Activity-Local Symbolic State Graph Generation for High-Level Stochastic Models

Kai Lampka and Markus Siegle

Universität der Bundeswehr München, Institut für Technische Informatik
{lampka,siegle}@informatik.unibw-muenchen.de

Abstract. This paper introduces a new, efficient method for deriving compact symbolic representations of very large (labelled) Markov chains resulting from high-level model specifications such as stochastic Petri nets, stochastic process algebras, etc.. This so called "activity-local" scheme is combined with a new data structure, called zero-suppressed multi-terminal binary decision diagram, and a new efficient "activity-oriented" scheme for symbolic reachability analysis. Several standard benchmark models from the literature are analyzed in order to show the superiority of our approach.

1 Introduction

High-level model description methods such as stochastic Petri nets (SPN) or stochastic process algebras (SPA) have shown to be powerful tools for describing and analyzing concurrent systems. In this work we consider high-level model specifications of this kind, where low-level representations, i.e. (labelled) Markov chains, are obtained by state graph (SG) exploration in order to carry out qualitative and quantitative analysis. Unfortunately, the standard interleaving semantics leads to the well-known phenomenon of state space explosion. In this paper, a new method for constructing and representing SGs, in combination with a new data structure and a new algorithm for symbolic reachability analysis, is presented. In contrast to all existing efficient techniques, our method *does not require the high-level model to have any particular compositional structure.* The only requirement is the existence of a dependency relation between the model's sets of activities and state variables. Results obtained from an implementation of our method in the context of the Möbius modelling framework [DCC$^+$02] show that our method is highly run-time efficient and highly memory efficient. The new data structure, as well as the new scheme for symbolic reachability analysis, are applicable beyond the *activity-local* approach developed here. Performance evaluation tools such as CASPA [KSW04] and PRISM [Par02], which are based on binary decision diagrams and traditional symbolic reachability analysis, may benefit from these innovations.
The paper is organized as follows: Sec. 2 reviews and classifies related work on symbolic SG generation. Sec. 3 introduces the model world and the symbolic encoding of labelled Markov chains by multi-terminal zero-suppressed binary

decision diagrams (ZDDs). Sec. 4 explains our new algorithms and discusses their features. Empirical results, including comparisons with other tools and implementations, are presented in Sec. 5, and Sec. 6 concludes the paper.

2 Related work

In the context of stochastic modelling, the most prominent decision diagrams (DDs) are *multi-terminal* or *algebraic BDDs* (ADDs) [FMY97], *multi-valued decision diagrams* (MDDs) [KVBSV98] and *matrix diagrams* [Min01]. In the following, a review and classification of symbolic SG generation schemes will be given. At the top level, we distinguish between monolithic and compositional approaches. Monolithic approaches do not exploit any structure of the high-level model, the SG is generated by exploring all enabled activities in each state. In contrast, compositional approaches are based on SG exploration of submodels and on operators for the symbolic composition of these local SGs. We further distinguish between fully symbolic approaches and hybrid approaches, where hybrid characterizes a combination of explicit exploration and symbolic encoding. Fully symbolic methods require a symbolic realization of the next-state function.

Monolithic approaches: These methods either suffer from long run-times or are specialised for a particular model description method.

1. **Hybrid**: In [DKK02] the reachability set of a SPN is generated by standard SG exploration, where each detected state is inserted into the symbolic representation of states reached so far. Since every arc of the SG is traversed, the approach suffers from long run-times. The memory savings are due to the use of P-invariants, which restricts the method to a certain class of SPN.
2. **Fully symbolic**: The method presented in [PRCB94] uses a symbolic next-state function for each activity[1] of a non-stochastic, 1-bounded PN. It generates the set of reachable markings by performing the standard breadth-first search (bfs) algorithm for symbolic reachability analysis. This approach is highly efficient, but its applicability is limited to PNs.

Compositional approaches: Compositionality has been considered to be crucial for the success of symbolic methods, since (a) it may significantly reduce the number of transitions to be explicitly explored and (b) it induces regularity and thus compactness of the symbolic structures. Since all of the approaches listed below require a compositional structure of the high-level model, one should remember that the partitioning of flat models into independent submodels with local SGs of adequate size is still an open question.

1. **Hybrid**: The SGs of the submodels are explored explicitly, each one being represented by its own DD. These local SG representations are then composed by applying a symbolic composition scheme. This yields a set of potential transitions which needs to be reduced to the set of actually reachable ones. Symbolic composition may take either of the two following forms:

[1] Contrary to standard PN terminology, the term *activity* is used here, because we use the term *transition* to denote the arcs of the SG.

   (a) Synchronization over a set of activities, either by employing a Kronecker structure [CM99], or by applying a symbolic synchronization operator [Sie98,Sie02].
   (b) Composition via state variable sharing, and application of a symbolic *Join*-operator [LS02].
2. **Fully symbolic**: In this case, a DD is derived directly from the modular high-level specification. The symbolic encodings of the local SGs are composed by applying symbolic synchronization operators, followed by symbolic reachability analysis [Par02,KS02].

The approaches listed above are all limited to specific model description methods or to cases where an upper bound for the value of each state variable (SV) is known a priori. This restricts their applicability to models where (a) the bounds are specified in the model [KS02,Par02], (b) bounds can be computed, e.g. by means of invariant analysis [PRCB94,DKK02], or (c) the local SGs can be generated in isolation [CM99,Sie98,Sie02,LS02]. In order to overcome this restriction, recently developed methods generate the local SGs in an interleaved fashion [CMS03,DKS04], but they depend on a good partitioning of the overall model, in order to be efficient. Thus their application is problematic in case of flat, non-modular models where an adequate partitioning is not obvious. Our activity-local scheme overcomes this drawback by maintaining compositionality at the lowest level, i.e. the level of individual activities, and only requires knowledge about the dependencies between activities and SVs. Due to the nature of Bryant's `Apply`-algorithm [Bry86], the generated activity-local SGs can be composed even if they do not fulfill a product-form requirement as it is essential for the Kronecker-based schemes [CM99,CMS03] and for the symbolic synchronization approaches [Sie98,Sie02,KS02,Par02].
In order to extend the saturation technique of [CMS03] to a general class of models, [Min04] describes an event-oriented approach, where a kind of dependency relation among events, as well as an `Apply`-algorithm for building the cross-product of two matrix diagrams, is employed. This allows [Min04] to use the same composition scheme in the context of matrix diagrams, as already introduced for BDD-based schemes in [LS02] and extended in [LS03]. These ideas, which allow one to apply symbolic SG generation techniques to models where the Kronecker-product-form requirement does not hold, are still at the core of the activity-local scheme described here, but the present paper has more to offer, namely a new data structure and a new scheme for symbolic reachability analysis, which follows an activity-wise strategy. Similiar to the approach of [BCL91], this new reachability scheme handles partitions of the overall transition system sequentially, rather than executing them all at once, which leads to runtime reduction. This also enables so-called *greedy chaining* on the set of states to be explored in the next step. A similar strategy had been proposed for non-stochastic Petri nets in [PRC97]. However, our experience showed, that this greedy chaining, even though it reduces the number of iterations of the reachability algorithm, often plays a minor role only.

3 Background

3.1 Properties of high-level model

Static properties: A model M consists of a finite ordered set of discrete state variables (SVs) $s_i \in S$, where each can take values from a finite subset of the naturals. Each state of the model is thus given as a vector $\vec{s} \in \mathbb{S} \subset \mathbb{N}^{|S|}$. Concerning the high-level model description by means of PNs or PAs, 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 PN-formalism, 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:

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

where $\overline{\mathcal{D}_l} = S \setminus \mathcal{D}_l$. Based on this, we define for each activity $l \in \mathcal{A}ct$ a projection function $\chi_l : \ \mathbb{N}^{|S|} \longrightarrow \mathbb{N}^{|\mathcal{D}_l|}$ which yields the sub-vector consisting of the dependent SVs only. We use the shorthand notation $\vec{s}_{d_l} := \chi_l(\vec{s}\,)$, where $\vec{s}_{d_l}$ is called the activity-local marking of state $\vec{s}$ with respect to activity l.
We have a reflexive and symmetric dependency relation $\mathcal{A}ct^{\mathcal{D}} \subseteq \mathcal{A}ct \times \mathcal{A}ct$. Two activities $l, k \in \mathcal{A}ct$ are called dependent if they share at least one SV, i.e. $(k, l) \in \mathcal{A}ct^{\mathcal{D}} \Leftrightarrow \mathcal{D}_k \cap \mathcal{D}_l \neq \emptyset$. Now the set of dependent activities for each activity l can be defined as $\mathcal{A}^{\mathcal{D}_l} := \{k \in \mathcal{A}ct \,|\, (l, k) \in \mathcal{A}ct^{\mathcal{D}}\}$. Note that according to this definition we have $l \in \mathcal{A}^{\mathcal{D}_l}$.
Dynamic properties: When an activity is executed, the model evolves from one state to another. The transition function $\delta : \mathbb{S} \times \mathcal{A}ct \longrightarrow \mathbb{S}$ 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, thus we write $\vec{s}^{\,\omega} := \delta(\ldots\delta(\delta(\vec{s}, \omega_1), \omega_2), \ldots, \omega_{|\omega|})$ where $\omega := (\omega_1, \ldots, \omega_{|\omega|}) \in \mathcal{A}ct^*$ and $\vec{s}^{\,\omega} \in \mathbb{S}$. If activity l is enabled in state $\vec{s}$ we write $\vec{s}[\triangleright l$. We also define the rate function $\eta : \mathbb{S} \times \mathcal{A}ct \times \mathbb{S} \longrightarrow \mathbb{R}^{\geq 0}$, which yields the transition rate at which the Markov chain moves from source to target state when a specific activity l occurs. During SG exploration, δ and η define 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 labelled transition system (SLTS), i.e. the underlying labelled Markov chain. The method described in this paper, as well as our implementation thereof, can handle not only purely Markovian models but also models with both, Markovian and immediate transitions (with priorities), but we decided not to describe this feature in order to keep the discussion simple. For each $l \in \mathcal{A}ct$ we partition T into sets of transitions with label l, where each state vector is reduced to the activity dependent markings:

$$T^l := \{(\vec{s}_{d_l}, l, \lambda, \vec{s}^{\,l}_{d_l}) \mid \vec{s}_{d_l} = \chi_l(\vec{s}\,) \wedge \ \vec{s}^{\,l}_{d_l} = \chi_l(\vec{s}^{\,l}\,) \wedge \ (\vec{s}, l, \lambda, \vec{s}^{\,l}) \in T\} \qquad (1)$$

For generating the sets of activity-local transitions T^l we will later 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 $\vec{s}$, we generate the set of successor states by executing only those enabled activities, which are also dependent on l:

$$\mathcal{A}^{D_l}_{\vec{s}^{\,l}} := \{k \in \mathcal{A}^{D_l} \mid \vec{s}^{\,l} [\triangleright k \ \wedge \ \vec{s}^{\,l}_{d_k} \notin \mathbb{E}_k\} \tag{2}$$

In eq. (2) the set $\mathbb{E}_k$ records the activity-local markings of states on which activity k was already tested in a previous step. Thus $\vec{s}^{\,l}_{d_k} \notin \mathbb{E}_k$ states that activity k was not yet tested on the activity-dependent marking of state $\vec{s}^{\,l}$.

3.2 Symbolic encodings of state graphs

Binary encodings of transitions: The value of a SV s_i can be encoded in binary form. For this purpose we define an injective encoding function $\mathcal{E}_i : \{0,\ldots,K_i\} \to \mathbb{B}^{n_i}$, where K_i is the maximum value of s_i and $n_i \geq \lceil \log_2(K_i+1) \rceil$. We define $n := \sum_{i=1}^{|S|} n_i$ which is the number of bits required for encoding the full state vector $\vec{s}$. For convenience, we define an encoding function for the full state vector $\mathcal{E}_S : \mathbb{N}^{|S|} \longrightarrow \mathbb{B}^n$, which is simply the combination of the individual ones. In a similar fashion one can encode the index of each activity label by an encoding function $\mathcal{E}_{\mathcal{A}ct}$ using $n_{\mathcal{A}ct}$ bits. This yields the following binary encoding scheme: $(\vec{s} \xrightarrow{l} \vec{s}^{\,l}) \mapsto \big(\mathcal{E}_{\mathcal{A}ct}(l), \mathcal{E}_S(\vec{s}), \mathcal{E}_S(\vec{s}^{\,l})\big)$ for each transition leading from source state $\vec{s}$ to target state $\vec{s}^{\,l}$ and labelled by l. The rate λ is unaccounted, since it will be stored in a terminal node of the DD.

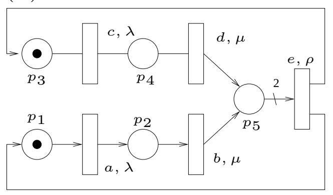
Zero-suppressed multi-terminal binary DDs (ZDDs): In a reduced ordered BDD, isomorphic subgraphs have been merged and don't care nodes[2] are eliminated. Zero-suppressed BDDs (ZBDDs) [Min93] are derivatives of BDDs for representing sparse sets efficiently. In ZBDDs, instead of eliminating don't-care nodes, one eliminates those non-terminal nodes whose 1-successor is the terminal 0-node. Analogously to ADDs we allow ZBDDs to have more than two terminal nodes and obtain zero-suppressed multi-terminal binary DDs (ZDDs). Standard arithmetic operators can be performed efficiently on the ZDD data structure with the help of a variant of Bryant's [Bry86] `Apply`-algorithm[3].

ZDD-based representation of SGs: Each transition of a SLTS is encoded by a Boolean vector whose bit positions correspond to the Boolean variables of the ZDD representing the SG. The symbolic representation of a SG T is a ZDD Z over the Boolean variables $\vec{\mathsf{a}}$, $\vec{\mathsf{s}}$ and $\vec{\mathsf{t}}$ where variables $\vec{\mathsf{a}}$ encode the activity label, variables $\vec{\mathsf{s}}$ encode the source state, and variables $\vec{\mathsf{t}}$ encode the target state of a transition. In the sequel we assume that the ZDD variables are ordered in the following way: At the first $n_{\mathcal{A}ct}$ levels from the root are the variables a_i, and on the remaining $2n$ levels we have the variables s_i and t_i in an interleaved fashion. This interleaved ordering of source and target bits is a commonly accepted heuristics for obtaining small BDD sizes which also works well for ZDDs.

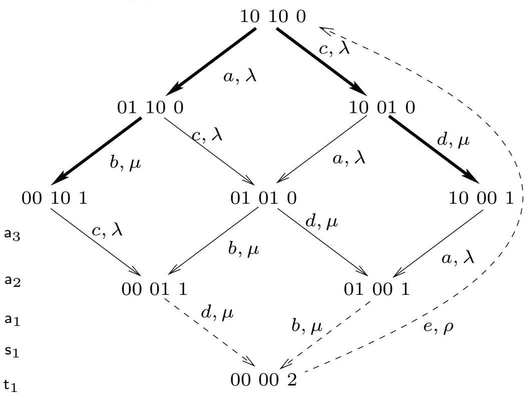
[2] A don't care node is a node whose 1- and 0-successors are identical.

[3] Our implementation is built on top of the CUDD package [Som], but we extend each DD by the set of variables on which it depends. This allows us to implement a new `Apply`-algorithm for *partially shared* ZDDs.

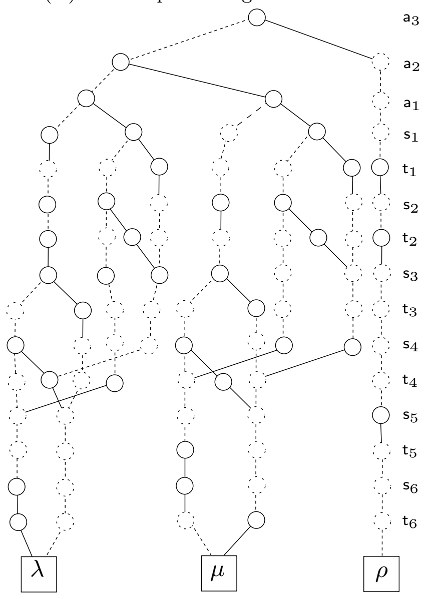
(A) A stochastic Petri net



(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

Fig. 1. From a SPN to the symbolic representation of its underlying SLTS

Unknown bounds for SVs: The values K_i are in general not known prior to SG generation. Contrary to ADDs, ZDDs have the nice feature that during SG generation and encoding one can allocate a new most significant bit for any SV s_i by simply adding a new Boolean variable to Z, i.e. without changing the structure of the DD's graph. Thus it is not necessary to know the maximum value K_i of SV variable s_i in advance, one simply introduces a new most-significant bit for s_i each time it is required, which does not slow down the SG exploration.
Example: Part (A) and (B) of Fig. 1 show a simple SPN and its underlying SLTS, where for the moment the bold, regular and dashed arrows of the SLTS have the same meaning (cf. Sec. 4.4). The Boolean encodings of the transitions of the SLTS as produced by function $\mathcal{E}_{\mathcal{A}ct}$ and $\mathcal{E}_S$ are specified in table (C). The 5 integer state variables of S are encoded by 6 Boolean variables, since only state variable 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.

4 Symbolic Activity-local State Graph Generation

After illustrating the main idea in sec. 4.1, sec. 4.2 covers the different elements of the activity-local scheme: explicit exploration and encoding of transitions, symbolic composition and re-initialization. Sec. 4.3 presents a new scheme for symbolic reachability analysis as required not only by the activity-local scheme presented here, but also by other BDD-based approaches [Sie02,LS02,Par02,KS02]. The final subsection discusses the correctness and completness of the activity-local scheme.

4.1 Main idea

The main idea of our approach is the explicit exploration of parts of the SG, where each detected transition is encoded and inserted into the respective *activity-local* ZDD. Each activity-local ZDD Z_l solely depends on the Boolean variables encoding the SVs of the set $\mathcal{D}_l$ and it represents only transitions induced by activity l (at the end of SG exploration, Z_l encodes all transitions of T^l, cf. eq. (1)). We define the sets of dependent Boolean source and target variables, and the sets of their independent counterparts:

$$\mathsf{D}_l := \{\vec{\mathsf{s}}^{\,i}, \vec{\mathsf{t}}^{\,i} | s_i \in \mathcal{D}_l\} \qquad \mathsf{I}_l := \{\vec{\mathsf{s}}^{\,i}, \vec{\mathsf{t}}^{\,i} | s_i \in \overline{\mathcal{D}_l}\} \tag{3}$$

In this equation, $\vec{\mathsf{s}}^{\,i}$ and $\vec{\mathsf{t}}^{\,i}$ denote those Boolean variables which encode the value of the dependent SV s_i in the source and target state of a transition $(\vec{s}, l, \lambda, \vec{t})$. After the generation and encoding of the individual transitions, the overall SG is obtained by symbolic composition of the ZDDs. A symbolic reachability analysis must follow, so that the obtained set of potential transitions is limited to the set of reachable ones. Several rounds of explicit SG generation, symbolic composition and symbolic reachability analysis may be required.

4.2 The Scheme

The top-level algorithm of the activity-local scheme is shown in Fig. 2.C. Lines 1 – 4 contain the initialization:

1. $\mathcal{A}_{\vec{s}^{\,\epsilon}}^{D_\epsilon}$ is the set of activities enabled in the initial state.
2. The ZBDDs E_k encode the sets $\mathbb{E}_k$ of activity-local markings of states on which activity k was already tested.
3. The $StateBuffer$ holds tuples of states and activity sets $(\vec{s}^{\,l}, \mathcal{A}_{\vec{s}^{\,l}}^{D_l})$ for explicit exploration.

(A) Encoding and insertion of transitions into Z_l

```
(0)  EncodeTransitions()
(1)    while ($TransBuffer \neq empty$) do
(2)      $TransBuffer \rightsquigarrow (\vec{s}, l, \lambda, \vec{s}^{\,l})$
(3)      $\mathcal{A}^{D_l}_{\vec{s}^{\,l}} := \emptyset$
(4)      for $k \in \mathcal{A}^{D_l}$ do
(5)        if $\mathcal{E}(\vec{s}^{\,l}_{d_k}) \notin \mathsf{E}_k \wedge \vec{s}^{\,l} [\triangleright k$ then
(6)          $\mathcal{A}^{D_l}_{\vec{s}^{\,l}} := \mathcal{A}^{D_l}_{\vec{s}^{\,l}} \cup \{k\}$
(7)        $\mathsf{E}_k := \mathsf{E}_k \cup \vec{s}^{\,l}_{d_k}$
(8)      end
(9)      if $\mathcal{A}^{D_l}_{\vec{s}^{\,l}} \neq \emptyset$ then
(10)       $(\vec{s}^{\,l}, \mathcal{A}^{D_l}_{\vec{s}^{\,l}}) \rightsquigarrow StateBuffer$
(11)     $\mathsf{Z}_l := \mathsf{Z}_l + \mathcal{E}(\vec{s}_{d_l}, \lambda, \vec{s}^{\,l}_{d_l})$
(12)   end
```

(B) Exploration of states, where $\vec{s} \notin \mathsf{Z}_l$

```
(0)  ExploreStates()
(1)    while ($StateBuffer \neq empty$) do
(2)      $StateBuffer \rightsquigarrow (\vec{s}^{\,l}, \mathcal{A}^{D_l}_{\vec{s}^{\,l}})$
(3)      for $k \in \mathcal{A}^{D_l}_{\vec{s}^{\,l}}$ do
(4)        $\vec{s}^{\,lk} := \delta(\vec{s}^{\,l}, k)$
(5)        $\lambda := \eta(\vec{s}^{\,l}, k, \vec{s}^{\,lk})$
(6)        if $\vec{s}^{\,l} \neq \vec{s}^{\,lk}$ then
(7)          $(\vec{s}^{\,l}, k, \lambda, \vec{s}^{\,lk}) \rightsquigarrow TransBuffer$
(8)      end
(9)    end
```

(C) Main routine for the symbolic activity-local scheme

```
(0)  ExploreStateGraph()
(1)    $\forall k \in \mathcal{A}ct$: if $\vec{s}^{\,\epsilon} [\triangleright k$ then $\{k\} \rightsquigarrow \mathcal{A}^{D_\epsilon}_{\vec{s}^{\,\epsilon}}$
(2)    $\forall k \in \mathcal{A}ct : \mathsf{E}_k := \mathcal{E}(\vec{s}^{\,\epsilon}_{d_k})$
(3)    $(\vec{s}^{\,\epsilon}, \mathcal{A}^{D_\epsilon}_{\vec{s}^{\,\epsilon}}) \rightsquigarrow StateBuffer$
(4)    $TransBuffer = \emptyset$
(5)    do
(6)      do
(7)        ExploreStates()
(8)        EncodeTransitions()
(9)      until $StateBuffer = \emptyset$
(10)     InitiateNewRound()
(11)   until $StateBuffer = \emptyset$
(12)   $\mathsf{Z}_T := (\sum_{l \in \mathcal{A}ct} \mathsf{Z}_l \cdot \mathbb{1}_l \cdot \mathsf{A}_l) \cdot \mathsf{Z}_R$
```

(D) Symbolic composition, symbolic reachability analysis and refill of $StateBuffer$

```
(0)  InitiateNewRound()
(1)    $\mathsf{Z}_R$ := ReachabilityAnalysis()
(2)    for $k \in \mathcal{A}ct$ do
(3)      Temp := $\mathsf{Z}_R \setminus \mathsf{E}_k$
(4)      while Temp $\neq \emptyset$ do
(5)        Temp $\rightsquigarrow \vec{s}$
(6)        if $\vec{s} [\triangleright k$ then
(7)          $(\vec{s}, \{k\}) \rightsquigarrow StateBuffer$
(8)        Temp := Temp $\setminus \{\mathcal{E}(\vec{s}_{d_k})\}$
(9)      end
(10)   end
```

Fig. 2. Algorithms for the activity-local scheme

4. The $TransBuffer$ holds explicitly generated transitions $(\vec{s}^{\,l}, k, \lambda, \vec{s}^{\,lk})$ to be encoded and inserted into the respective activity-local ZDDs.

In the inner loop (lines 6 – 9) 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 fixed point is reached, the re-initialization routine InitiateNewRound, as called in line 10, searches for new states triggering new model behaviour. If such states exist, a new round of explicit SG exploration and encoding will follow, i.e. one re-enters the outer loop (lines 5 – 11). Otherwise the SG generation procedure is finished and the set of all reachable transitions is computed in line 12, where Z_R is the set of reachable states returned by symbolic reachability analysis (which is carried out by one of the routines of Fig. 3 and called in procedure InitiateNewRound).

Explicit SG generation: The explicit SG generation and encoding is realized with the help of two complementary procedures, called EncodeTransitions and ExploreStates (Fig. 2.A and 2.B). In line 2 of algorithm EncodeTransitions a transition is read from the $TransBuffer$, and in lines 3 – 8 the set $\mathcal{A}^{D_l}_{\vec{s}^{\,l}}$ of dependent activities enabled in the target state is determined. The state/activity-set tuple for further exploration is inserted into the $StateBuffer$ in line 10, and the activity-local encoding of the current transition is then inserted into ZDD Z_l. The complementary exploration routine ExploreStates for executing the set of dependent activities $\mathcal{A}^{D_l}_{\vec{s}}$ on a state $\vec{s}^{\,l}$ works as shown in Fig. 2.B. In line 2, a state together with a set of activities is read from the $TransBuffer$. For each activity k from that set, the successor state $\vec{s}^{\,lk}$ and the corresponding rate λ are computed (lines 4 and 5). Each resulting transition is inserted into the $TransBuffer$ (line 7), provided it is not a self-loop. Executing procedures ExploreStates and EncodeTransitions alternatingly, a fixed point will be reached, where EncodeTransitions has been executed and the $StateBuffer$ is still empty. At that point, one has visited all states reachable from the initial state through sequences of dependent activities. What follows next, is the composition of the activity-local markings of the states reached so far, in order to obtain also states resulting from the interleaved execution of independent activities.

Symbolic composition: At the end of an exploration and encoding phase we have $|\mathcal{A}ct|$ ZDDs Z_l each of which encodes T^l (cf. eq. (1)) (already complete or still incomplete, depending on the number of rounds completed). Before composition can take place, Z_l needs to be supplemented by the activity's set of independent Boolean variables I_l (cf. eq. (3)), yielding the symbolic representation of the set of potential transitions induced by activity l. When activity l takes place, the SVs $s_i \in \overline{\mathcal{D}_l}$ do not change their values, which is expressed by the pairwise identity over the Boolean s- and t- variables of activity l's set of independent SVs I_l: $\mathbb{1}_l := \bigwedge_{\vec{\mathsf{s}}^i \in \mathsf{I}_l} \bigwedge_{j=1}^{n_i} (\mathsf{s}_j^i \leftrightarrow \mathsf{t}_j^i,)$ During composition, the activity-local ZDDs are combined in order to obtain the transition relation of the overall model:

$$\mathsf{Z}_T := \sum_{l \in \mathcal{A}ct} \mathsf{Z}_l \cdot \mathbb{1}_l \cdot \mathsf{A}_l \tag{4}$$

In the above equation, A_l represents the binary encoding of activity label l. The ZDD Z_T thus constructed encodes a set of potential transitions. Therefore, at this point it is necessary to perform symbolic reachability analysis.

Re-initialization: So far we have not considered states resulting from the combined execution of independent activities for further explicit exploration (states of this type are a result of symbolic composition, cf. eq. (4)). Since such states may trigger new model behaviour, algorithm InitiateNewRound (Fig. 2.D) checks this and inserts the corresponding state-activity tuples into the $StateBuffer$. The algorithm first calls the procedure ReachabilityAnalysis (cf. Sec. 4.3) which performs symbolic composition and reachability analysis in order to return the set of reachable states Z_R. In lines 2 – 10 the algorithm determines those reachable states on which a given activity has not yet been tested, since these need to be examined further. The obtained pairs of states and enabled activities are inserted into the $StateBuffer$ (line 7), yielding the input for the next round of explicit SG exploration and encoding. If the $StateBuffer$ is still empty after the execution of InitiateNewRound, the activity-local scheme has reached a fixed point and a symbolic representation of the complete SG has been generated.

4.3 Symbolic reachability analysis

We now discuss two variants of a reachability algorithm as required by algorithm InitiateNewRound (line 1 of Fig. 2.D). Line 1 of the algorithm of Fig. 3.A computes ZBDD Z_T which represents the set of *potential* transitions (for simplicity, activity-labels are omitted and rates are dropped). The algorithm employs

(A) Bfs. symbolic reachability analysis as proposed by [PRCB94,Sie02]

```
(0)  ReachabilityAnalysis()
(1)    $Z_T := \sum_{l \in \mathcal{A}ct} Z_l \cdot \mathbb{1}_l$
(2)    $Z_R := \mathcal{M}(\vec{t}, \mathcal{E}(\vec{s}^{\epsilon}))$
(3)    $Z_U := \mathcal{M}(\vec{s}, \mathcal{E}(\vec{s}^{\epsilon}))$
(4)    do
(5)      $Z_{tmp} :=$ Abstract($Z_T \wedge Z_U, \vec{s}, \vee) \setminus Z_R$
(6)      $Z_R = Z_R \vee Z_{tmp}$
(7)      $Z_U := Z_{tmp}\{\vec{s} \leftarrow \vec{t}\}$
(8)    until $Z_U = \emptyset$
(9)    $Z_R := Z_R\{\vec{s} \leftarrow \vec{t}\}$
(10)   return $Z_R$
```

(B) Sequential activity-oriented symbolic reachability analysis organised as quasi-dfs-traversal

```
(0)  ReachabilityAnalysis()
(1)    $Z_R := \emptyset$
(2)    $Z_U := \mathcal{M}(\vec{s}, \mathcal{E}(\vec{s}^{\epsilon}))$
(3)    $\forall k \in \mathcal{A}ct : \widetilde{Z_k} := Z_k \cdot \mathbb{1}_k$
(4)    do
(5)      $Z_R := Z_R \vee Z_U$
(6)      for $k \in \mathcal{A}ct$ do
(7)        $Z_{tmp} :=$ Abstract($\widetilde{Z_k} \wedge Z_U, \vec{s}, \vee) \setminus Z_R$
(8)        $Z_U := Z_U \vee Z_{tmp}\{\vec{s} \leftarrow \vec{t}\}$
(9)      end
(10)     $Z_U := Z_U \setminus Z_R$
(11)   until $Z_U = \emptyset$
(12)   return $Z_R$
```

Fig. 3. Pseudo-code of symbolic reachability analysis variants

another three ZBDDs: ZBDD Z_U for representing the set of *unexplored states*, ZBDD Z_R for representing the set of *reached states*, and ZBDD Z_{tmp} which represents the set of states detected in the current iteration. The former two ZBDDs are initialized with the symbolic representation of the initial state $\vec{s}^{\epsilon}$ (constructed by the function $\mathcal{M}$ (lines 2 and 3)). The standard breadth-first-search (bfs) symbolic reachability analysis is realized by the loop of lines 4 − 8. The conjunction of Z_U (unexplored states) and Z_T (potential transitions) delivers all transitions emanating from the states of Z_U. The subsequent abstraction of the source states as encoded by variables $\vec{s}$ yields the set of newly reached target states stored as Z_{tmp} (line 5). One may consider this step as set-oriented and "parallel", since Z_U may represent more than one state and one obtains all successor states at once. We propose now the following improvements (cf. Fig. 3.B):

(i) replace the *"parallel"* scheme of line 5 Fig. 3.A by an *activity-wise* scheme,
(ii) update the set of unexplored states as soon as possible (a.k.a. greedy chaining).

If Z_U of Fig. 3.B were not updated with the newly reached states in line 8, but outside the inner `for`-loop, one would obtain the same number of iterations of the main (outer) `do-until` loops for both algorithms. The activity-wise iteration of Fig. 3.B combined with an early update of Z_U realizes a set-oriented *quasi* depth-first-search (dfs.) scheme, since *all* successor states of Z_U reachable by the same activity k are generated in one step. Consequently this procedure leads to a significant reduction of the number of iterations of the main (outer) `do-until loop`. In Sec. 5 we will refer to this reduction by the ratio r_{iter}.

4.4 Comments on the activity-local generation scheme

Correctness of the generated transitions: Our algorithm starts from the initial state. For a given state $\vec{s}^{\,l}$ reached by activity l, the algorithm explores activity k if and only if

1. l and k share dependent SVs,
2. k is enabled in $\vec{s}^{\,l}$,

3. k has not yet been explored from any other state $\vec{t}$ whose projection to the set of dependent SVs $\mathcal{D}_k$ (activity-local marking) is identical to that of $\vec{s}^{\,l}$.

Instead of encoding a detected transition $(\vec{s}, l, \lambda, \vec{s}^{\,l})$ as a whole, the algorithm only encodes the SVs in the set $\mathcal{D}_l$. The SVs outside the set $\mathcal{D}_l$ may take arbitrary values, but they must remain stable upon execution of activity l, which is expressed by the multiplication with $\perp_l$. This has the effect that a single detected transition is encoded as a possibly huge set of potential transitions. Symbolic reachability analysis reduces this set of potential transitions to the transitions which are actually reachable from the initial state, yielding only legal transitions.
Completeness of the generation scheme: According to the diamond property for two independent activities l and k (here $(l,k) \notin \mathcal{A}ct^{\mathcal{D}}$), the order of their execution is without significance, i.e. one may execute these activities independently on a given source state $\vec{s}$. The target state of the combined sequential execution of either kl or lk can then be obtained by combining the activity-dependent markings as contained in the intermediate states $\vec{s}^{\,l}$ and $\vec{s}^{\,k}$. This property also holds for sequences of pairwise independent activities, yielding the well-known trace equivalence relation on the set of sequences of executed activities. Under this equivalence relation two sequences $\omega, \rho \in \mathcal{A}ct^*$ are considered equivalent if and only if they can be obtained from each other by swapping the execution order of adjacent independent activities [God95]. Consequently one only needs to generate the sequences of dependent activities explicitly. All other states can be obtained by a composition of the kind mentioned above. This is exactly the functionality of the algorithms presented in Fig. 2.
Example: We consider again the example depicted in Fig. 1. We may for the moment ignore the rate information, since it is irrelevant for the following discussion. Starting from the initial state (10100), the activity-local scheme will explore those transitions explicitly which are drawn by fat arrows in the figure. As an example, transition 10100 $\xrightarrow{a}$ 01100 will be explored and then encoded in the activity-local ZDD Z_a of activity a as 10*** $\longrightarrow$ 01***, where the symbol * denotes a don't care, since the respective variables are not visible within Z_a (only p_1 and p_2 belong to the set of dependent SVs of activity a). The transitions drawn by regular arrows are the ones which are generated during the composition of the activity-local ZDDs, which can be seen as a cross product construction followed by reachability analysis as realized by one of the algorithms of Fig. 3 as called by procedure InitiateNewRound. We will now explain why the transitions drawn as dashed arrows in the figure are not generated during the first round of exploration. Consider, for example, transitions caused by activity d: In the first round the algorithm explicitly generates the transition 10010 $\xrightarrow{d}$ 10001, which is encoded in the activity-local ZDD of activity d as ***10 $\longrightarrow$ ***01. The cross product construction yields any transition +++10 $\xrightarrow{d}$ +++01 (where the +-positions are arbitrary *but stable*), but it does not yield the dashed transition 00011 $\xrightarrow{d}$ 00002. During procedure InitiateNewRound, one detects that state 00011 is reachable and that activity d has not yet been tested in states of the type ***11. Therefore the tuple (00011,d) will be inserted into

Fault-tolerant Multiprocessor (FTMP) [SM92]

N	*states*	*trans*	$trans_e$
2	$2.5693E5$	$1.6978E6$	688
3	$1.2408E8$	$1.1513E9$	1,548
4	$5.5039E10$	$6.6113E11$	2,752
5	$2.3549E13$	$3.4847E14$	4,300
6	$9.9082E15$	$1.7463E17$	6,192

Kanban System (KS) [CT96]

N	*states*	*trans*	$trans_e$
5	$2.5464E6$	$2.4460E7$	1,860
6	$1.1261E7$	$1.1571E8$	4,116
7	$4.1645E7$	$4.5046E8$	8,232
8	$1.3387E8$	$1.5079E9$	15,192
9	$3.8439E8$	$4.4746E9$	26,280
10	$1.0059E9$	$1.2032E10$	43,120

Courier Protocol (CP) [WL91]

N	*states*	*trans*	$trans_e$
4	$9.7102E6$	$5.7005E7$	142
5	$3.2405E7$	$1.9983E8$	206
6	$9.3302E7$	$5.9818E8$	289
7	$2.3965E8$	$1.5858E9$	394
8	$5.6182E8$	$3.8166E9$	524

Tandem Queueing Network (TQN)[HMKS99]

N	*states*	*trans*	$trans_e$
127	$3.2640E4$	$1.1328E5$	32,639
128	$3.3153E4$	$1.1507E5$	33,152
255	$1.3082E5$	$4.5594E5$	130,815
256	$1.3184E5$	$4.5920E5$	131,840
511	$5.2378E5$	$1.8294E6$	$1.050E6$
512	$5.2583E5$	$1.8365E6$	$1.054E6$

Flexible Manufacturing System (FMS) [CT93]

N	*states*	*trans*	$trans_e$
6	$5.3777E5$	$4.2057E6$	434
7	$1.6394E6$	$1.3553E7$	616
8	$4.4595E6$	$3.8534E7$	840
9	$1.1058E7$	$9.9075E7$	1,110
10	$2.5398E7$	$2.3452E8$	1,430

Table 1. Model specific data for the various case studies

the $StateBuffer$ at this point, and this dashed transition (as well as the other two dashed transitions) will be explored in the second round.

5 Empirical Evaluation

In order to evaluate the proposed innovations, we analyzed five models which are commonly employed as benchmarks in the literature. Table 1 gives the sizes of their SGs, i.e. the number of states (*states*), number of transitions (*trans*), and the number of transitions explicitly explored under our activity-local scheme ($trans_e$). The experiments carried out with our implementation, as well as those carried out with the tools CASPA [KSW04] and PRISM [Pri], were run on a Pentium 4 (3 GHz) with 1 GByte of RAM and a Linux OS. All run-time results were averaged from 100 runs. In order to simplify the comparison, we decided to present most of our results in the form of ratios, where the respective figures are always normed to the new activity-local scheme. Ratios > 1 indicate an advantage of the innovations developed in this work, and ratios < 1 indicate their disadvantage. The figures which serve as norm concerning run-time and space complexity are given in Table 2.A, i.e. all other figures can be obtained by simply scaling these figures with the respective ratios.Within the Möbius modelling framework [DCC+02] the local exploration of submodel SGs in isolation is not feasible, due to the nature of the *Join* model composition formalism, and the fact, that one can not calculate SV capacities in advance. Thus this framework is highly suited for implementing the activity-local approach. Our implementation consists of three main modules:

1. A module for the explicit SG generation (derived from the standard SG generator of Möbius) which constitutes the interface between the symbolic engine and Möbius (algorithm of Fig. 2.B).
2. The symbolic engine (mainly algorithm (A) and (D) of Fig. 2, in combination with one of the algorithms of Fig. 3).
3. A ZDD-library (based on the CUDD-package [Som]), which contains the new algorithms for manipulating *partially shared ZDDs* and implements a C++ wrapper for them.

(A) ZDD-based scheme

N	n	nodes			t_g in
		Z_R	Z_T	*peak*	sec
FTMP					
2	142	256	5,792	7.483 $E4$	0.277
3	196	610	16,225	2.546 $E5$	1.179
4	262	1044	30,892	6.201$E5$	3.268
5	326	1556	49,845	9.742$E5$	7.257
6	390	2146	73,002	2.278$E6$	14.082
KS					
5	96	163	2751	8.3250 $E4$	0.39
6	96	215	3,704	1.40709 $E5$	0.84
7	96	273	4,758	2.22282 $E5$	1.66
8	128	341	6,020	3.33193 $E5$	3.26
9	128	416	7,414	5.44593 $E5$	6.76
10	128	497	8,933	6.60963 $E5$	9.99
CP					
4	144	271	3,490	4.168 $E5$	3.781
5	144	353	4,715	7.303 $E5$	5.871
6	144	433	5,941	1.212 $E6$	8.778
7	144	515	7,184	1.943 $E6$	13.493
8	166	603	8,487	2.964 $E6$	20.128
TQN					
127	30	22	204	5.922 $E4$	2.338
128	34	18	201	5.398 $E4$	2.376
255	34	25	235	2.354 $E5$	12.353
256	38	20	228	2.647 $E5$	12.729
511	38	28	266	8.576 $E5$	61.146
512	42	22	255	8.143 $E5$	62.907
FMS					
6	96	559	17,557	1.786 $E5$	0.521
7	98	756	26,567	2.845 $E5$	0.852
8	118	992	38,610	4.323 $E5$	1.409
9	120	1,262	53,740	6.260 $E5$	2.062
10	124	1,566	72,341	8.724 $E5$	3.067

(B) ADD-based scheme

r_R	r_T	r_{pk}	r_{time}
2.01	2.38	2.36	1.81
1.99	2.40	2.31	1.59
2.00	2.41	2.31	1.80
1.99	2.41	2.90	1.53
1.99	2.41	2.20	1.44
1.97	2.29	2.50	2.27
1.81	2.13	2.33	1.96
1.68	2.00	2.20	1.78
2.14	2.44	2.76	2.26
2.01	2.32	2.35	1.69
1.92	2.23	2.60	1.70
2.11	2.65	2.80	2.03
1.93	2.43	2.67	1.97
1.82	2.29	2.59	2.06
1.74	2.2	2.49	1.78
1.98	2.48	2.86	2.00
0.41	1.40	1.96	1.14
1.06	1.68	2.60	1.33
0.40	1.40	1.59	1.11
1.05	1.68	1.72	1.18
0.39	1.40	1.51	1.05
1.05	1.68	1.95	1.17
1.86	2.29	2.34	2.23
1.72	2.14	2.20	2.08
2.04	2.50	2.54	2.24
1.95	2.41	2.46	2.62
1.86	2.32	2.38	2.26

Table 2. Empirical comparison of ADDs and ZDDs for various case studies

Comparing ADD and ZDD data structures: Table 2 illustrates the difference between the ADD- and ZDD-based encoding schemes with respect to their space and time complexity for the different models. The first column gives the model scaling parameter N, the second gives the total number of Boolean variables required for encoding all SVs, where we encoded each SV by a minimum number of bits. In practice such an allocation strategy is not feasible for ADDs, due to the lack of a priori knowledge of the maximum value K_i taken by SV s_i, which means that practical figures would be even more favourable for our ZDD approach. In case of ZDDs, pre-allocation of Boolean variables is unnecessary, since skipped variables are interpreted as being 0-assigned.
Table 2.A gives the number of nodes required for representing the set of reachable states (encoded by ZBDD Z_R), the transition system (encoded by ZDD Z_T), as well as the peak number of nodes (*peak*) as allocated during the process of symbolic SG construction. In our implementation, since we employed the CUDD-package, each node consumes 16 bytes of memory. Column t_g contains the SG generation time in seconds. In Table 2.B the ADD-based activity-local scheme is compared to the ZDD-based version, by providing ratios of memory consumption concerning Z_R, Z_T and the peak number of nodes (r_{pk}). The comparison is rounded by finally giving the ratio of the construction times r_{time}. As illus-

(A) ZDD-based reachability schemes within Möbius

N	r_{time}	r_{pk}	r_{c2ut}	r_{c2ct}
FTMP				
2	0.711	0.366	0.764	0.698
3	2.668	0.280	3.571	3.429
4	4.868	0.248	6.754	5.714
5	11.947	0.294	22.065	17.240
6	73.533	0.217	10.808	10.158
CP				
4	2.272	0.291	2.437	3.756
5	2.420	0.313	2.564	3.085
6	2.670	0.411	2.988	3.494
7	3.484	0.498	4.387	4.912
8	6.018	0.649	7.197	8.005

N	r_{time}	r_{pk}	r_{c2ut}	r_{c2ct}
KS				
6	3.456	0.599	4.558	5.262
7	5.196	0.658	7.076	8.385
8	5.391	0.743	6.988	8.412
9	6.534	0.771	9.708	11.897
10	9.693	0.912	13.548	16.737
FMS				
6	1.246	0.282	1.672	1.560
7	1.626	0.268	2.195	2.015
8	2.438	0.261	3.023	2.750
9	3.605	0.253	4.608	4.214
10	5.143	0.245	6.208	5.595

(B) ADD-based reachability schemes within CASPA

t_{old}	t_{new}	pk_{old}	pk_{new}	r_{iter}
KS				
0.690	0.333	47,538	14,544	4.474
1.310	0.531	76,384	18,439	4.500
2.424	0.892	116,978	25,840	4.520
3.870	1.560	168,921	34,641	4.536
6.710	2.570	248,749	48,034	4.548
FMS				
1.612	0.948	95,412	60,337	4.083
3.366	1.830	167,392	93,834	4.071
7.299	3.169	287,628	136,405	4.063
12.910	5.700	473,845	197,777	4.056
23.420	8.670	728,265	268,851	4.050

Table 3. Empirical comparison of the two variants of symbolic reachability analysis

trated by the ratios for the various case studies, the use of ZDDs reduces memory consumption. As a a consequence of smaller DD sizes, the caching behaviour of the ZDD-based scheme is much better. Thus the improvment in run-time is not really surprising.
The TQN model [HMKS99] constitutes a very interesting case study. We specified this model as a SPN consisting of 3 places, where two may contain the number of tokens specified by the scaling parameter N (let us say places 1 and 2), and the remaining place (place 3) contains either one or zero tokens. Consequently, for $N = 2^{n_i} - 1$ the model uses a very dense Boolean enumeration scheme, where n_i is the number of bits used for encoding place $i \in \{1, 2\}$. As we expected, and as supported by the experimental data, in these cases the space requirement of the ADD-based representation of the reachable states Z_R is better (see col. r_R of Table 2.B). If N is a power of two, the enumeration scheme is much sparser and a different picture has to be drawn. Concerning the symbolic representation of the SG Z_T, it is interesting to note that ZDDs are always more compact (see col. r_T of Table 2.B). Furthermore, the ZDD-based scheme maintains its run-time advantage in both cases, which is significant, since the TQN-model is a worst case scenario concerning the number of transitions to be explicitly explored (cf. col. $trans_e$ in Table 1).

Assessment of the new reachability analysis algorithm: For most case studies, the number of explicitly explored and encoded transitions ($trans_e$) under the activity-local scheme is very low (see Table 1). Therefore, under this scheme, similar to the fully symbolic approaches, most of the execution time is consumed by symbolic reachability analysis. The precise portion of time differs, of course, for different models. For instance, for the KS and FTMP model one spends about 70% on reachability analysis, whereas for the FMS and CP models symbolic reachability analysis accounts for 99% of the run-time. As a consequence, most of the CPU time is spent in routines for manipulating the DD structures. Profiling reveals that a dominant fraction of the run-time, between 35% and 68%, is spent in the CUDD-functions *UniqueInter* and *CacheLookup*, where other functions consume less than 10%. *UniqueInter* delivers either an existing node found in the "unique table", or a newly allocated node. The *Cache-Lookup* function accesses the "computed table" in order to fetch results from previous recursions of the `Apply`- or `Abstract`-algorithm. Table 3 compares standard bfs. to the new quasi-dfs. algorithm (see Fig. 3). The data is based on the run-times, the peak memory requirements (*pk*), and on the number of calls to *UniqueInter* (c2ut) and *CacheLookup* (c2ct). As before we only give ratios by norming everything to the figures of the new quasi-dfs. variant. Table 3.A shows the figures for our ZDD-based implementation as realized within Möbius, where the new scheme produces fewer calls to *UniqueInter* and *CacheLookup*, making it substantially faster. But on the other hand it consumes more memory as indicated by r_{pk}. This increase of the peak memory consumption seems to be closely related to the sparseness of the ZDDs, since during ZDD manipulation many nodes are allocated, which can be eliminated for the final representation of the SG. In contrast, using ADDs for SG representation, the new scheme is not only faster, but also consumes less memory. We can report this result not only for the ADD-based experiments carried out with our Möbius implementation, but also for experiments carried out with the tool CASPA, whose results are listed in Table 3.B. This table shows results as obtained from employing (i) the standard bfs. reachability analysis (*old*) and (ii) the new quasi-dfs. reachability analysis (*new*) within the tool CASPA, which is based on ADDs. Even though the number of iterations of the outer `do-until` loop of the algorithm (Fig. 3.A) is reduced by a factor of about 4 (cf. r_{iter}, Table 3.B) under the new scheme, the run-times only halve (cf. col. 1 and 2), and the peak memory requirement is reduced by a factor of around 2 to 3.[4] The moderate speed-up might be a consequence of the very compact encodings of each state by CASPA, since it employs a dense enumeration scheme of submodel states, leading to much "flatter" DD-structures, i.e. DDs with fewer Boolean variables, than our Möbius implementation and PRISM. Thus the quasi-dfs. scheme becomes more advantageous concerning run-time and space complexity, the larger the generated DDs are. This is not only supported by the figures produced by CASPA, but also by the results obtained for the FTMP model (cf. Table 3.A) which required the largest number of Boolean variables for representing the SG symbolically.

Assessment of the activity-local scheme: We now compare our implementation of the activiy-local scheme to the MDD- and Kronecker-based approach of [DKS04] because our implementation is within Möbius and – similar to [DKS04] – uses the Möbius high-level model specification and the standard next-state function of Möbius for explicit exploration. We also compare to PRISM (and tested the new reachability scheme within CASPA) because these tools are both based on the CUDD library which we also use. Since only PRISM, similar to our own implementation, allows a freely chosen ordering of the Boolean variables and also encodes states in exactly the same way as we do, the focus of the comparison in Table 4.C is on PRISM rather than CASPA. We decided not to compare to the tool SMART and the results of [Min04], since there would be

[4] We decided to give absolute run-times and peak numbers of nodes, in order to enable the reader to compare the activity-local approach also to CASPA. An explicit comparison will be omitted, since a comparison with the PRISM tool seems to be more appropriate (cf. Sec. 5: Assessment of the activity-local scheme).

(A) Comparison to [DKS04]

	N	r_{mem}	r_{time}
FTMP	2	1.14	4.69
	3	1.08	21.29
	4	1.06	61.20
	5	1.05	180.52
	6	0.36	372.82
CP	4	7.32	1.15
	5	12.26	3.53
	6	19.95	9.55
	7	31.57	25.72
	8	48.08	51.67

(B) Data produced by PRISM

	N	n_{PRISM}	*peak*	*iter*
KS	5	96	59,731	71
	6	96	93,464	85
	7	96	135,514	99
	8	128	246,750	113
	9	128	333,388	127
	10	128	437,910	141
FMS	6	110	226,441	49
	7	110	348,540	57
	8	140	679,426	65
	9	140	971,954	73
	10	140	1,359,552	81

(C) Comparing the activity-local approach to PRISM

		ADD based SG representation							3. ZDD + quasi-dfs. scheme			
		nodes		1. bfs. scheme		2. quasi-dfs. scheme						
	N	M_R	M_T	r_{pk}	r_{time}	r_{pk}	r_{iter}	r_{time}	r_R	r_T	r_{pk}	r_{time}
KS	5	321	6,308	0.051	0.060	0.29	2.96	0.88	1.97	2.29	0.72	2.00
	6	389	7,876	0.045	0.065	0.29	3.04	0.90	1.81	2.13	0.66	1.77
	7	458	9,521	0.041	0.031	0.28	3.09	0.81	1.68	2.00	0.61	1.44
	8	731	14,698	0.037	0.031	0.27	3.14	0.68	2.14	2.44	0.74	1.53
	9	837	17,196	0.034	0.050	0.26	3.18	0.69	2.01	2.32	0.61	1.17
	10	952	19,877	0.031	0.047	0.25	3.20	0.81	1.92	2.23	0.73	1.34
FMS	6	1,039	40,274	0.11	0.42	0.54	2.33	2.56	2.47	4.45	1.27	5.69
	7	1,298	56,853	0.08	0.46	0.56	2.48	3.40	2.29	4.55	1.23	7.07
	8	2,022	96,647	0.06	0.77	0.62	2.50	7.32	2.79	5.57	1.57	16.38
	9	2,455	129,644	0.05	0.68	0.63	2.52	6.91	2.62	18.09	1.55	18.09
	10	2,907	167,798	0.04	0.74	0.65	2.53	10.39	2.49	5.61	1.56	23.51

Table 4. Comparing the activity-local scheme and other symbolic methods

neither a common implementation framework nor a common data structure.

In order to compare our full scheme (activity-local + quasi-dfs. reachability analysis + ZDDs) to the symbolic approach of [DKS04], identical Möbius model specifications for the FTMP and CP models were employed. Once again, ratios for run-time[5] and memory consumption are provided, where everything is normed to the figures of the activity-local scheme. As an example, the last entry in the fifth row of Table 4.A states that our activity-local approach is 372.82 times faster. The fact that the activity-local approach is significantly faster than the MDD-based approach shows that the partial-order style strategy of exploring only paths of dependent activities pays off, especially for models without strongly modular structure (cf. col. *trans* and $trans_e$ of Table 1). Furthermore the memory requirement for storing the set of reachable states is better as well (r_{mem}), except in case of the FTMP model with $N = 6$.

Table 4.B gives the basic model data of the KS and FMS model when executed by PRISM, including the number of Boolean variable required for encoding the transitions (n_{PRISM}), as well as the number of iterations (*iter*) as required by the standard bfs reachability analysis (basically the `do-until` loop of algorithm Fig. 3.A). For the KS model our implementation encodes the model in exactly the same way as PRISM does, consequently the generated ADDs are identical. But in case of the FMS model, we employed a slightly different model, due to the

[5] The results of [DKS04] were obtained on an AMD Athlon 2400. Furthermore, the timing information includes time for state lumping, but since this is below 0.3% of the overall time we can safely neglect it.

different elimination of immediate transitions. As a consequence of employing fewer SVs, we were able to encode each state by a smaller number of Boolean variables, (see col. n_{PRISM} of Table 4.B for details), leading already to slightly smaller DD structures, whose sizes are given in col. M_R and M_T of Table 4.C. In order to evaluate the different aspects of the work presented here, we chose to investigate the activity-local scheme in three different settings, comparing each one to PRISM:

1. In the first setting we combined the activity-local scheme with ADDs and standard bfs. reachability analysis.
2. In the second setting we switched to the new quasi-dfs reachability scheme.
3. In the third setting we additionally switched to the ZDD data structure.

The figures of the different settings are shown in col. 4 to 12 of Table 4.C, where we once again normed all data to the figures of the activity-local schemes. As an example, the last entry in the last row of the table states that the activity-local scheme is 23.51 times faster than PRISM. By considering Table 2 one can then compute that PRISM consumed $3.067\text{sec} \cdot 23.51 = 72.11$ sec of CPU time for generating the SG of the FMS model for $N = 10$. From Table 4.C (col. 5 (r_{time})) one can conclude that the explicit handling of transitions induces a non-neglible run-time overhead, although it is caused by only a small fraction of the overall number of transitions (cf. columns *trans* and $trans_e$ of Table 1). However, this overhead is justified by two aspects:

(a) The activity-local approach – in contrast to the fully symbolic ones – is not restricted to any specific model description method.
(b) Monolithic models or models not suitable for being partitioned and composed via activity-synchronization (such as FTMP, FMS and CP) can be analyzed very efficiently, where submodel-oriented approaches are problematic.

As shown by the last 7 columns of Table 4.C, the new scheme for reachability analysis (q.-dfs) as well as the use of ZDDs improves the situation significantly. As with all symbolic representation techniques, memory space is not an issue. Even though we store redundant DDs in order to simplify and speed up the activity-local scheme, the CP model, which is the largest model concerning peak memory requirement, consumed only 45 MByte for symbolic SG generation and representation. If memory were at a premium, the redundancy could easily be eliminated without a dramatic increase in run-time.

6 Summary and Future Work

In this paper, we presented the following innovations: (i) *Zero-suppressed multi-terminal binary Decision Diagrams (ZDDs)*, which proved to be an excellent data structure for symbolic SG representation. (ii) *A new algorithm for symbolic reachability analysis*, organized as a quasi-dfs. scheme, which demonstrated significant run-time and (in case of ADDs) also memory savings. Innovations (i) and (ii) can easily be integrated into existing BDD-based tools such as PRISM and CASPA, in order to improve run-time and reduce memory space. (iii) We presented the *activity-local scheme* for generating the state graph by explicit exploration and symbolic composition. The scheme does not only yield compact symbolic representations, but also has the advantage that only a small fraction of the SG needs to be explored explicitly. Consequently, it achieves substantial run-time savings, especially if the high-level model does not have a compositional structure.

Since we develop our implementations in the context of Möbius, we are currently working on an efficient symbolic realization of the "Replicate" feature and on the symbolic treatment of reward variables. Another important step is the availability of efficient numerical algorithms for steady-state and transient analysis. To this end we have adapted the approach of [Par02] to the ZDD data structure. First results are very positive, e.g. applying the pseudo Gauss-Seidel solution method to the FMS model with $N = 8$, reduces run-time for computing steady state probabilities by a factor of 3.11, when compared to the original ADD-based approach of [Par02]. Details as well as further results will be presented in a forthcoming paper.

Acknowledgment: We would like to thank the Möbius, PRISM and CASPA developer groups for their support.

References

[BCL91] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In *Int. Conf. on Very Large Scale Integration*, pages 49–58, Edinburgh, 1991. North-Holland.

[Bry86] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE ToC*, C-35(8):677–691, August 1986.

[CM99] G. Ciardo and A. S. Miner. Efficient reachability set generation and storage using decision diagrams. In *Proc. of ATPN*, LNCS 1639, pages 6–25, 1999.

[CMS03] G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Proc. of TACAS*, LNCS 2619, 2003.

[CT93] G. Ciardo and K. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993.

[CT96] G. Ciardo and M. Tilgner. On the use of Kronecker operators for the solution of generalized stochastic Petri nets. ICASE Report 96-35, 1996.

[DCC$^+$02] 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.

[DKK02] I. Davies, W.J. Knottenbelt, and P.S. Kritzinger. Symbolic Methods for the State Space Exploration of GSPN Models. In *Proc. of TOOLS*, pages 188–199. LNCS 2324, 2002.

[DKS04] S. Derisavi, P. Kemper, and W. H. Sanders. Symbolic State-space Exploration and Numerical Analysis of State-sharing Composed Models. *Linear Algebra and its Applications (LAA)*, 386:137–166, 2004.

[FMY97] M. Fujita, P. McGeer, and J.C.-Y. Yang. Multi-terminal Binary Decision Diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149–169, April/May 1997.

[God95] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems. An Approach to the State-Explosion Problem.* PhD thesis, Université de Liege, 1995.

[HMKS99] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains. In *Proc. of 3'rd NSMC*, pages 188–207. Prensas Universitarias de Zaragoza, 1999.

[KS02] M. Kuntz and M. Siegle. Deriving Symbolic Representations from Stochastic Process Algebras. In *Proc. of PAPM-PROBMIV*, LNCS 2399, pages 1–22, 2002.

[KSW04] 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.

[KVBSV98] 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.

[LS02] K. Lampka and M. Siegle. Symbolic Composition within the Moebius Framework. In *Proc. of 2'nd MMB Workshop*, pages 63–74, September 2002. Forschungsbericht der Universität Hamburg Fachbereich Informatik.

[LS03] K. Lampka and M. Siegle. MTBDD-based activity-local state graph generation. In *Proc. of PMCCS 6*, pages 15–18, 2003.

[Min93] 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.

[Min01] A. Miner. Efficient solution of GSPNs using matrix diagrams. In *Proc. of PNPM*, pages 101–110. IEEE Computer Society Press, 2001.

[Min04] A. Miner. Saturation for a general class of models. In *Proc. of QEST*, pages 282–291. IEEE Computer Society Press, 2004.

[Par02] D. Parker. Implementation of Symbolic Model Checking for Probabilistic Systems, Ph.D. Thesis, University of Birmingham (U.K.), 2002.

[PRC97] E. Pastor, O. Roig, and J. Cortadella. Symbolic Petri Net Analysis using Boolean Manipulation. Technical report, Univ. Politec. de Catalunya, DAC/UPC Report No. 97/8, 1997.

[PRCB94] E. Pastor, O. Roig, J. Cortadella, and R.M. Badia. Petri Net Analysis Using Boolean Manipulation. In R. Valette, editor, *Proc. of ATPN*, LNCS 815, pages 416–435. Springer, June 1994.

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

[Sie98] M. Siegle. Compact representation of large performability models based on extended BDDs. In *Proceedings of PMCCS 4*, pages 77–80, Williamsburg, VA, Sept. 1998.

[Sie02] M. Siegle. Behaviour analysis of communication systems: Compositional modelling, compact representation and analysis of performability properties. Shaker Verlag Aachen, 2002.

[SM92] W. H. Sanders and L. M. Malhis. Dependability evaluation using composed SAN-based reward models. *Journal of Parallel and Distributed Computing*, 15:238–254, 1992.

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

[WL91] 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.