Analysis of Markov Reward Models using Zero-suppressed Multi-terminal BDDs

K. Lampka and M. Siegle
University of the Federal Armed Forces Munich, Institute for Computer Engineering
{kai.lampka,markus.siegle}@unibw.de

ABSTRACT

High-level stochastic description methods such as stochastic Petri nets, stochastic UML statecharts etc., together with specifications of performance variables (PVs), enable a compact description of systems and quantitative measures of interest. The underlying Markov reward models (MRMs) often exhibit a significant blow-up in size, commonly known as the state space explosion problem. In this paper we employ our recently developed type of symbolic data structure, *zero-suppressed multi-terminal binary decision diagram* (ZDD). In addition to earlier work [12] the following innovations are introduced: (a) new algorithms for efficiently generating ZDD-based representation of user-defined PVs, (b) a new ZDD-based variant of the approach of [17] for computing state probabilities, and (c) a new ZDD-based algorithm for computing moments of the PVs. These contributions yield a ZDD-based framework which allows the computation of complex performance and reliability measures of high-level system specifications, whose underlying MRMs consist of more than 10^8 states. The proposed algorithms for generating user-defined PVs and computing their moments are independent of the employed symbolic data type. Thus they are highly suited to fit into other symbolic frameworks as realized in popular performance evaluation tools. The efficiency of the presented approach, which we incorporated into the Möbius modeling framework [16], is demonstrated by analyzing several benchmark models from the literature and comparing the obtained run-time data to other techniques.

Keywords

Discrete Event Systems, Markov Chain, Numerical Solution, Symbolic Data Structure, Performance Evaluation Tool

1. INTRODUCTION

(A) Motivation: High-level stochastic model description methods, such as stochastic Petri nets (SPN), stochastic UML statecharts or stochastic process algebras, etc., have shown to be powerful tools for describing and analyzing distributed hardware and software systems. Performance variables (PV) enable the modeler to define complex performance and reliability (performability) measures on the level of the high-level model description, rather than on the level of its semantic model. A high-level model description together with its user-defined PVs can be mapped to a continuous time Markov chain (CTMC) and a set of rate and impulse rewards for each state and / or transition, such that one obtains a (low-level) Markov reward model (MRM). The numerical solution of the latter allows one to determine complex performability measures for the system under study. However, the interleaving semantics of standard high-level model description methods often leads to an exponential blow-up in the number of states of the low-level MRM (*state space explosion*), where standard approaches require the explicit evaluation of the user-defined PVs for each state. This hampers the analysis of complex and large systems, if not making it impossible. In this paper, we present a framework for the analysis of very large MRMs using a new type of symbolic data structure, called *zero-suppressed multi-terminal binary decision diagram* (ZDD). Our framework allows us to solve MRMs consisting of more than 10^8 states on a commodity PC and to efficiently compute performability measures of interest. Since this framework is independent of the employed symbolic data type, as long as algorithms for its efficient manipulation exist, it is also highly suited to fit into other symbolic performance evaluation tools such as Prism [18], Caspa [10] and Smart [21], to name only a few of them.

(B) Contributions and related work: In addition to earlier work [11, 12], where we discussed the efficient construction of symbolically represented CTMCs, we present here new algorithms for generating and solving symbolically represented MRMs. In contrast to standard techniques, these algorithms for generating symbolic representations of user-defined PVs exploit locality, such that the explicit evaluation of reward functions can be limited to fractions of states of the MRM.
Reduced ordered Binary Decision Diagrams (BDDs) [2, 3] are state-of-the-art when it comes to state-based system verification. In the context of stochastic modeling, the most prominent decision diagrams (DDs) are *multi-terminal* or *algebraic BDDs* (ADDs) [1], *multi-valued decision diagrams* [9] and *matrix diagrams* [15]. ZDDs, which are the multi-terminal extensions of zero-suppressed BDDs [14] and which we introduced in [12], are employed here for the first time for representing user-defined PVs, computing the state proba-

bilities of the MRM and finally for computing moments of the PVs.
Our new ZDD-based solvers for computing state probabilities are based on the hybrid solution method developed in [17] for ADDs, where details on the implementation can be found in [24, 7]. Based on the computed state probabilities, as well as the ZDD-based representations of user-defined PVs, we finally introduce a new ZDD-based algorithm, which allows the efficient computation of their moments, giving one the performability measures of the system under study. The discussion is limited to the computation of mean and variance of instant-of-time PVs [19], where an extension to (time-averaged) interval-of-time PVs is straight forward. For simplification we will restrict ourselves to the handling of pure Markovian models. A slightly extended scheme can handle high-level models containing not only Markovian activities, but also prioritized immediate ones.

(C) Organization: The paper is organized as follows: Sec. 2 introduces the model world and provides basic definitions. Sec. 3 introduces ZDDs and the general idea of employing them for the compact representation of activity-labeled CTMCs. Sec. 4 introduces the ZDD-based scheme for efficiently generating a symbolic representation of a low-level MRM. Based on the ZDD-based representations this section also introduces the basic idea of the new ZDD-based numerical solvers, as well as a new algorithm for efficiently computing moments of user-defined PVs. A detailed empirical evaluation of the ZDD-based framework, which we implemented within the Möbius modeling framework [16], is presented in Sec. 5. Sec. 6 concludes the paper by summarizing the achieved innovations and mentioning future steps.

2. MODEL WORLD

(A) Static properties: A high-level model M consists of a finite ordered set of discrete state variables (SVs) $s_i \in \mathcal{S}$, where each can take values from a finite subset of the naturals. Each state of the model is thus given as a vector $\vec{s} \in \mathbb{S} \subset \mathbb{N}^{|S|}$. Concerning the high-level model description methods, the current value of a SV may describe the number of tokens in a place, the current state of a process, or the value of a process parameter. A model has a finite set of activities ($\mathcal{A}ct$). Analogously to the Petri Net based model description methods, SVs and activities are assumed to be connected through a connection relation $\mathcal{C}on \subseteq (S \times \mathcal{A}ct) \cup (\mathcal{A}ct \times S)$, such that the enabling and the execution of an activity l depends on a set of SVs:

$$\mathcal{S}_l^{\mathcal{D}} := \{s_i \in S \mid (s_i, l) \in \mathcal{C}on \vee (l, s_i) \in \mathcal{C}on\}, \qquad (1)$$

where $\mathcal{S}_l^{\mathcal{I}} = S \setminus \mathcal{S}_l^{\mathcal{D}}$. Two activities are defined to be dependent if their sets of dep. SVs are not disjoint. We also define a projection function

$$\chi\colon\ (\mathcal{S}_l^{\mathcal{D}}, \mathbb{N}^{|S|}) \longrightarrow \mathbb{N}^{|\mathcal{S}_l^{\mathcal{D}}|} \qquad (2)$$

which yields the sub-vector consisting of the dependent SVs only. We use the shorthand notation $\vec{s}_{d_l} := \chi(\mathcal{S}_l^{\mathcal{D}}, \vec{s})$, where $\vec{s}_{d_l}$ is called the activity-local marking of state $\vec{s}$ with respect to activity l.

(B) Dynamic properties: When an activity is executed, the model evolves from one state to another. For each activity $l \in \mathcal{A}ct$ we have a transition function $\delta_l : \mathbb{S} \longrightarrow \mathbb{S}$, whose specific implementation depends on the model description method. Concerning the target state of a transition, we use the superscript of a state descriptor to indicate the sequence of activities leading to that state. I.e. for an activity execution sequence $\omega := (\omega_1, \ldots, \omega_{|\omega|}) \in \mathcal{A}ct^*$ we write $\vec{s}^{\,\omega} := \delta_{\omega_{|\omega|}}(\ldots \delta_{\omega_2}(\delta_{\omega_1}(\vec{s}, \omega_1), \omega_2), \ldots, \omega_{|\omega|})$. The set of all activities enabled in a state $\vec{s}$ will be denoted as $Enabled_{\vec{s}}$. For each activity $l \in \mathcal{A}ct$ we also define a rate function $\eta_l : \mathbb{S} \times \mathbb{S} \longrightarrow \mathbb{R}^{\geq 0}$, which yields the rate at which the model moves from source to target state under activity l. Hereby it is assumed, that the computation of δ_l and η_l depends solely on those positions of $\vec{s}$ referring to the SVs contained in $\mathcal{S}_l^{\mathcal{D}}$. By state graph (SG) exploration one can construct the successor-state relation as a set of quadruples $T \subseteq (\mathbb{S} \times \mathcal{A}ct \times \mathbb{R}^{>0} \times \mathbb{S})$, which is the set of transitions of a stochastic labeled transition system (SLTS), i.e. the underlying activity-labeled CTMC. If activity labels are removed, transitions between the same pair of states are aggregated via summation of the individual rates.

(C) Performance variables: PVs enable the modeler to define complex performability measures on the basis of the high-level model, rather than on the level of the underlying CTMC [19]. A performance variable consists of a rate reward and/or an impulse reward definition. A rate reward defines the reward gained by the model in a specific state. In contrast, an impulse reward defines the reward as obtained by completing the execution of a specific activity in a specific state. This gives us the following setting:

1. A rate reward r defined on a high-level model is specified by the rate reward returning function $\mathcal{R}_r : \mathbb{S} \to \mathbb{R}^{\geq 0}$, and where $\mathcal{S}_r^{\mathcal{D}} \subseteq \mathcal{S}$ is the set of SVs on which the computation of r actually depends. Analogously to activity-local markings we will also employ the shorthand notation $\vec{s}_{d_r} := \chi(\mathcal{S}_r^{\mathcal{D}}, \vec{s}\,)$. The set of all rate rewards defined for a given high-level model will be denoted as $\mathcal{R}$.

2. An impulse reward i is received each time an activity k from the impulse reward's set of activities $\mathcal{A}ct_i$ is executed, where the reward may also be state-dependent, yielding $\mathcal{I}_k^i : \mathbb{S} \to \mathbb{R}^{\geq 0}$. This allows us to define the impulse reward returning function $\mathcal{I}^i : \mathbb{S} \to \mathbb{R}^{\geq 0}$, for impulse reward i as follows:

$$\mathcal{I}^i(\vec{s}\,) := \sum_{k \in \mathcal{A}ct_i \cap Enabled_{\vec{s}}} \mathcal{I}_k^i(\vec{s}) \cdot \eta_k(\vec{s}, \vec{s}^{\,k})$$

Hereby we restrict the computation of $\mathcal{I}_k^i$ to those positions of $\vec{s}$ referring to SVs of $\mathcal{S}_k^{\mathcal{D}}$. The set of all impulse rewards defined for a given high-level model will be denoted as $\mathcal{I}$.

3. SYMBOLICALLY REPRESENTING CTMCS

In the following, we briefly explain how ZDDs can be employed for representing activity-labeled CTMCs.

3.1 Zero-suppressed multi-terminal BDDs

A Binary Decision Tree (BDT) is a binary tree $\mathsf{B} := \{\mathcal{V}, \mathcal{K}, \texttt{value}, \texttt{var}, \texttt{then}, \texttt{else}\}$, where:

1. $\mathcal{V}$ is a finite set of Boolean variables,
2. $\mathcal{K} = \mathcal{K}_T \cup \mathcal{K}_{NT}$ is a finite non-empty set of nodes, consisting of the disjoint sets of terminal nodes $\mathcal{K}_T$ and non-terminal nodes $\mathcal{K}_{NT}$,
3. and where the following functions are defined:
   (a) $\texttt{value} : \mathcal{K}_T \mapsto \mathbb{B}$ with $\mathbb{B} := \{0, 1\}$,
   (b) $\texttt{var} : \mathcal{K}_{NT} \mapsto \mathcal{V}$,
   (c) $\texttt{else}, \texttt{then} : \mathcal{K}_{NT} \mapsto \mathcal{K}$, and
   (d) $\texttt{getRoot} : \mathsf{B} \mapsto \mathcal{K}$ for extracting the dedicated root node.

A BDD is a modified BDT, such that: (a) on all paths from the root to a terminal node the Boolean variables obey a fixed ordering, which allows to organize BDDs such that each level is associated with a specific Boolean variable. (b) Isomorphic subgraphs are merged: Within a shared BDD-environment this means, that each BDD node within the different but shared graphs represents a unique function. (c) Non-terminal nodes whose 1- and 0-successors are identical (commonly denoted as don't care nodes) are eliminated. A BDD [2, 3] over $< \mathcal{V}, \pi >$ is known to be a canonical representation of Boolean function.
If one shifts the range of the function represented by a BDD from $\mathbb{B}$ to $\mathbb{D}$, where $\mathbb{D}$ is a finite set, one ends up with multi terminal BDDs, also called algebraic BDDs (ADDs) [1]. DDs of this type are known to be canonical representations of (pseudo-Boolean) functions $f_{\mathsf{M}} : \mathbb{B}^n \mapsto \mathbb{D}$.

In zero-suppressed BDDs (z-BDDs) [14], instead of eliminating don't-care nodes, one eliminates those non-terminal nodes whose 1-successor is the terminal 0-node. Analogously to ADDs, we allow z-BDDs to have more than two terminal nodes and obtain ZDDs for representing pseudo-Boolean functions. Since the ideas developed in the following also apply to ADDs, we will often generically speak of Mt-DDs and only refer to the specific variant (ZDD or ADD) if necessary. Standard arithmetic operators can be applied to Mt-DDs efficiently with the help of Bryant's [3] $\texttt{Apply}$-algorithm or variants thereof. Since for a ZDD it is important to know the set of Boolean variables on which it depends, we were forced to extend each decision diagram by the set of variables on which it depends. Furthermore, this required a new $\texttt{Apply}$-algorithm for efficiently manipulating *partially shared* ZDDs, denoted as $p\texttt{ZApply}$-algorithm. Hereby we define ZDDs to be partially shared if they do not necessarily have identical sets of Boolean variables, leading to different semantics of skipped levels while traversing the ZDDs. We also implemented the operation $\texttt{Restrict}(\mathsf{Z}, \mathsf{v}, b)$ which restricts the ZDD Z to those paths where the variable v takes the value $b \in \{0, 1\}$. Furthermore, we implemented the operation $\texttt{Abstract}(\mathsf{Z}, \mathsf{v}, \texttt{op})$, which gives $\texttt{Restrict}(\mathsf{Z}, \mathsf{v}, 0)$ $\underline{\texttt{op}}$ $\texttt{Restrict}(\mathsf{Z}, \mathsf{v}, 1)$. For converting a ZDD Z to a z-BDD $\widetilde{\mathsf{Z}}$ by replacing all non-zero terminal nodes with the terminal one-node, we employ the function $\mathsf{ZDD2zBDD}()$.

3.2 ZDD-based representation of CTMCs

Each transition within an activity-labeled CTMC T is encoded by applying a binary encoding scheme which represents the transition $(\vec{s} \xrightarrow{l,\lambda} \vec{s}^{\,l})$ as the bit-vector: $(\mathcal{E}_{\mathcal{A}ct}(l)$, $\mathcal{E}_S(\vec{s}\,)$, $\mathcal{E}_S(\vec{s}^{\,l}))$. The rate λ is hereby unaccounted, since it will be stored in a terminal node of the Mt-DD. The individual bit positions of the obtained vectors correspond to the Boolean variables of the Mt-DD. Hereby we use the vectors $\vec{\mathsf{a}}$, $\vec{\mathsf{s}}$ and $\vec{\mathsf{t}}$ of Boolean variables, such that $\vec{\mathsf{a}}$ holds the encodings of activity labels (e.g. l), $\vec{\mathsf{s}}$ holds the encodings of the source states (e.g. $\vec{s}$), and $\vec{\mathsf{t}}$ holds the encodings of the target states (e.g. $\vec{s}^{\,l}$) of the elements of T [20]. In the sequel we assume that the Mt-DD variables are ordered in the following way: $\mathsf{a}_1 \prec \ldots \prec \mathsf{a}_{n_{\mathcal{A}ct}} \prec \mathsf{s}_1 \prec \mathsf{t}_1 \prec \ldots \prec \mathsf{s}_n \prec \mathsf{t}_n$. I.e. at the first $n_{\mathcal{A}ct}$ levels from the root are the variables a_i encoding the activity labels, and on the remaining $2n$ levels we have the variables s_i and t_i encoding source and target states of each transition in an interleaved fashion. Such an interleaved ordering of source and target bits is a commonly accepted heuristics for obtaining small BDD sizes [6] which also works well for ZDDs. For convenience we will use the somewhat sloppy notation $\mathsf{Z} := \mathcal{E}(\vec{s}\,)$ to denote, that the symbolic encoding of a certain state $\vec{s}$ is assigned to ZDD Z. The notation $\mathsf{Z} \setminus \mathsf{Z}'$ states, that the set of states represented by ZDD Z' is removed from the set of states represented by ZDD Z. The notation $\mathsf{Z}_l \xleftarrow{\vec{s}} \mathsf{Z}_U$ expresses that $\mathsf{Z}_l := \mathcal{E}(\vec{s})$, where $\vec{s}$ is an arbitrary state as contained in the set of states represented by ZDD Z_U.

3.3 Example

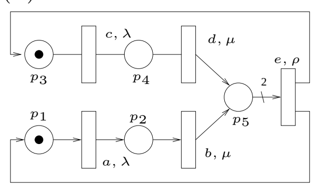
Part (A) and (B) of Fig. 1.I show a simple SPN and its underlying activity labeled CTMC, where for the moment the regular and dashed arrows have the same meaning (cf. Sec. 4.2.A). The Boolean encodings of the transitions of the CTMC as produced by function $\mathcal{E}_{\mathcal{A}ct}$ and $\mathcal{E}_S$ are specified in table (C), where activity labels are encoded by a-bits, source states by s-bits and target states by t-bits. The 5 integer SVs of S are encoded by 6 Boolean variables, since only SV s_5, which represents the marking of place p_5, can take a value other than 0 or 1. Part (D) shows the corresponding ADD M, where the Boolean s-variables and the Boolean t-variables are ordered in an interleaved fashion. The rates of the transitions are stored in the terminal nodes. The ADD is ordered, i.e. on all paths from the root to a terminal node we have the same variable ordering, and it is reduced, i.e. all isomorphic substructures have been merged. In the ADD, a dashed (solid) arrow indicates the value assignment 0 (1) to the corresponding Boolean variable on the respective path. The nodes printed in dotted lines are those which get eliminated when applying the *zero-suppressing* reduction rule for ZDDs, which is applicable here in a straight-forward manner, since incidently the ADD M has no don't care nodes.

3.4 ZDD-based representation of matrices

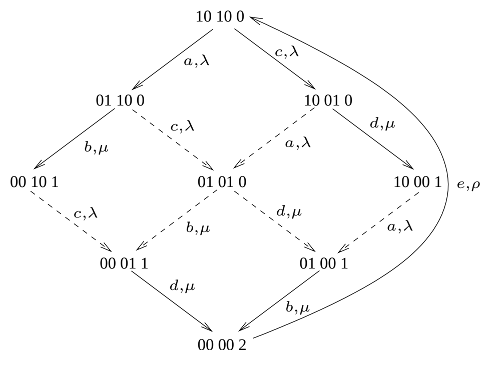
A real-valued $(2^n \times 2^n)$ matrix M is a (finite) discrete function, such that a pair of indices (r, c) is mapped to a real number $a_{r,c} \in \mathbb{R}$. If one encodes each pair of indices as a bit vector one obtains a pseudo-Boolean function. For representing $(m \times m)$ matrices, where $m \neq 2^n$, one simply needs to add an adequate number of rows and columns, containing zeroes only. Thus a Mt-DD M represents a real-valued $(2^{\frac{|\mathcal{V}|}{2}} \times 2^{\frac{|\mathcal{V}|}{2}})$ matrix M ($M \equiv \mathsf{M}$) *iff* $\forall (r, c) \in R \times C$:

(I) From a SPN to the symbolic representation of its underlying CTMC

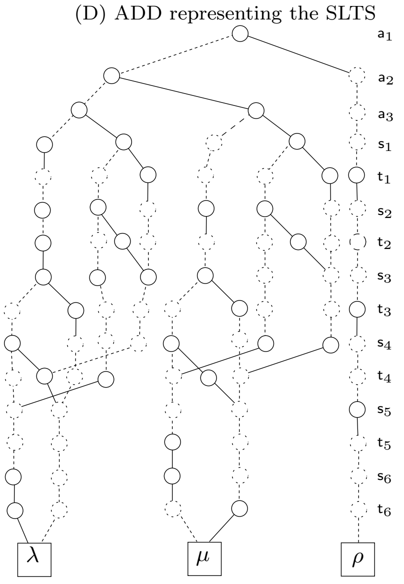
(A) A stochastic Petri net



(B) The corresponding SLTS



(D) ADD representing the SLTS



(C) Binary encodings of the SLTS

l	$\vec{a}$	$\vec{s}$	$\vec{t}$	f_M
	$a_1a_2a_3$	$s_1s_2s_3s_4s_5s_6$	$t_1t_2t_3t_4t_5t_6$	
a	000	101000	011000	λ
		100100	010100	
		100001	010001	
c	001	101000	100100	λ
		011000	010100	
		001001	000101	
b	010	011000	001001	μ
		010100	000101	
		010001	000010	
d	011	100100	100001	μ
		010100	010001	
		000101	000010	
e	100	000010	101000	ρ

(II) From the set of reachable states, to the symbolic rate reward function R^r

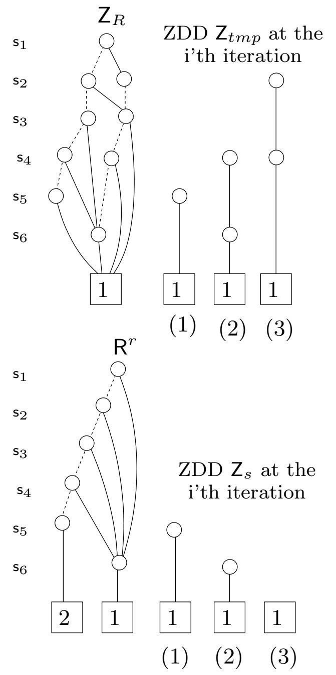


Figure 1: From a SPN to the symbolic representation of its underlying MRM

$f_M(\mathcal{E}(c), \mathcal{E}(r)) = M(r, c)$, where C, R is the set of column- and row-indices and f_M is the pseudo-Boolean function represented by Mt-DD M. Given an Mt-DD-based representation of an activity-labeled CTMC, one simply has to abstract over the binary encoded activity labels, in order to obtain a symbolic representation of the corresponding transition rate matrix. I.e. by applying the Abstract-operation on M for all variables of $\vec{a}$ one obtains the desired result ($\text{Abstract}(M, \vec{a}, +)$). In case of the example of Fig. 1.I one simply needs to abstract from the first three levels and obtains a symbolic representation of a (64×64) matrix. Within this matrix only 9 rows and 9 columns contain elements different from 0, addressing the set of reachable states. The remaining 55 dummy entries (55 columns and 55 rows) refer to encodings of unreachable states.

4. ZDD-BASED GENERATION AND SOLUTION OF MRMS

The ZDD-based scheme for analyzing MRMs consists of two stages: First one needs to generate a ZDD-based representation of the MRM as defined by the high-level model description and its user-defined PVs. Secondly one needs to compute the desired performability measures. This latter stage is realized by computing a probability for each system state of the MRM and by computing the moments of the PVs, where both steps are carried out on the basis of the ZDD-based representations as generated at the first stage.

4.1 Preliminaries

Based on the definition of sets of dependent and independent SVs (cf. eq. (1)), and based on the encoding scheme of Sec. 3.2, we define the sets of dependent Boolean variables, and the sets of their independent counterparts for each activity:

$$\mathcal{V}^{D_l} := \{\vec{s}^{\,i}, \vec{t}^{\,i} | s_i \in \mathcal{S}_l^{\mathcal{D}}\} \qquad \mathcal{V}^{I_l} := \{\vec{s}^{\,i}, \vec{t}^{\,i} | s_i \in \mathcal{S}_l^{\mathcal{I}}\} \qquad (3)$$

In this equation, $\vec{s}^{\,i}$ and $\vec{t}^{\,i}$ denote those Boolean variables which encode the value of the dependent SV s_i in the source and target state of a transition $(\vec{s}, l, \lambda, \vec{s}^{\,l})$. For convenience we gather now the dependent and independent variables in different sets, i.e. we distinguish whether they encode parts of the source states (s−variable) or target states (t-variables), yielding:

- dep. s−vars.: $\mathcal{V}_s^{D_l} := \{\vec{s}^{\,i} | \vec{s}^{\,i} \in \mathcal{V}^{D_l}\}$
- dep. t−vars.: $\mathcal{V}_t^{D_l} := \{\vec{t}^{\,i} | \vec{t}^{\,i} \in \mathcal{V}^{D_l}\}$
- indep. s−vars.: $\mathcal{V}_s^{I_l} := \{\vec{s}^{\,i} | \vec{s}^{\,i} \in \mathcal{V}^{I_l}\}$
- indep. t−vars.: $\mathcal{V}_t^{I_l} := \{\vec{t}^{\,i} | \vec{t}^{\,i} \in \mathcal{V}^{I_l}\}$

Analogously to activities, it is now assumed, that each rate reward r also has its set of dependent and independent Boolean SVs, denoted $\mathcal{V}^{D_r}$ and $\mathcal{V}^{I_r}$. For impulse rewards this is not necessary, since as one may recall, each impulse reward function $\mathcal{I}_l^i$ was defined to be limited to the set of

(A) Top level algorithm

```
GenerateSymbolicMRM()
(1)   do
(2)     do
(3)       ExploreStates()
(4)       EncodeTransitions()
(5)     until $LocalFixedPointIsReached$
(6)     $Z_T$ := ComposeActLocalSLTS()
(7)     $Z_R$ := SymbolicReachability()
(8)     InitiateNewRound()
(9)   until $GlobalFixedPointIsReached$
(10)  $Z_T = Z_T \cdot Z_R$
(11)  $Z_R^o$ := OffsetLabel($Z_R$)
(12)  $prob$ := ComputeStateProbabilities($Z_R^o, Z_T$)
(13)  MakeRateRewards($Z_R$)
(14)  MakeImpulseRewards($Z_R$)
(15)  for $p \in PV$
(16)    $Z_{rate} := \sum_{r \in \mathcal{R}^p} R^r$
(17)    $Z_{imp} := \sum_{i \in \mathcal{I}^p} I^i$
(18)    $n$ := getRoot($Z_{rate}$), $r$ := getRoot($Z_R^o$)
(19)    ComputeRew($n, r, 0, p.r\_mean, p.r\_var$)
(20)    $n$ := getRoot($Z_{imp}$)
(21)    ComputeRew($n, r, 0, p.i\_mean, p.i\_var$)
(22)    $p.r\_var := p.r\_var - p.r\_mean^2$
(23)    $p.i\_var := p.i\_var - p.i\_mean^2$
(24)  end for
```

(B) Generating symbolic rate reward functions

```
MakeRateRewards($Z_R$)
(1)   for $r \in \mathcal{R}$
(2)     $R^r := \emptyset, Z_U := Z_R$
(3)     while $Z_U \neq \emptyset$
(4)       $Z_{tmp} \xleftarrow{\vec{s}} Z_U$
(5)       $Z_s$ := Abstract($Z_{tmp}, \mathcal{V}_s^{I_r}, +$)
(6)       $rew := \mathcal{R}_r(\vec{s})$
(7)       IF $(rew \neq 0)$ THEN
(8)         $R^r := R^r + rew \cdot (Z_s \cdot Z_U)$
(9)       $Z_U := Z_U \setminus Z_s$
(10)    end while
(11)  end for
```

(C) Generating symbolic impulse reward functions

```
MakeImpulseRewards($Z_R$)
(1)   for $i \in \mathcal{I}$: $\widetilde{Z_T}$ := ZDD2zBDD($Z_T$)
(2)     for $k \in \mathcal{A}ct_i$
(3)       $I_k^i := \emptyset$
(4)       $Z_U$ := Abstract($\widetilde{Z_T} \cdot Z_k, \mathcal{V}_t, +$)
(5)       while $Z_U \neq \emptyset$
(6)         $Z_{tmp} \xleftarrow{\vec{s}} Z_U$
(7)         $Z_s$ := Abstract($Z_{tmp}, \mathcal{V}_s^{I_k}, +$)
(8)         $imp := \mathcal{I}_k^i(\vec{s})$
(9)         IF $(imp \neq 0)$ THEN
(10)          $I_k^i := I_k^i + imp \cdot (Z_s \cdot Z_U)$
(11)        $Z_U := Z_U \setminus Z_s$
(12)      end while
(13     end for
(14)    $I^i := \sum_{k \in \mathcal{A}ct_i} I_k^i$
(15)  end for
```

(D) Computing Rewards

```
ComputeRew($n, r, off, m, v$)
(1) IF $n \in \mathcal{K}_T$ THEN
(2)     $m := m + prob[off] * $value($n$)
(3)     $v := v + prob[off] * $value($n$)$^2$
(4) ELSE IF var($n$) $_\pi>$ var($r$) THEN
(5)     ComputeRew($n$, else($r$), $off, m, v$)
(6) ELSE
(7)     ComputeRew(then($n$), then($r$),
                 $r.offset + off, m, v$)
(8)     ComputeRew(else($n$), else($r$),
                 $off, m, v$)
```

Figure 2: Algorithms for generating and solving a symbolic representation of a MRM

the dependent variables of the reward inducing activity l (cf. last paragraph of Sec. 2).

4.2 Constructing a ZDD-based representation of a MRM

(A) Constructing the representation of the CTMC: The scheme, denoted as activity-local SG generation scheme, for generating a ZDD-based representation of a CTMC from a high-level model description was already introduced in [12]. For a better understanding it is roughly recapitulate now. The main idea of the activity-local SG generation scheme is the partitioning of the SLTS T to be generated, into sets of transitions with label $l \in \mathcal{A}ct$, where each state is reduced to the activity-dependent markings:

$$T^l := \{(\vec{s}_{d_l}, l, \lambda, \vec{s}^{\,l}_{d_l}) \mid (\vec{s}, l, \lambda, \vec{s}^{\,l}) \in T\} \qquad (4)$$

During SG generation the activity-local transitions T^l are successively generated, where each is encoded by its own (activity-local) ZDD Z_l, which solely depends on the Boolean variables of $\mathcal{V}^{D_l} = \mathcal{V}_s^{D_l} \cup \mathcal{V}_t^{D_l}$. However, instead of a standard search scheme, we follow a selective breadth-first-search strategy, i.e. for a detected state $\vec{s}^{\,l}$, which was reached by firing action l in state $\vec{s}$, one generates the set of successor states by executing those enabled activities $k \in \mathcal{A}ct$, which are also dependent on l ($\mathcal{V}^{D_l} \cap \mathcal{V}^{D_k} \neq \emptyset$), and which have not already been tested on the activity-local marking of state $\vec{s}^{\,l}$. This functionality is realized by the routines ExploreStates, EncodeTransitions, ComposeActLocalSLTS, SymbolicReachability and InitiateNewRound as called in the top-level algorithm of Fig. 2.A. Hereby the procedures ExploreStates and EncodeTransitions are called in an alternating fashion in order to carry out explicit SG exploration and the encoding of the detected transitions. If a local fixed point is reached, i.e. if from a given set of states all sequences of dependent activities are extracted explicitly, symbolic composition (line 6) and symbolic reachability (line 7) take place, yielding the set of reachable states generated so far. Since this might result in states which may trigger new model behavior, InitiateNewRound is called. This procedure tests states for new model behavior and may therefore trigger new rounds of explicit SG exploration and encoding. Several rounds of explicit SG generation, symbolic composition, symbolic reachability analysis and re-initialization may be required until a complete representation of the user-defined CTMC is constructed. After reaching a global fixed point, the procedure is complete and one simply needs to restrict the set of potential transitions to the reachable ones by multiplying the respective ZDDs (line 10 of Fig. 2.A).
For exemplification we return once again to Fig. 1.I.B. The transitions explicitly generated and encoded are depicted as regular arrows, in contrast the transitions resulting from symbolic composition are given as dashed arrows. Consequently, states (01 00 1) and (00 01 1) are only generated on the level of the symbolic SG representation, however they trigger new explicit model behavior, state (01 00 1) for activity b and state (00 01 1) for activity d. This is detected by procedure InitiateNewRound, so that a new round of explicit SG exploration follows until the complete activity-labeled CTMC is generated.

(B) Generating ZDD-based representations of PVs: As pointed out in Sec. 2.C, user-defined PVs consist of a set of rate reward - and / or impulse reward definitions. Consequently at first one generates the symbolic representations of the underlying rate and impulse reward functions (line 13-14 of top-level algorithm of Fig. 2.A). Hereby the main idea is once again to exploit locality, so that the explicit evaluation of each reward function is limited to a fraction of states of the CTMC, rather than evaluating the reward functions for each state.

(i) *Generating ZDD-based representations of rate rewards:* Algorithm MakeRateRewards as specified in Fig. 2.B consists of two nested loops. The outer `for`-loop processes each rate reward definition as contained in a user-defined PV, whereas in the inner `while`-loop sets of states are processed. I.e. at first one pops an arbitrary state from the set of reachable states (line 4). This state is reduced to the positions referring to the rate reward-dependent SVs by simply abstracting Z_{tmp} from those Boolean variables referring to the rate reward's set of independent SVs (line 5). Now one simply calculates r's rate reward for the popped state vector by executing the respective rate reward function $\mathcal{R}_r(\vec{s})$ (line 6). In case the obtained reward rew is not equal to 0, one multiplies Z_s, Z_R and rew. The newly obtained pairs of full (!) states and rate rewards are then added to the previously computed pairs as represented by Mt-DD R^r (line 8). Now one removes all states from the set of states represented by Z_U, containing the rate reward-dependent state marking as encoded by Z_s, which might remove a whole set of states form Z_U. The whole procedure is repeated until all rate reward-dependent partitions of Z_R are processed, i.e. until Z_U is empty. At termination a ZDD-based representation for each rate reward function as contained within a PV is generated.
For exemplification, assume that rate reward r is defined as the number of tokens contained in place p_5 of the SPN of Fig. 1.I.A. ZDD Z_R of Fig. 1.II encodes the set of reachable states, which is the initial value of Z_U. Let us further assume, that the states popped from the ZDD Z_U in the inner for loop are the following: (00 00 2), (00 01 1) and (01 01 0). The corresponding ZDDs Z_{tmp} and Z_s, as obtained after executing line 4 and 5 of algorithm MakeRateRewards for the three states are also depicted in Fig. 1.II. The final encoded rate reward function obtained at termination is given as ZDD R^r. Rather than computing and encoding the rate reward r for each of the 9 states, this is only done 3 times, namely once for the states where $p_5 = 2$, once for the states where $p_5 = 1$, and once for the states where $p_5 = 0$.

(ii) *Generating ZDD-based representations of impulse rewards:* The algorithm MakeImpulseRewards for calculating impulse reward functions is specified in Fig. 2.C. Since each activity may generate different impulse rewards for different impulse reward definitions, one needs to iterate over three nested loops. The outer two `for`-loops process each impulse reward definition and its respective sets of activities. The inner `while`-loop processes one state for each activity-local marking in which the respective activity is enabled and calculates the respective impulse reward (line 5-12). In case the obtained impulse reward for a state is not equal to 0, one multiplies the ZDD-based representation of all states being equivalent (concerning the activity-local marking) to the currently processed state with the previously computed impulse reward imp ($imp \cdot \mathsf{Z}_s \cdot \mathsf{Z}_U$) (line 10). However, due to the construction of Z_U (line 4), the obtained pairs of states and impulse rewards are automatically weighed by the execution rate of the activity under process. The newly obtained pairs of full states and weighed impulse rewards are then added to the set of previously computed impulse rewards. This procedure is repeated until all "*activity-local*" markings are processed, i.e. until ZDD Z_U is empty. This yields a ZDD for each impulse reward function as contained within a user-defined PV.

4.3 Computing the performability measures

(A) Computing state probabilities After a symbolic representation of the MRM is generated, the ZDD representing the set of reachable states is augmented by offset-labels (line 11 of algorithm of Fig. 2.A). Steady state or transient state probabilities are subsequently computed by applying the ZDD-based variant of the numerical solution method as incorporated into routine ComputeStateProbabilities (line 12 of algorithm of Fig. 2.A).
The iterative solvers 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. 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 (as an example, 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-labeling, as had been first suggested in [17] for the ADD data structure. In an offset-labeled ADD, each node is equipped with an offset value. While traversing the ADD representation of a 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 $\vec{s}$ and $\vec{t}$ vectors to a pair (r, c) of row and column indices. Using ZDDs we had to adapt the concept of offset-labeling:

- With ADDs, skipped nodes (corresponding to don't cares) must be reinserted, because they carry an offset (which is relevant if their `then`-edge is followed). With ZDDs, skipped nodes correspond to zero-valued variables for which the offset is irrelevant. Therefore, in the ZDD case, skipped nodes do not have to be reinserted, which keeps the symbolic data structure compact.
- Similar to the ADD case, a ZDD node may have to be duplicated if the offset of a shared node is different on different paths (also called "offset clash").

The space efficiency of ZDD-based matrix representation comes at the cost of computational overhead, caused by the recursive traversal of the Mt-DD during access to the matrix entries. Analogously to [17], we replace the lower levels of the ZDDs by explicit sparse matrix representations, which works particularly well for block-structured matrices. We call the resulting data structure *hybrid offset-labeled* Mt-DD (HO Mt-DD), where a Mt-DD is either an ADD or ZDD. The level at which one replaces the remaining Mt-DD-levels with a sparse matrix representation is called *sparse level*. It depends on the available memory space, i.e. there is a typical time/space tradeoff. In the following we will refer to this level by the ratio s, such that $sparse\_level := \lfloor |\mathcal{V}|(1-s) \rfloor$.

N	$states$	$trans$	N	$states$	$trans$	N	$states$	$trans$	N	$states$	$trans$
Kanban [4]			Courier [23]			FMS [5]			Polling [8]		
5	$2.5464E6$	$2.4460E7$	3	$2.3812E6$	$1.31037E7$	6	$5.3777E5$	$4.2057E6$	15	$7.3728E5$	$6.144E6$
6	$1.1261E7$	$1.1571E8$	4	$9.7102E6$	$5.7005E7$	8	$4.4595E6$	$3.8534E7$	18	$7.0779E6$	$6.9599E7$
7	$4.1645E7$	$4.5046E8$	5	$3.2405E7$	$1.9983E8$	10	$2.5398E7$	$2.3452E8$	20	$3.1457E7$	$3.4078E8$
			6	$9.3302E7$	$5.9818E8$	12	$1.11415E8$	$1.07892E9$	21	$6.6060E7$	$7.4868E8$

Kanban: Kanban Manufacturing System, Courier: Courier Protocol,
FMS: Flexible Manufacturing System, Polling: Cyclic Server Polling System

Table 1: Model specific data for the various case studies

For numerical analysis, it is well-known that the Gauss-Seidel (GS) scheme and its over-relaxed variant typically exhibit much better convergence than the Jacobi (JAC), Jacobi-Over-relaxation (JOR) or power method. However, Gauss-Seidel requires row-wise access to the matrix entries, which, unfortunately, cannot be realized efficiently with Mt-DD-based representations. As a compromise we adapted the so-called pseudo-Gauss-Seidel (PGS) iteration scheme [17] to the case of HO ZDDs. For doing so the overall matrix is partitioned into blocks (not necessarily of equal size, due to unreachable states). Within each block, access to matrix entries is in arbitrary order, but the blocks are accessed in ascending order. PGS requires one complete iteration vector and an additional vector whose size is determined by the maximal block size. Given a HO Mt-DD which represents the matrix, each inner node at a specific level corresponds to a block. Pointers to these nodes can be stored in a sparse matrix, which means that effectively the top levels of the HO Mt-DD have been replaced by a sparse matrix of block pointers. The level at which the root nodes of the matrix blocks reside is called *block level*. In the sequel we will refer to this level by the ratio b, such that $block\_level := \lfloor |\mathcal{V}|b \rfloor$. Overall, this yields a memory structure in which some levels from the top and some levels from the bottom of the HO Mt-DD have been replaced by sparse matrix structures. We call such a memory structure a block-structured hybrid offset-labeled Mt-DD (BHO Mt-DD), where Mt-DD is once again either an ADD or a ZDD. The choice of an adequate s and an adequate b is an optimization problem. In general, increasing b improves convergence of the PGS scheme, and replacing more Mt-DD levels by sparse structures improves speed of access. Since ZDDs are often more compact, their processing requires less CPU-time, if compared to ADDs. Due to their lower memory requirements they furthermore allow larger values of b and s, yielding an additional speed-up, since the number of nodes to be traversed is reduced. If the *block-level* meets the *sparse-level*, as has been described in [13] and [24], all Mt-DD levels have disappeared and the PGS scheme becomes a proper GS scheme, but in most interesting cases this situation cannot be realized since memory is at a premium. Our experiments, carried out in [24], showed that using BHO-ZDDs an optimal choice for b lies often beyond $\frac{1}{2}$, where the heuristic developed in [17] for ADDs suggests $b := \frac{1}{3}$.

(B) Computing PVs Routine ComputeStateProbabilities delivers the vector *prob*, containing either steady state or transient state probabilities (line 12 of algorithm of Fig. 2.A). What follows next is the generation of the ZDD-based representations of rate and impulse reward functions (line 13 and 14), as well as their aggregation as specified by each user-defined PV p (line 16-17). Given the resulting symbolic representations $\mathsf{Z}_{rate}, \mathsf{Z}_{imp}$ and the probability vector, one is enabled to compute the first and second moment of PV p by simultaneously traversing the offset-labeled Z-BDD Z^o_R and Z_{rate} or Z_{imp}, which is the idea behind the algorithm of Fig. 2.D. While traversing the ZDDs, the state index of the traversed path is obtained by summing over the offsets of nodes left via `then`-edge (line 7-8 of algorithm of Fig.2.D). In case one reaches a terminal non-zero node, the index of the current state is determined. Now one may successively compute mean and second moment of the reward, as we do in line 2-3, where the respective steady state or transient state probability is stored within the probability vector at the position given by off. After calculating the variance of the impulse and rate reward of PV p (line 22-23 of algorithm of Fig.2.A) the process is complete and one may resume with the next PV.

5. EMPIRICAL EVALUATION

We implemented the presented Mt-DD-based framework within the Möbius modeling tool [16], where the implementation consists of four modules:

1. A module for the explicit generation of states, which constitutes the interface between the symbolic engine and Möbius (algo. ExploreStates).

2. The symbolic SG generation engine (mainly algo. EncodeTransitions, ComposeActLocalSLTS, SymbolicReachability and InitiateNewRound) which generates a Mt-DD-based representation of the CTMC of the low-level MRM.

3. A ZDD-library, which is based on the CUDD-package [22]. This library mainly contains the C++ class definition of *partially shared ZDDs*, the new recursive algorithms for manipulating them and the operator-caches. In case of ADDs we employ the C++ classes, algorithms and operator caches as provided by the CUDD-package.

4. A library for computing the user-defined performance variables. I.e. this module contains (a) algorithm `ComputeStateProbabilities` by implementing the new ZDD-based solver and our versions of the ADD-based solvers of [17], (b) the new algorithms MakeRateRewards and MakeImpulseRewards for efficiently generating symbolic representations of rate and impulse reward functions and (c) the new algorithm ComputeRew for computing first and second moment of user-defined PVs via Mt-DD-traversal (Fig. 2.B-D).

In order to evaluate the proposed innovations, we analyzed

FMS

N	JAC with HO Mt-DDs				PGS with BHO Mt-DDs				Uniform. with HO Mt-DDs			
	# iter	t_{iter} in sec.		r_{iter}	# iter	t_{iter} in sec.		r_{iter}	# step	t_{step} in sec.		r_{step}
		ADD	ZDD			ADD	ZDD			ADD	ZDD	
6	845	0.1878	0.0945	1.99	569	0.2083	0.0753	2.77	1,508	0.09695	0.0557	1.7406
8	1,127	1.5520	0.6445	2.41	737	1.6935	0.5439	3.11	1,864	1.55200	0.8768	1.7701
10	1,415	8.7106	4.3969	1.98	892	9.6432	3.8183	2.53	2,217	8.71055	5.3400	1.6312
12					1,038	39.831	23.405	1.70				

Kanban

N	JOR with HO Mt-DDs				PGS with BHO Mt-DDs				Uniform. with HO Mt-DDs			
	# iter	t_{iter} in sec.		r_{iter}	# iter	t_{iter} in sec.		r_{iter}	# step	t_{step} in sec.		r_{step}
		ADD	ZDD			ADD	ZDD			ADD	ZDD	
5	1977	0.6849	0.3233	2.12	1542	0.8345	0.2878	2.90	1,157	0.52530	0.3060	1.7167
6	2785	3.2299	1.4929	2.16	2176	3.8845	1.3681	2.84	1,157	2.48796	1.4700	1.6925
7	3724	10.9477	5.0642	2.16	2913	15.0764	5.1502	2.93	1,157	9.47386	5.5929	1.6939

(A) Steady state analysis, $b := 0.35$ and $s := 0.35$ (B) Transient analysis, $s := 0.7$

Table 2: ADD- and ZDD-based solution of CTMCs, with relative convergence criterion and accuracy $\epsilon = 10^{-9}$

four models which are commonly employed as benchmarks in the literature. Table 1 gives the sizes of their CTMCs, i.e. the number of states (*states*) and number of transitions (*trans*). The experiments of Table 2.A (except FMS 12) were carried out on a Pentium IV with 3 GHz, 1 GByte RAM and a Linux OS. All other results, i.e. the experiments of Table 2.B and 3 and the FMS 12 model of Table 2, were collected on a Pentium IV 2.88 GHz, equipped with 3 GByte of RAM and a Linux OS. Since current Linux kernels limit the memory space of a single process to 3 GByte, the MRM to be solved are limited to models where the probability vector, iteration vector and vector of diagonal matrix elements is at most $\sim$ 2 GByte. In order to simplify the comparison, we decided to present also ratios, where the respective figures are always normed to the figures of the proposed innovations. Ratios > 1 indicate an advantage of the innovations developed in this work, and ratios < 1 indicate their disadvantage. CPU times are given in seconds and memory consumption is given in MByte.

5.1 ADD and ZDD data structures

[12] already reported that the use of ZDDs may reduce space and time for generating activity-labeled CTMCs for different high-level models by a factor of 2-3, if compared to ADDs. A similar picture can be drawn when it comes to the computation of steady state and transient state probabilities. Table 2.A shows the run-time data when computing steady state probabilities for FMS and Kanban under different scaling parameters (N). Here we restrict ourselves to applying JAC or JOR and the backward PGS method. Furthermore, we converted approx. the lower third of the index-labeled Mt-DDs into sparse matrices, i.e. $s := 0.35$. In case of the PGS method also the upper third of the Mt-DDs was removed, i.e. $b := 0.35$. Given the higher sparseness of ZDDs, one is enabled to choose the block-level at a lower level than under the BHO ADD-based layout and gain even more advantage of the good convergence behavior of the PGS method. However, doing so increases the number of blocks, so that a sparse-matrix layout for administering them often requires more memory than available. We eliminated this drawback by employing a linked list for administering the root nodes of the HO Mt-DDs representing non-empty block entries. As it turns out, such a layout reduces not only the memory space but also computation time (for the PGS method only!), since empty blocks can simply be ignored. But due to fairness we removed under ADDs and ZDDs only the upper third of Mt-DD levels, which is the heuristic suggested in [17].

From Table 2 one may conclude, that the employment of ZDDs yields clear runtime advantages, which stems from the maintenance of their compactness under the offset-labeling scheme. Under the PGS method this speed-up could be even more improved, since the compactness of ZDDs allows to choose the block-level at a lower level than under the BHO ADD-based layout. I.e. in terms of absolute numbers one only needs 4.17 hours for solving the Kanban system for $N = 7$, rather than 12.20 hours under the PGS-method in case of BHO ADDs. If block- and sparse level are chosen in such a way that ZDD- and ADD-based BHO-Mt-DDs consume almost the same size of memory (for the BHO-ZDD-based we have $b := 0.5$ and $s := 0.4$), the computation of steady state probabilities for the FMS 12 model requires only 3.83 ($\approx 16.86 * 821/3600$) hours of CPU time (cf. table 3.A col. *each iter.*), where the original ADD-based variant with $b := 0.35$ and $s := 0.35$ requires 11.48 ($\approx 39.83 * 1038/3600$) hours (cf. table 2.A col. t_{iter}, for the FMS 12 model).

Table 2.B shows the run-time data when computing transient state probabilities, where we employed the uniformization method and the Fox-Glynn method for computing the values of the Poisson distribution. Here we decided to set $s := 0.7$, since memory space was available and doing so speeds up the solution. As a consequence of this, the sparseness of HO ZDDs is less significant over their ADD-based counterparts. Given also the fact, that the actual amount of CPU time spent for computing new vector entries (not traversing the Mt-DD-structures but computing the Poisson probabilities) is also higher than in case of the iterative methods for computing steady state probabilities, it is not surprising, that ZDDs realize here smaller speed-ups.

5.2 ZDD and sparse matrix layouts

Table 3 gives empirical results as obtained from steady state analysis for the benchmark models, where typical performance measures such as the mean value of a set of SVs had to be computed. For obtaining steady state solutions, the Gauss-Seidel method for the sparse matrix layouts and

(B) Memory consumption of solvers

	N	Symbolic solver: MByte of memory consumed for		Sparse solver: MByte of memory consumed for		ratios for	
		overall exec.	matrix rep.	overall exec.	matrix	overall mem.	matrices
FMS	6	21	1.743	80	56.336	3.810	32.332
	8	96	5.248	688	509.032	7.167	96.992
	10	458	16.146	xxx	xxx	xxx	xxx
	12	1876	43.485	xxx	xxx	xxx	xxx
Kanban	5	49	0.835	451	318.778	9.204	381.657
	6	191	1.809	xxx	xxx	xxx	xxx
	7	670	3.600	xxx	xxx	xxx	xxx
Courier	3	60	1.805	323	186.294	5.383	103.210
	4	195	6.581	1190	800.535	6.103	121.640
	5	571	13.449	xxx	xxx	xxx	xxx
	6	1551	24.756	xxx	xxx	xxx	xxx
Polling	15	16	0.501	121	81.562	7.563	162.640
	18	117	1.345	xxx	xxx	xxx	xxx
	20	495	3.129	xxx	xxx	xxx	xxx
	21	1052	3.220	xxx	xxx	xxx	xxx

(A) CPU time consumption of solvers

	N	Symbolic solver: CPU time in sec. consumed for			Sparse solver: CPU time in sec. consumed for				ratios for	
		SG gen.	each iter.	PV calc.	file reading	each iter.	PV calc.		iter. time	PV time
FMS	6	0.29	0.073728	0.11	15.02	0.0562	6.47	1.14	0.76	69.17
	8	0.79	0.61602	0.71	137.58	0.509	56.05	15.57	0.83	100.88
	10	1.64	3.595877	4.15	xxx	xxx	xxx	xxx	xxx	xxx
	12	3.92	16.8645	17.71	xxx	xxx	xxx	xxx	xxx	xxx
Kanban	5	0.51	0.347	0.26	102.86	0.249	30.36	5.6	0.718	138.32
	6	0.58	1.578	1.22	xxx	xxx	xxx	xxx	xxx	xxx
	7	1.17	6.170	4.27	xxx	xxx	xxx	xxx	xxx	xxx
Courier	3	1.78	0.241	0.18	53.36	0.178	29.33	7.16	0.737	202.70
	4	2.92	1.003	0.76	316.53	0.751	118.72	35.11	0.749	202.41
	5	4.54	3.415	2.62	xxx	xxx	xxx	xxx	xxx xxx	xxx
	6	6.61	9.948	7.25	xxx	xxx	xxx	xxx	xxx	xxx
Polling	15	0.03	0.085	0.03	27.21	0.065	8.76	1.5	0.772	341.89
	18	0.06	0.937	0.31	xxx	xxx	xxx	xxx	xxx	xxx
	20	0.08	4.445	1.39	xxx	xxx	xxx	xxx	xxx	xxx
	21	0.09	9.603	2.91	xxx	xxx	xxx	xxx	xxx	xxx

Table 3: Computing performability measures

the PGS method for the ZDD-based matrix layout were applied. As a consequence, the ZDD-based solver had to execute sometimes a clearly increased number of iterations (factor 1.77 up to 6.23). However, as illustrated by Table 3, this is justified, since the employment of a ZDD-based engine within the Möbius modeling environment allows the analysis of models, which were not analyzable under Möbius' conventional schemes for constructing and solving a MRM.[1] This limitation has to do with the fact that Möbius stores the generated CTMC and its reward information in a non-compact ASCII format on hard drive, limiting the size of MRMs to be handled ($\sim 5 * 10^6$ states). Not enough, this information must be reloaded into RAM before the iterative solution process can be initiated (cf. col. *"file reading"* in Table 3.A, which we did not include in the CPU time consumed for each iteration as given in col. *"each iter."*). In addition to these tool-specific disadvantages, conventional schemes also have the following problems: (a) the sparse matrix format is hampered by its memory requirements as illustrated in col. *"matrix"* of Table 3.B. (b) computation of PVs for each state during SG exploration (cf. left figure of col. *"PV calc."* of Table 3.A), as well as reading the PVs from an ASCII-file, allocating a respective PV vector of appropriate size and finally computing the moments and variance of the PV (right figure col. *"PV calc."* of Table 3.A), induce a run-time overhead.

In contrast, the proposed ZDD-based scheme generates a symbolic representation of the MRM each time the solver is started. Hereby the times for generating a ZDD-based representation of the CTMC as well as generating ZDD-based representations of the reward functions and computing mean and variance of the user-defined PVs, once steady state or transient state probabilities have been computed, is obviously negligible (cf. col. *"SG gen."* and *"PV calc."* of Table 3.A). Furthermore, the compactness of the (B)HO-ZDD-based representation speaks to the advantage of the here presented approach, since it is still superior even though we employed a setting which improves the CPU time consumption per iteration at the disadvantage of space complexity ($b := 0.5$ and $s := 0.4$). This might explain why the ZDD-based solvers are not significantly slower than the standard sparse matrix ones. Since the matrix representation under such a choice is still very compact, it is clear that the memory space for storing the probability vectors is the limiting factor as the low-level MRMs become larger. This

[1]In Table 3 columns filled with xxx correspond to experiments which could not be solved by the sparse matrix-based solver due to memory limitations.

also exhibits another advantage of the ZDD-based scheme. The proposed scheme for generating and computing PVs (algorithm of Fig. 2.B and 2.C in combination with algorithm of Fig. 2.D) allows not only the efficient construction of a ZDD-based representation and a ZDD-based computation of reward functions (cf. columns *"PV calc."* and col. *"PV time."* of Table 3.A), it also avoids to employ additional vectors for storing the individual reward values of each state as realized by the standard Möbius solver module. This also explains why Möbius' sparse matrix solver modules require significant more memory for the overall process (cf. columns *"overall exec."* of Table 3.B).

6. SUMMARY AND FUTURE WORK

This paper presented a ZDD-based framework for analyzing high-level MRMs, containing the following three innovations: (a) a scheme for efficiently generating a symbolic representation of a low-level MRM, including ZDD-based representations of user-defined PVs, (b) *ZDD-based solvers* for computing steady state and transient probabilities, resulting in a reduction in computation time, and (c) an algorithm for efficiently computing moments of user-defined PVs once their ZDD-based representation and a vector of state probabilities is given.
Since we develop our implementations in the context of Möbius, we are currently implementing an efficient symbolic realization of the "Replicate" feature, such that the lumping theorem for MRMs can be applied in a straight forward manner. In addition, aggregation methods for the approximate solution of ZDD-represented MRMs seem to be a promising candidate for future research.

Acknowledgment: We thank the Möbius developer group for their support, and our students D. Zimmermann and S. Harwarth for contributing to the ZDD-based solvers.

7. REFERENCES

[1] *Formal Methods in System Design: Special Issue on Multi-terminal Binary Decision Diagrams*, Volume 10, No. 2-3, April - May 1997.

[2] S.B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

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

[4] G. Ciardo and M. Tilgner. On the use of Kronecker operators for the solution of generalized stochastic Petri nets. ICASE Report 96-35, 1996.

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

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

[7] S. Harwarth. Computation of transient state probabilities and implementing Möbius' "state-level abstract functional interface" for the data structure ZDD, 2006. Masters Thesis. University of the Federal Armed Forces Munich (Germany).

[8] O. Ibe and K. Trivedi. Stochastic Petri net models of polling systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, 1990.

[9] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1-2):9–62, 1998.

[10] M. Kuntz, M. Siegle, and E. Werner. Symbolic Performance and Dependability Evaluation with the Tool CASPA. In *Proc. of EPEW*, pages 293–307. Springer, LNCS 3236, 2004.

[11] K. Lampka and M. Siegle. MTBDD-based activity-local State Graph Generation. In *Proc. of PMCCS 6*, pages 15–18, 2003.

[12] K. Lampka and M. Siegle. Activity-Local Symbolic State Graph Generation for High-Level Stochastic Models. In *Proc. of 13'th GI/ITG Conference Measuring, Modelling and Evaluation of Communication and Computer Systems (MMB)*, pages 245–263. VDE-Verlag, 2006.

[13] R. Mehmood. Disk-based techniques for efficient solution of large Markov chains, Ph.D. Thesis, University of Birmingham (U.K.), 2005.

[14] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proc. of DAC*, pages 272–277, Dallas (Texas), USA, June 1993. ACM Press.

[15] A. S. Miner. Efficient solution of GSPNs using Canonical Matrix Diagrams. In *Proc. of the 9'th Int. Workshop on Petri Nets and Performance Models (PNPM'01)*, pages 101–110, Aachen, Germany, September 2001.

[16] Möbius web page. http://www.mobius.uiuc.edu/.

[17] D. Parker. Implementation of Symbolic Model Checking for Probabilistic Systems, Ph.D. Thesis, University of Birmingham (U.K.), 2002.

[18] PRISM web page. http://www.cs.bham.ac.uk/∼dxp/prism/.

[19] W. H. Sanders and J. F. Meyer. A unified Approach for specifying Measures of Performance, Dependability, and Performability. In *Dependable Computing for Critical Applications*, Vol. 4, pages 215–237. Springer-Verlag, 1991.

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

[21] SMART web page. http://www.cs.ucr.edu/∼ciardo/SMART.

[22] F. Somenzi. CUDD Package, Release 2.4.x. http://vlsi.colorado.edu/˜fabio.

[23] M. Woodside and Y. Li. Performance Petri net analysis of communications protocol software by delay-equivalent aggregation. In *Proc. of 4'th PNPM*, pages 64–73, 1991.

[24] D. Zimmermann. Implementierung von Verfahren zur Lösung dünn besetzter linearer Gleichungssysteme auf Basis von Zero-suppressed Multi-terminalen Binären Entscheidungsdiagramme, 2005. Masters Thesis (in German), University of the Federal Armed Forces Munich (Germany).