and SI 710/3

2. Kronecker representation, where the transition rate matrix is represented implicitly. On each access to a matrix element, a Kronecker expression needs to be evaluated for a set of (local) transition rate matrices.

In this paper, we also describe an implementation of the AFI within the Moebius modelling framework [9], but based on the ZDD data structure. For carrying out a sound analysis, not only of our AFI implementation, but for the AFI in general, two typical benchmark models are analyzed, where different standard solution methods in combination with different matrix storage formats, ranging from sparse matrix storage formats over matrix diagrams (MxD) [24] up to our new type of DD, are employed.

1.1 The Moebius Performability Evaluation Tool

Moebius [9] is a comprehensive software tool for the modelling and evaluation of discrete-event systems, developed by W.H. Sanders and his group at the University of Illinois. Moebius supports several formalisms for model specification and offers different analysis methods for the derivation of quantitative performability measures, in particular numerical analysis of Markov chains and discrete-event simulation. The present paper concentrates on Moebius' state-level abstract functional interface (state-level AFI) [12, 10] through which the various numerical solvers access the state-level object that was generated from the high-level model description[1].

1.2 Organization

The paper is further organized as follows: Sec. 2 presents details about our implementation of symbolic matrix representation and our ZDD-based implementation of the AFI. Empirical results, obtained from our proprietary ZDD-based solvers, our implementation of the AFI, as well as its sparse matrix format and MxD-based implementation [12], are presented in Sec. 3, and Sec. 4 concludes the paper.

2. THE ZDD-BASED AFI

Our ZDD approach consists of two parts: (1) First we generate the symbolic representation of a given high-level model's underlying CTMC (including the symbolic representation of rate and impulse reward functions). (2) Either by making use of the Moebius solvers, which access the symbolic representations of the CTMC transition rate matrix via the AFI, or by using our proprietary non-AFI solvers, we then compute the desired state probabilities.

In the following, we briefly explain how ZDDs are employed for symbolically representing CTMCs in a compact way. Our semi-symbolic technique[2] for efficiently generating symbolic representations of CTMCs was explained in detail in [19]. The symbolic handling of reward functions is discussed in [19]. We will also discuss ZDD-based matrix representation, which is a prerequisite for understanding the matrix iterators as implemented within the ZDD-based version of the AFI. These iterators are described in Sec. 2.3.

[1]Moebius' so-called model-level AFI, which constitutes the interface between the user-level model description and the tool's internal data structures, is not of interest for the present paper.

[2]We call this technique "semi-symbolic", since it combines explicit exploration with symbolic composition and symbolic reachability analysis. The explicit exploration is thereby limited to sequences of dependent activities, such that in practice only a small fraction of the transitions needs to be explored.

2.1 ZDD-based representation of activity-labelled CTMCs

2.1.1 *Preliminaries:*

A high-level model M consists of a finite ordered set of discrete state variables (SVs) $\mathfrak{s}_i \in S$, where each can take values from a finite subset of the naturals. Each state of the model is thus given as a vector $\vec{s} \in \mathbb{S} \subset \mathbb{N}^{|S|}$, where $\vec{s}[i]$ refers to the current value of the i'th SV in state $\vec{s}$. When an activity is executed, the model evolves from one state to another. For each activity l we have a transition function $\delta_l : \mathbb{S} \longrightarrow \mathbb{S}$, the specific implementation of which depends on the model description method. These transition functions allow one to map a high-level model M to its underlying activity-labelled CTMC (*aCTMC*). If activity labels are removed, transitions between the same pair of states are aggregated via summation of the individual rates.

2.1.2 *The ZDD data structure:*

Zero-suppressed BDDs (z-BDDs) [23] are derivatives of ordered BDDs for representing sparse sets efficiently. In z-BDDs, one eliminates all non-terminal nodes which have the terminal 0-node as their 1-successor (this is called the zero-suppressing reduction rule). We allow z-BDDs to have more than two terminal nodes, thereby obtaining zero-suppressed multi-terminal binary decision diagrams (z-MTBDDs, or ZDDs for short). ZDDs are a weakly canonical representation of pseudo-Boolean functions. Standard arithmetic operators can be performed efficiently on them with the help of a variant of Bryant's `Apply`-algorithm [2]. For simplicity, a complete (formal) definition and details of the algorithms are skipped here (they will be published in a forthcoming paper [20]), but for convenience some elements of ZDDs are introduced now.

1. The disjoint sets of non-terminal nodes ($\mathcal{K}_{NT}$) and terminal nodes ($\mathcal{K}_T$).

2. A finite (possibly empty) set of Boolean variables $\mathcal{V}$ with a strict total ordering π. Within an ordered DD environment, all nodes labelled with the same Boolean variable $\mathsf{x}_i \in \mathcal{V}$ appear at level i.

3. The function `then` $: \mathcal{K}_{NT} \mapsto \mathcal{K}_{NT} \cup \mathcal{K}_T$, which yields the *then* or *1*-child of node n.

4. The function `else` $: \mathcal{K}_{NT} \mapsto \mathcal{K}_{NT} \cup \mathcal{K}_T$, which yields the *else* or *0*-child of node n.

5. The function `value` $: \mathcal{K}_T \mapsto \mathbb{D}$ maps a terminal node to an element of the finite set $\mathbb{D}$, where usually $\mathbb{D} \subset \mathbb{R}^+$.

2.1.3 *ZDD based representation of aCTMCs:*

Let the transitions defining an *aCTMC* T be given as quadruples $(a, \vec{s}, \vec{t}, \mu)$, where $\vec{s}, \vec{t} \in \mathbb{S}$, the label a is an element of the finite set of activity labels $\mathcal{A}ct$ and the rate μ is an element of $\mathbb{D} \subset \mathbb{R}^+$. If one defines an adequate Boolean encoding function $\mathcal{E} : \mathcal{A}ct \times \mathbb{S} \times \mathbb{S} \mapsto \mathbb{B}^{n_\mathcal{V}}$ for the transitions, one ends up with a function table of a pseudo-Boolean function, where the transition rates are considered as being the function values. A ZDD T is a symbolic representation of a given *aCTMC* T if the following holds for all transitions of T:

$$f_{\mathsf{T}}(\mathcal{E}(a,\vec{s},\vec{t}) = \mu \quad \Leftrightarrow \quad (a,s,t,\mu) \in T$$

(The notation f_{T} denotes the function represented by ZDD T). In case $(a,s,t,\mu) \notin T$ the characteristic function f_{T} evaluates to 0. Within our model world, we employ the following ordered sets of Boolean variables:

1. $\vec{\mathsf{a}} := (\mathsf{a}_1,\ldots,\mathsf{a}_{B_{\mathcal{A}ct}})$ for encoding the activity labels,
2. $\vec{\mathsf{s}}^{\,i} := (\mathsf{s}^i_1,\ldots,\mathsf{s}^i_{B_i})$ for encoding the SV $\mathfrak{s}_i$ in the source state $\vec{s}$ of a transition,
3. $\vec{\mathsf{t}}^{\,i} := (\mathsf{t}^i_1,\ldots,\mathsf{t}^i_{B_i})$ for encoding the SV $\mathfrak{s}_i$ in the target state $\vec{t}$ of a transition.

The s- and t-variables are collected as two ordered tuples, where a most-significant bit first order is assumed, yielding:

$$\begin{array}{l}\vec{\mathsf{s}} := (\mathsf{s}_1,\ldots,\mathsf{s}_m) := (\mathsf{s}^1_{B_1},\ldots,\mathsf{s}^1_1,\ldots,\mathsf{s}^n_{B_n},\ldots\mathsf{s}^n_1) \text{ and} \\ \vec{\mathsf{t}} := (\mathsf{t}_1,\ldots,\mathsf{t}_m) := (\mathsf{t}^1_{B_1},\ldots,\mathsf{t}^1_1,\ldots,\mathsf{t}^n_{B_n},\ldots\mathsf{t}^n_1).\end{array} \qquad (1)$$

In order to keep the DDs small, we employ a variable ordering in which the Boolean vector $\vec{\mathsf{a}}$ appears first. Starting at level $B_{\mathcal{A}ct}+1$ the Boolean vectors encoding source and target states follow in an interleaved fashion. This interleaved ordering of s and t variables is a commonly accepted heuristics for obtaining small DD sizes [13, 14, 29].
As an example, the reader is referred to Fig. 1. Part (i) depicts a small *aCTMC* T, where the states consist of two SVs only $(\mathfrak{s}_1,\mathfrak{s}_2)$. The binary encodings of the transitions is given in Fig. 1.ii. This function table defines a pseudo-Boolean function, which enables one to construct the respective ZDD B, as depicted in Fig. 1.iii.

It has been (empirically) found that the ZDD-based representation is more compact than the representation based on standard MTBDDs, where a factor of approximately two to three in space and runtime to the advantage of ZDDs has been observed. This not only has the positive effect that the construction and manipulation times for the symbolic representation are reduced, but also memory space and CPU time required for computing state probabilities are reduced by about the same factor [18, 19, 21].

2.2 The hybrid approach to computing state probabilities

2.2.1 Preliminaries

(a) ZDD based representations of matrices: If row and column indices of a matrix M are encoded in binary form, each real-valued $(2^n \times 2^n)$ matrix M can be interpreted as a pseudo-Boolean function, so that $f_{\mathsf{M}}(\mathcal{E}(r,c)) = M(r,c)$, with $|\mathcal{V}| = 2n$. The connection to the ZDD based encoding of an *aCTMC* is as follows: One abstracts from the Boolean variables encoding the activity labels (the a-variables), and interprets the Boolean variables collected in the vectors $\vec{\mathsf{s}}$ and $\vec{\mathsf{t}}$ (cf. Eq. 1) as binary encoded row and column indices. An example is shown in Fig. 1: Abstracting from the activity labels (Fig. 1.iii), one obtains ZDD A which directly encodes the underlying transition rate matrix as shown in Fig. 1.iv.

(b) Access-pattern to the matrix entries: Since we defined a “most-significant-bit-first” ordering as well as an interleaved ordering of the $\vec{\mathsf{s}}$ and $\vec{\mathsf{t}}$ variables, a depth-first traversal of the ZDDs realizes a block-wise access-pattern to the elements of the represented matrix. I.e. the Boolean expansion for variable s_1 is $f_{\mathsf{M}} = \mathsf{s}_1 f^{\mathsf{M}}_1 + \neg\mathsf{s}_1 f^{\mathsf{M}}_0$, where the respective (sub-)graphs of $f^{\mathsf{M}}_{\{0,1\}}$ give the upper or lower half of the matrix M. The subsequent expansion of t_1 gives one then the individual quadrants of M. Boolean expansion for the first pair of variables $\mathsf{s}_1,\mathsf{t}_1$ thus yields:

$$f_{\mathsf{M}} = \mathsf{s}_1\mathsf{t}_1 f^{\mathsf{M}}_{11} + \mathsf{s}_1\neg\mathsf{t}_1 f^{\mathsf{M}}_{10} + \neg\mathsf{s}_1\mathsf{t}_1 f^{\mathsf{M}}_{01} + \neg\mathsf{s}_1\neg\mathsf{t}_1 f^{\mathsf{M}}_{00}.$$

The graph rooted in node representing $f^{\mathsf{M}}_{\vec{b}}$ is a symbolic representation of sub-matrix $M_{i,j}$, with $\mathcal{E}(i,j) = \vec{b}$. This access scheme can be applied recursively to each submatrix until one reaches the level of terminal nodes. For a (4×4) matrix M this would give us the matrix elements $m_{r,c}$ in the following sequence: $m_{0,0}, m_{0,1}, m_{1,0}, m_{1,1}$, which are the matrix entries of the upper left quadrant of M. The next elements to follow are $m_{0,2}, m_{0,3}, m_{1,2}, m_{1,3}$, which are the elements of the upper right quadrant, and so on.

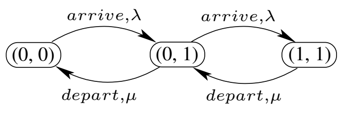
As an example, we refer to Fig. 2 (left) which shows a matrix C and its ZDD-based representation. In order to illustrate the block-wise addressing scheme realized by the chosen variable ordering, the matrix is shown as a table equipped with Boolean variables. The valuation $\neg\mathsf{s}_1\neg\mathsf{t}_1$, which corresponts to the diagonally hatched nodes of ZDD C, leads to the sub-ZDD rooted in node n_1 at level s_2, representing the sub-function $f^{\mathsf{C}}_{00}(\mathsf{s}_2,\mathsf{t}_2)$. This means that we extracted the upper left block-matrix $C(0,0)$, whose entries are given by the values of the terminal nodes reachable from n_1 (including the *0-entries* that we chose to ignore so far). A depth-first traversal with `else`-edge first delivers the sequence 0, 0, μ, 0 of matrix entries.

2.2.2 Extending ZDDs for efficiently computing matrix-vector products

The symbolic solvers as well as the ZDD-based implementation of the Moebius AFI considered in this paper employ an approach in which the generator matrix is represented by a symbolic data structure and the probability vectors are stored as arrays.

(a) Offset-labelling of ZDD-nodes: If n Boolean variables are used for state encoding, there are 2^n potential states, of which only a small fraction may be reachable. Allocating entries for unreachable states in the vectors would be a waste of memory space and would severely restrict the applicability of the algorithms (for instance, storing probabilities as doubles, a vector with about 134 million entries already requires 1 GByte of RAM). Therefore a dense enumeration scheme for the reachable states has to be implemented. This is achieved via the concept of offset-labelling, as had been first suggested for the MTBDD data structure by [25]. In an offset-labelled ZDD, each node is equipped with an offset value. While traversing the ZDD encoding the matrix, in order to extract a matrix entry, the row and column index in the dense enumeration scheme can be determined from the offsets, basically by adding the offsets of those nodes where the `then`-Edge is taken. In other words, the offsets are used to map the $\vec{\mathsf{s}}$ and $\vec{\mathsf{t}}$ vectors to a pair (r,c) of (densely enumerated) row and column indices.
As an example, one may refer to Fig. 2.iii. On the left the ZDD of Fig. 1.iii is depicted once again. At the right, one can see the offset-labelled variant of the same ZDD. This makes it possible to interpret the ZDD now as a (3×3) matrix, where in contrast to the matrix of Fig. 1.iv the third row and column are masked, since state 10 is not reachable.

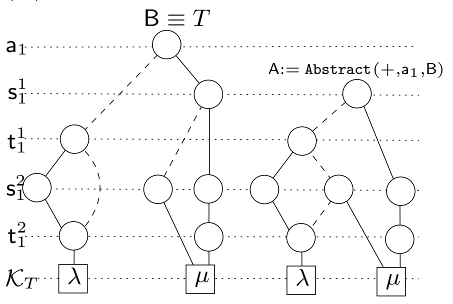
(i) Act.-labeles CTMC T



(ii) Binary encodings

$\vec{s} \xrightarrow{l,q} \vec{t} \in T$	a_1	s_1^1	t_1^1	s_1^2	t_1^2	f_B
$(0,0) \xrightarrow{arrive,\lambda} (0,1)$	0	0	0	0	1	λ
$(0,1) \xrightarrow{arrive,\lambda} (1,1)$		0	1	1	1	
$(0,1) \xrightarrow{depart,\mu} (0,0)$	1	0	0	1	0	μ
$(1,1) \xrightarrow{depart,\mu} (0,1)$		1	0	1	1	

(iii) ZDD-based representations



(iv) Matrix A

$$\begin{pmatrix} 0 & \lambda & 0 & 0 \\ \mu & 0 & 0 & \lambda \\ 0 & 0 & 0 & 0 \\ 0 & \mu & 0 & 0 \end{pmatrix}$$

Figure 1: *aCTMC*, its ZDD based representation and underlying transition rate matrix

As Fig. 2.iii also illustrates, within offset-labelled ZDDs isomorphic nodes are merged only if they also carry the same offset. Thus the diagonally hatched node within the original ZDD needs to be duplicated, once the offset-labelling is added.

(b) Block-structured hybrid offset-labelled ZDDs: The space efficiency of symbolic matrix representation comes at the cost of computational overhead, caused by the recursive traversal of the ZDD during access to the matrix entries. For that reason, [25] introduced the idea of replacing the lower levels of the MTBDD by explicit sparse matrix representations, which works particularly well for block-structured matrices. In the context of our work, we call the resulting data structure *hybrid offset-labelled* ZDD. The level at which one switches from symbolic representation to sparse matrix representation, called *sparse level*, depends on the available memory space, i.e. there is a typical time/space tradeoff.

The Gauss-Seidel method requires row or column-wise access to the matrix entries. Unfortunately, this cannot be realized efficiently with ZDD-based representations, if the interleaved variable ordering as described above is chosen. As a compromise, [25] developed the so-called pseudo-Gauss-Seidel (PGS) iteration scheme. Given a ZDD which represents the matrix, each inner node at a specific level corresponds to a block. Pointers to these nodes can be stored, which means that effectively the top levels or variables of the ZDD have been removed. The ZDD level at which the root nodes of the sub-matrices reside is called *block level*. When removing the upper b levels, one partitions the matrix into blocks, not necessarily of equal size, due to unreachable states. This allows one to access the blocks in a descending or ascending order and thus employ the Gauss-Seidel iteration scheme among the blocks, where within the blocks the Jacobi iteration scheme must be used.

Fig. 2.iv shows an example where the offset-labelled ZDD of Fig. 2.iii is block-structured. In order to achieve a correct indexing of the matrix entries, one not only has to store references to the root nodes of each block, but also the initial row and column offset. In terms of Fig. 2.iv, these pairs are depicted on top of the root nodes.

In total, block-structuring and the hybrid storage format yield a memory structure in which some levels from the top and some levels from the bottom of the ZDD have been replaced by sparse matrix structures. We call such a memory structure a block-structured hybrid offset-labelled ZDD. The choice of an adequate sparse level s and an adequate block level b is an optimization problem. In general, increasing b improves convergence of the PGS scheme, and replacing more ZDD levels by sparse structures improves speed of access.

2.3 Accessing the transition rate matrix via the AFI

Analogously to our proprietary ZDD-based solvers described in [19], our AFI implementation operates on offset-labelled (hybrid) ZDDs representing the transition rate matrix. However, contrary to the symbolic solvers, the AFI-based solvers do not have any knowledge about the symbolic data types employed, they simply iterate over the matrix in an element-wise, row-wise, column-wise or sub-matrix-wise fashion. The implementation of these different access patterns will be discussed in greater detail now.

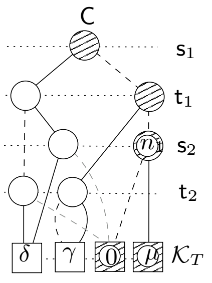
2.3.1 Iteration over all matrix entries

The *allEdges* container defined in the AFI returns all matrix entries in an arbitrary order, where the value and the pair of indices must be delivered. In case of ZDDs, this can be achieved by a hanging recursive depth-first search graph traversal. The matrix entry to be delivered by the iterator is given by the value of the currently visited terminal node, as well as the computed row and column offsets. In details the algorithm work as follows: Upon initialization, the iterator descends to the first non-0 terminal node by repetitively taking the *else*-edge if it does not lead to terminal 0-node or the *then*-edge otherwise. Every time the *else*-edge is chosen, the current node and the row and column offset values are pushed onto a stack so the *then*-successor can be proceeded later. Consequently, this stack implements the functionality of a program stack, such that the recursion can be split over the subsequently executed accesses. When the solver calls for the next matrix element via the ++-iterator or *next*-operator, the topmost element is removed from the stack and the next recursive step is performed until a non-zero terminal node is reached, where the traversed nodes are once again pushed onto the stack. An empty stack signals the end of the graph traversal, causing the iterators

(i) Matrix C

		$\neg$t$_1$		t$_1$	
		$\neg$t$_2$	t$_2$	$\neg$t$_2$	t$_2$
$\neg$s$_1$	$\neg$s$_2$	0	0	γ	γ
	s$_2$	μ	0	0	0
s$_1$	$\neg$s$_2$	0	δ	0	0
	s$_2$	0	0	δ	0

(ii) ZDD-based representation of matrix C



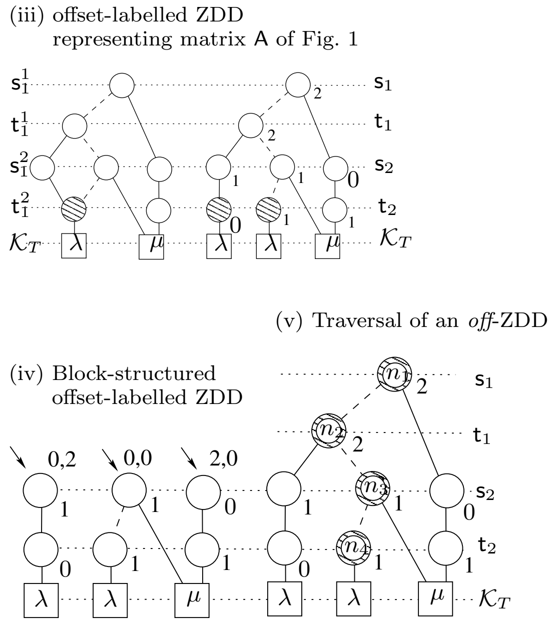

(iii) offset-labelled ZDD representing matrix A of Fig. 1
s$_1^1$
t$_1^1$
s$_1^2$
t$_1^2$
$\mathcal{K}_T$
s$_1$
t$_1$
s$_2$
t$_2$
λ
μ
(iv) Block-structured offset-labelled ZDD
0,2
0,0
2,0
(v) Traversal of an *off*-ZDD
n_1
n_2
n_3
n_4

Figure 2: Block-wise access (left), and extending ZDDs by offsets (right)

to set their *end*-flag, which may cause the solvers to start with the next iteration of the solution method. Since the hanging recursion follows always the *else*-edge first, the algorithm implementing the above functionality is denoted as `GetLeftChildFST`.

For exemplification one may refer to Fig. 2.v. At the first call `GetLeftChildFST` traverses the path with the diagonally hatched nodes. When leaving node n_4 via its *then*-edge, the routine pops it from the stack. Since in the next step a terminal node is reached `GetLeftChildFST` stops and returns $(\lambda, (0, 1))$ to the iterator calling function. Node n_3 is hereby left at the top of the stack, so that within the next step `GetLeftChildFST` can resume with traversing its sub-graph.

For hybrid ZDDs the above procedure needs to be slightly adapted: when reaching the sparse level, one starts iterating over the sparse matrices, thereby book-keeping row and column offsets accordingly.

2.3.2 *Access to matrix rows and columns*

As explained above, the interleaved ordering of variables encoding row- and column indices is a commonly accepted heuristics for obtaining small DD sizes. However, this comes with the drawback that graph traversal organized in a depth-first style does not visit the matrix elements in row or column order (see once again Fig. 2 (left)). For resolving the collision between ordering and access scheme, one may extract the currently required column or row-entries by executing a multiplication between the BDD Z_c representing the binary encoded column (or row) index and the ZDD Z_M representing the transition rate matrix. I.e. by computing $\mathsf{Z}_{tmp} := \mathsf{Z}_c \cdot \mathsf{Z}_M$ and subsequently abstracting over the variables encoding the row (or column) indices (`ZAbstract` $(\mathsf{Z}_{tmp}, \vec{s}, +)$) one obtains a symbolic representation of the column (or row) vector representing the respective matrix column (or row). The fact that employed DDs, i.e. especially the ZDD representing the transition rate matrix (Z_M), are offset-labelled, does not affect the operation, we only have to copy the offsets of the nodes of Z_M to the nodes in the resulting DD Z_{tmp}, so that its traversal yields the correct row and column indices of the dense enumeration as induced by the offset labelling scheme.

Encoding and constructing Z_c is hereby very efficient, whereas the multiple execution of $\mathsf{Z}_c \cdot \mathsf{Z}_M$ is computationally expensive. As an alternative one could think of generating and storing the ZDDs representing the different rows or columns separately. However, such a strategy is not feasible, it would significantly increase the number of ZDD nodes for storing the respective symbolic structures. But most importantly, the storage of the root nodes would induce a non-tolerable memory requirement. Therefore the extraction of row or column entries as illustrated above is executed on-the-fly, i.e. by means of a hanging recursive graph-traversal. This severely complicates the iterator code, since not only the zero-suppressing reduction rule must be taken into account, but also the branching is crucial, since one needs to traverse each path until it is evident that it does not encode the indices of the row or column currently accessed. In case this is known, the traversing algorithm performs a rollback to the last valid branching.

The column and row iterator as described above, do not currently support hybrid offset-labelled ZDDs nor the block-structured variant, but only pure offset-labelled ZDDs.

2.3.3 *Submatrix access to the matrix:*

Access to submatrix partitions is implemented with two different approaches:

(a) Block-structuring with variably sized blocks: The first implementation follows the block-partitioning approach as employed under the pseudo Gauss-Seidel method. Since each inner node of the ZDD Z_M represents a submatrix, one simply needs to remove the upper b levels, so that the root nodes of the block matrices all reside at the same level,

commonly denoted as *block level.* Due to the fact that the blocks may contain different numbers of reachable states, the blocks are in general neither quadratic nor of the same size. For compensating this problem, each root node must be furthermore equipped with an initial row and column offset. Arising from the different sizes of the blocks, the block-structured access to the matrix entries can only be used with solvers that make no special demands on the partitioning.

(b) Block-structuring with specifically sized blocks: In the attempt to allow arbitrary matrix partitioning, i.e. a partitioning where the blocks must be of a specific size, e.g. all sub-matrices need to be square matrices of the same size, a second implementation of submatrix access exists. This implementation mainly follows the idea of the row/column access, i.e. one intends to create a ZDD that encodes all states in the requested sub-matrix partition by multiplying the transition matrix Z_M with the symbolic encodings belonging to the current partition (of reachable states) $\mathbb{S}_j$. Let the set of reachable states be somehow partitioned ($\mathbb{S} = \mathbb{S}_1 \uplus \ldots \uplus \mathbb{S}_n$). Based on this partitioning, the respective block entries contained in the overall transition matrix Z_M can be extracted as follows:

$$\forall\, \mathbb{S}_j, \mathbb{S}_k \in \{\mathbb{S}_1, \ldots, \mathbb{S}_n\} :$$

$$\mathsf{Z}_M^{\mathbb{S}_{jk}} := \left(\sum_{\vec{x} \in \mathbb{S}_j} \mathsf{Z}_{\vec{\mathsf{s}} := \vec{x}} \right) \cdot \left(\sum_{\vec{y} \in \mathbb{S}_k} \mathsf{Z}_{\vec{\mathsf{t}} := \vec{y}} \right) \cdot \mathsf{Z}_M,$$

The remarks about memory and time consumption made in the context of the row/column-access scheme also apply in principle to this access scheme. However, the number of blocks is much smaller than the number of reachable states, thus a caching of the symbolically represented sub-matrix $\mathsf{Z}_M^{\mathbb{S}_{jk}}$ seems useful and will significantly speed-up the access times.

3. EMPIRICAL EVALUATION

We employed the following solution methods for benchmarking the implemented AFI-iterators: (a) Jacobi solution method (JAC) for accessing the matrix entries in an arbitrary order[3]. (b) Gauss-Seidel method (GS) for accessing the matrix entries in a row-wise manner. (c) Takahashi solution method for accessing the matrix entries in block-wise style. Before we give details about the collected runtime data, we briefly describe the software structure, employed benchmark models and platform on which the software was executed.

3.1 Preliminaries

3.1.1 Implementation

The software employed for benchmarking the ZDD-based AFI implementation consists of the following modules:

(a) The ZDD-based framework: This framework is implemented by us and incorporated into the Moebius modelling tool:

1. A symbolic engine for generating a ZDD-based representation of the high-level model's underlying (activity/reward-labelled) CTMC.

2. Proprietary ZDD-based hybrid solvers for computing steady state and transient state probabilities, which access the matrix element directly, i.e. not via the AFI. The run-time data of these solvers will be headed by the title *ZDD no AFI* in the following tables.

3. The ZDD-based implementation of Moebius AFI as described in the previous section. The run-time data of these ZDD and AFI-based solvers will be headed by the title *ZDD stand* (for standard) in the tables.
   For speeding up matrix access we also implemented an optimized version, where we removed all `s` and `t` variables residing above the sparse level yielding a nested sparse matrix structure. I.e. similar to the approach of [22], every ZDD-path from the root node to the sparse matrix structures is substituted by a pointer. Since the blocks are in general not of equal sizes, each pointer must be equipped with the appropriate row and column offsets. The run-time data of the optimized ZDD and AFI-based solvers will be headed by the title *ZDD opt* (for optimized). In contrast to [22], we make use of a linked list and not a sparse matrix storage scheme for administering the root nodes and initial offsets of the symbolic submatrix representations. This decreases memory requirements clearly, since blocks containing only zeros can be omitted. The major disadvantage of the optimized approach, no matter if one uses a sparse matrix format or a linked list, is the increase in memory consumption, since the pointers (and their pair of offsets) must be stored for each submatrix.

(b) Components of the Moebius performance evaluation tool:

1. Implementation of the AFI iterating over matrices stored in sparse-matrix format [12] and an implementation of iterators working on MxD-based representations of the transitions matrix [10].

2. The AFI-based numerical solvers for computing steady state and transient state probabilities as provided by the Moebius tool.

3.1.2 Benchmark models and platform

For evaluating our ZDD-based implementation of the AFI we employed two well-known benchmark models, namely the Kanban model (Kanban) [7] and the Flexible Manufacturing System model (FMS) [8]. Both models are parameterized by the initial number of tokens within dedicated places, where in the following these numbers are described by parameter N. Depending on N, column 2 and 3 of Tab. 1.B give the number of states (*states*) and the number of transitions among them (*trans*). The number of transitions is equal to the number of non-zero entries of the transition rate matrix, and the number of states indicates its dimension.
All benchmarking experiments were executed on Pentium 4 systems with a Linux OS, where we employed either 3 GHz or 2.88 GHz machines. I.e. the figures of Tab. 1 were produced on a P4 with 3.0 GHz and 1 GByte of RAM, whereas the figures of all other tables were produced on a P4 with 2.88 GHz and a maximum of 3 GByte of RAM.

3.2 Jacobi solution method (JAC)

Tab. 1 shows the iteration time and memory consumption when employing Moebius' JAC-solver in combination with

[3]For access in arbitrary order, one could also employ the Power method for computing steady-state probabilities or the uniformization method for computing transient probabilities.

(A) CPU time consumed per iteration (in seconds)

(a) Flexible Manufacturing System (FMS)

			ZDD opt.		ZDD stand		ZDD
N	sparse	MxD	$s := 0.0$	$s := 0.66$	$s := 0.0$	$s := 0.66$	no AFI
6	0.2400	0.2548	0.6068	0.2120	0.7760	0.2560	0.0560
8	2.1889	2.2521	5.5464	1.8801	7.7265	2.4522	0.5184
10	???	14.2405	34.3484	11.6827	47.0149	13.9949	3.2734

Time per iteration in seconds

N	sparse	MxD	ZDD opt. $s := 0.0$	ZDD opt. $s := 0.66$	ZDD stand $s := 0.0$	ZDD stand $s := 0.66$
6	4.2857	4.5500	10.8357	3.7857	13.8570	4.5714
8	4.2222	4.3441	10.6984	3.6265	14.9036	4.7299
10	???	4.3504	10.4932	3.5690	14.3627	4.2753

Ratios, normed to ZDD (no AFI)

(b) Kanban Manufacturing System (Kanban)

N	sparse	MxD	ZDD opt. $s := 0.0$	ZDD opt. $s := 0.66$	ZDD stand $s := 0.0$	ZDD stand $s := 0.66$	ZDD no AFI
5	0.8213	2.3669	2.9758	1.1713	4.0815	1.5433	0.2884
6	???	9.9166	14.2427	5.5343	20.5173	7.4737	1.3533
7	???	37.3503	56.1271	21.0813	78.6449	27.8845	5.4195

Time per iteration in seconds

N	sparse	MxD	ZDD opt. $s := 0.0$	ZDD opt. $s := 0.66$	ZDD stand $s := 0.0$	ZDD stand $s := 0.66$
6	2.8474	8.2067	10.3178	4.0610	14.1512	5.3509
8	???	7.3278	10.5246	4.0896	15.1611	5.5226
10	???	6.8918	10.3564	3.8899	14.5114	5.1452

Ratios, normed to ZDD (no AFI)

(B) Memory space consumed (in MByte)

(a) FMS

						ZDD	
N	$\#states$	$\#trans$	iter vecs.	sparse	MxD	$s := 0.66$	$s := 0.33$
6	$0.54E6$	$4.21E7$	8.21	87	16	23	20
8	$4.46E6$	$3.85E7$	68.05	760	109	135	116
10	$2.54E7$	$2.35E8$	387.54	???	595	679	606

(b) Kanban

N	$\#states$	$\#trans$	iter vecs.	sparse	MxD	ZDD $s := 0.66$	ZDD $s := 0.33$
5	$2.55E6$	$2.45E7$	38.86	447	70	73	66
6	$1.13E7$	$1.16E8$	171.83	???	284	294	272
7	$4.16E7$	$4.50E8$	635.45	???	1008	1044	979

Table 1: Times per iteration for the AFI-based implementation of the JAC method

different data types and their different AFI-based implementation for iterating over all matrix entries in an arbitrary order. The upper tables of Tab. 1.A give the plain iteration times in seconds, whereas the lower tables contain ratios by norming the AFI-based iteration times to the iteration times of the proprietary (hybrid) symbolic solver. The positions filled with ??? refer to cases, where a solution could not be computed due to memory restrictions. Col. *sparse* contains the iteration time when the AFI in combination with a sparse matrix format was employed [12] and column *MxD* contains the figures as produced by the MxD-based AFI implementation as described in [10]. Concerning a ZDD-based representation of transition rate matrices one needs to distinguish between the optimized version (ZDD opt) and the standard version (ZDD stand). In both cases we varied the sparse level, so that either 0% or 66% of the ZDD was converted into sparse matrix format. In the last column of Tab. 1.A the iteration times of the proprietary ZDD-based JAC-solvers are given, where we also converted 66% of the offset-labelled ZDD into sparse matrix format. As one can obtain from the lower tables (the ones containing the ratios) the use of ZDDs in combination with the AFI imposes a severe run-time overhead. However, when also employing hybrid ZDDs this overhead can be reduced clearly, but nevertheless the imposed runtime overhead is not ignorable when it comes to practice, e.g. for 1,415 iterations required for computing steady state for the FMS ($N := 10$) model one consumes 4.59 h CPU time with the optimized ZDD-version ($s := 0.66$), whereas the proprietary ZDD JAC-solvers only requires 1.29 h. The overhead is not really surprising, since profiling reveals that 35% of the CPU time is spent for routine `GetLeftChildFST`, 22% for the routine executing each numerical iteration step (`perform_iter()`) and 16% for the ++-operator, which calls `GetLeftChildFST`. The alternate execution of these methods, which are implemented either within the AFI-based solver or within the state-level object encapsulating the transition matrix, imposes additional load on the operating system, which in fact clearly reduces the performance of the solver.

Tab. 1.B shows the overall memory consumption of the different solvers, including the memory consumption of the two probability vectors as shown in column 4 (iter vecs.), which in case of the symbolic matrix representations (MxD or ZDD-based) is the most resource consuming part. By comparing the memory requirements of the ZDD-based solvers (AFI-based and proprietary), we found out, that the additional memory overhead of the AFI is negligible. This is of course obvious if one keeps in mind that probability vectors and sparse matrices of the hybrid ZDDs are the most memory consuming parts. For investigating this effect, we therefore lowered the number of levels converted into sparse matrix format to 33%, which reduces the memory consumption as shown in the last two columns of Tab. 1.B.

3.3 Gauss-Seidel solution method (GS)

Tab. 2 shows the runtime data as collected when solving the FMS and Kanban model with the Gauss-Seidel method. It is important to note that in case of the proprietary ZDD-based solvers the *pseudo Gauss-Seidel method*, as described in the previous section, was employed, where the upper 50% of the DD-levels was replaced. The positions within Tab. 2 filled with xxx refer to cases, where the computation of a solution was not feasible due to time restrictions. As expected, the row-wise access to the matrix elements yields in case of ZDDs a non-tolerable run-time overhead, which makes a dynamic row-extraction useless in practice. This might also be the reason, why – as far as we know – the MxD-based version of row-wise access was never implemented.

The disappointing runtimes in case of the ZDD-based AFI as given in Tab. 2.A and B mostly stem from the excessive calls to the routine `AFIApplyMult`(), which extracts the different rows (or columns) from the matrix and annotates the nodes of the resulting ZDD with the correct offsets. But besides the bad runtimes in case of the ZDD-based AFI, Tab. 2.A and B also indicate the overhead imposed by the AFI when standard sparse matrix technology is employed. As one can read from the tables, the AFI increases the CPU times by a factor of approximately 3. Similar to what we observed with the JAC-solver, such an increase is not dramatic, but from a practical point of view severely reduces the applicability of the AFI.

In contrast to the *allEdges*-iterator, the row-iterator as employed by the GS-based solver does currently not make use of block-structured and/or hybrid offset-labelled ZDDs (it uses pure offset-labelled ZDDs). Implementing these features would improve CPU time consumption to a certain extent, but it would certainly not reduce the runtime by two orders of magnitude.

In contrast to the JAC method, the (pure) GS method does not require the use of an additional iteration vector, which significantly reduces the memory requirement of the method. Thus, if memory limitation is not an issue, one may think of pre-generating symbolic representations for each row, so that the computationally expensive generation during each numerical iteration is avoided. Alternatively, one may also think of replacing the interleaved ordering scheme by a sequential order, where the variables encoding the rows come first and the variables encoding the column thereafter. However, we did not implement and investigate such strategies, since the ZDD-based PGS-method already delivers highly competitive results.

3.4 Takahashi solution method

The method of Takahashi is an iterative aggregation/disaggregation method for computing the steady-state probability vector of a Markov chain [32, 31]. The state space is partitioned into K blocks, and in each iteration an aggregated system of size K is constructed based on the current approximation of the solution vector. Each outer iteration

N	sparse matrix		ZDD		ratio
	t_{part}	$t_{iter.}$	t_{part}	$t_{iter.}$	
(A) Flexible Manufacturing System					
6	0.8121	1.5952	157.3498	14.4299	0.1106
8	10.2246	16.5459	1,787.4957	293.2779	0.0564
(B) Kanban System					
5	3.2002	20.0452	577.8441	74.5958	0.2687

Table 3: CPU times when employing the Takahashi solution method

of the Takahashi algorithm includes the solution of the aggregated system and the solution of the K systems corresponding to the individual blocks, i.e. in one iteration $K+1$ smaller sized systems must be solved in order to obtain a new approximation of the overall solution.

In the implementation under study, the overall matrix is partitioned into blocks of predefined size, whereby for each block a separate ZDD is constructed with the help of the AFI routine `getSubMatrixPartition`. During iteration, the individual blocks are accessed through routine `getSubMatrix`. Tab. 3 shows the timing results for the Takahashi method, where the AFI is employed for accessing the matrix blocks represented as sparse matrices (columns 2 and 3) and as ZDDs (columns 4 and 5). The table lists the times for partitioning the matrix into blocks, and the times for one outer Takahashi iteration. The last column ("ratio") has now a different meaning than in the previous tables, it denotes the ratio between the sparse matrix iteration time and the ZDD iteration time. Looking at both, the absolute times and the ratios, it is obvious that the partitioning into blocks of predefined size, which in general do not correspond to ZDD subgraphs, causes an immense overhead. It should be pointed out that, in the current implementation, the blocks are represented by pure (i.e. non-hybrid) ZDDs. The iteration times could be improved to some extent by replacing the lower parts of these block-ZDDs by sparse matrices, thereby speeding up the access to the matrix elements.

4. CONCLUSION

This paper investigated the pros and cons of a state-level abstract functional interface (AFI) in the context of Markovian performability modelling. Such an interface can be used for accessing the entries of a matrix represented by a symbolic data structure. It separates numerical solution methods from the underlying data structure used for matrix representation: A given numerical method accesses matrix elements through the AFI and therefore does not need to know any details of the data structure.

The Moebius state-level AFI had previously been implemented (at least partially) for the sparse matrix, Kronecker and MxD data structures. The present paper described a complete implementation for state-level objects in the ZDD format. We conducted an empirical assessment which basically resulted in the two following findings:

1. Even for access pattern which match very well with the state-level object, the AFI may pose a significant runtime overhead.

2. Numerical algorithms which require a particular access pattern to the matrix entries will always work through

	sparse				ZDD
N	no AFI	AFI	ZDD opt	ZDD stand	no AFI
(A) FMS					
6	0.0556	0.1608	74.8367	133.7204	0.1368
8	0.5080	1.4925	832.0120	1,512.2945	0.8693
10	???	???	xxx	xxx	5.0279
	Time per iteration in seconds				
6	0.4064	1.1754	977.4237	977.4237	
8	0.5844	1.7170	1,739.7614	1,739.7614	
10	???	???	xxx	xxx	
	Ratios, normed to ZDD (no AFI)				
(B) Kanban					
5	0.2545	0.7360	298.6627	715.8367	0.3648
6	???	???	1,406.4719	3,633.2871	1.7157
7	???	???	xxx	xxx	6.5536
	Time per iteration in seconds				
5	0.6976	2.0175	818.6509	1,962.1480	
6	???	???	819.7623	2,117.6617	
7	???	???	xxx	xxx	
	Ratios, normed to ZDD (no AFI)				

Table 2: Times per iteration when employing the GS method

the AFI, but they will not achieve high performance, unless this pattern conforms with the underlying data structure.

Overall, an AFI makes it easier to introduce new storage formats for the matrices while reusing existing solution methods. However, it is clear that numerical algorithms will not achieve high performance unless they are somewhat tailored to the underlying data structure.

Acknowledgements: The authors would like to cordially thank Bill Sanders and the Moebius developers group from the University of Illinois for making Moebius available to us and for several fruitful discussions.

5. REFERENCES

[1] P. Bazan and R. German. Approximate Analysis of Stochastic Models by Self-Correcting Aggregation. In *2nd Int. Conf. on Quantitative Evaluation of Systems (QEST'05)*, pages 134–144. IEEE Comp. Soc., 2005.

[2] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[3] P. Buchholz. Numerical Solution Methods Based on Structured Descriptions of Markovian Models. In G. Balbo and G. Serazzi, editors, *Proc. 5th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 242–258. Elsevier Science Publisher B.V., 1992.

[4] P. Buchholz. Structured Analysis Techniques for Large Markov Chains. In *Proc. 1st Workshop on Tools for Solving Structured Markov Chains*, Pisa, 2006. ACM Press, CD Edition.

[5] P. Buchholz and P. Kemper. Kronecker Based Matrix Representations for Large Markov Models. In C. Baier, B. Haverkort, H. Hermanns, J.P. Katoen, and M. Siegle, editors, *Validation of Stochastic Systems – A Guide to Current Research*, pages 256–295. Springer, LNCS 2925, 2004.

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

[7] G. Ciardo and M. Tilgner. On the use of Kronecker operators for the solution of generalized stochastic Petri nets. Technical Report 96-35, Institute for Computer Applications in Science and Engineering, 1996.

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

[9] D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. Doyle, W.H. Sanders, and P. Webster. The Moebius Framework and Its Implementation. *IEEE Transactions on Software Engineering*, 28(10):956–969, 2002.

[10] S. Derisavi. The Moebius State-level Abstract Functional Interface, 2005. Master Thesis. University of Illinois at Urbana-Champaign (IL, USA).

[11] S. Derisavi. A Symbolic Algorithm for Optimal Markov Chain Lumping. In O. Grumberg and M. Huth, editors, *TACAS 2007*, pages 139–154. Springer, LNCS 4424, 2007.

[12] S. Derisavi, T. Courtney, P. Kemper, and W. H. Sanders. The Moebius State-level Abstract Functional Interface. In *Proc. of Performance Tools 2002: 12th Int. Conf. on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation*, pages 31–50, 2002.

[13] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. *Distributed Computing*, 6(3):155–164, 1993.

[14] M. Fujita and P. McGeer, editors. *Formal Methods in System Design: Special Issue on Multi-terminal Binary Decision Diagrams*, 1997. Vol. 10, No. 2/3.

[15] B.R. Haverkort, A. Bell, and H. Bohnenkamp. On the

Efficient Sequential and Distributed Generation of very Large Markov Chains from Stochastic Petri Nets. In *Proc. of IEEE Petri Nets and Performance Models*, pages 12–21, 1999.

[16] H. Hermanns and M. Ribaudo. Exploiting Symmetries in Stochastic Process Algebras. In *Simulation-Past, Present and Future. 12th European Simulation Multiconference*, pages 763–770. SCS International, June 1998.

[17] W.J. Knottenbelt. *Parallel Performance Analysis of Large Markov Models*. PhD thesis, University of London, Imperial College, Dept. of Computing, 1999.

[18] K. Lampka and M. Siegle. Activity-Local State Graph Generation for High-Level Stochastic Models. In *Measuring, Modelling, and Evaluation of Systems 2006*, pages 245–264. VDE-Verlag, April 2006.

[19] K. Lampka and M. Siegle. Analysis of Markov Reward Models using Zero-supressed Multi-terminal decision diagramms. In *Proceedings of VALUETOOLS 2006 (CD-edition)*, October 2006.

[20] K. Lampka, M. Siegle, J. Ossowski, and C. Baier. Zero-Suppressed Multi-Terminal BDDs: Concept, Algorithms and Applications. Manuscript in preparation.

[21] K. Lampka, M. Siegle, and M. Walter. An easy-to-use, efficient tool-chain to analyze the availability of telecommunication equipment. In *Proc. Formal Methods on Industrial Critical Systems 2006*, LNCS 4346, pages 35–50, 2006.

[22] R. Mehmood. *Disk-based techniques for efficient solution of large Markov chains*. PhD thesis, University of Birmingham, University of Birmingham (U.K.), October 2004.

[23] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proc. 30th Design Automation Conference (DAC)*, pages 272–277, Dallas (Texas), USA, June 1993. ACM / IEEE.

[24] A. Miner and D. Parker. Symbolic Representations and Analysis of Large State Spaces. In *Validation of Stochastic Systems*, LNCS 2925, pages 296–338. Springer, 2004.

[25] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, Birmingham (U.K.), 2002.

[26] Brigitte Plateau. On the stochastic structure of parallelism and synchronization models for distributed algorithms. In *Proc. SIGMETRICS'85*, pages 147–154. ACM Press, 1985.

[27] 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.

[28] M. Siegle. *Beschreibung und Analyse von Markovmodellen mit großem Zustandsraum*. PhD thesis, Friedrich-Alexander-Universität Erlangen–Nürnberg, Erlangen (Germany), 1995.

[29] M. Siegle. Advances in model representation. In *Proc. of the Joint Int. Workshop PAPM-PROBMIV 2001*, LNCS 2165, pages 1–22. Springer, September 2001.

[30] M. Siegle. *Behaviour analysis of communication systems: Compositional modelling, compact representation and analysis of performability properties*. Shaker Verlag, Aachen, 2002.

[31] W.J. Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994.

[32] Y. Takahashi. A Lumping Method for Numerical Calculation of Stationary Distributions of Markov Chains. Technical Report B-18, Tokio Institute of Technology, Dpt. of Information Sciences, June 1975.