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

Kai Lampka, Markus Siegle

Email:{klampka,siegle}@informatik.unibw-muenchen.de

Institute for Computer Engineering

Bericht 2005-02
September 2005

Universität der Bundeswehr München

Fakultät für

INFORMATIK

Werner-Heisenberg-Weg 39 • D-85577 Neubiberg

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2005	Report

4. TITLE AND SUBTITLE

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

5. FUNDING NUMBERS

6. AUTHOR(S)

Kai Lampka and Markus Siegle

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

UNIBW

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES

Universität der Bundeswehr, Munchen

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

Bericht 2005-02, September 2005

11. SUPPLEMENTARY NOTES

Text in English, 28 pages, 32 references, 3 tables, 5 figures.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

ABSTRACT (Maximum 200 words)

We introduce a new, efficient method for constructing compact symbolic representations of very large stochastic labelled transition systems. Contrary to known symbolic state space generation techniques, our technique is applicable to general high-level models which do not have to possess any particular structure. The method is based on zero-suppressed binary decision diagrams which we extended to the multi-terminal case. The symbolic representation is obtained by evaluating the semantics of the high-level model. During this step of explicit state graph exploration one constructs a separate symbolic representation of all transition induced by the same activity in an on-the-fly fashion. The obtained "activity-local" structures are finally composed in order to obtain a compact symbolic representation of the state graph of the overall system. For the then required step of symbolic reachability analysis we propose a new, sequential and activity-oriented scheme which leads to better run-times than conventional symbolic reachability computation. Comparing our new method to previously published schemes, the paper demonstrates the following advantages: (a) The approach is applicable to a general class of high-level stochastic models. (b) In partial-order style we avoid the explicit generation of shuffled sequences of independent activities, which results in much higher generation speed. (c)The composition scheme, as well as the new data structure, results in extremely compact symbolic representations. Furthermore, the composition scheme does not require any product-form of the models sub-units to be composed, as in case of the Kronecker-based approaches. (d) The proposed variant of symbolic reachability analysis significantly reduces run-time, where other symbolic SG representation methods, e.g. like the ones implemented in the tools CASPA and PRSIM, may benefit from.

14. SUBJECT TERMS

ISL, German, Stochastic modeling, Multi-valued decision diagrams, Matrix diagrams, Binary decision diagrams

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

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

Kai Lampka, Markus Siegle

Email:{klampka,siegle}@informatik.unibw-muenchen.de

Institute for Computer Engineering

Bericht 2005-02
September 2005

Universität der Bundeswehr München

Fakultät für

INFORMATIK

Werner-Heisenberg-Weg 39 • D-85577 Neubiberg

Abstract

We introduce a new, efficient method for constructing compact symbolic representations of very large stochastic labelled transition systems. Contrary to known symbolic state space generation techniques, our technique is applicable to general high-level models which do not have to possess any particular structure. The method is based on zero-suppressed binary decision diagrams which we extended to the multi-terminal case. The symbolic representation is obtained by evaluating the semantics of the high-level model. During this step of explicit state graph exploration one constructs a seperate symbolic representation of all transition induced by the same activity in an on-the-fly fashion. The obtained "activity-local" structures are finally composed in order to obtain a compact symbolic representation of the state graph of the overall system. For the then required step of symbolic reachability analysis we propose a new, sequential and activity-oriented scheme which leads to better run-times than conventional symbolic reachability computation. Comparing our new method to previously published schemes, the paper demonstrates the following advantages: (a) The approach is applicable to a general class of high-level stochastic models. (b) In partial-order style we avoid the explicit generation of shuffled sequences of independent activities, which results in much higher generation speed. (c)The composition scheme, as well as the new data structure, results in extremely compact symbolic representations. Furthermore, the comopsition scheme does not require any product-form of the models sub-units to be composed, as in case of the Kronecker-based approaches. (d) The proposed variant of symbolic reachability analysis significantly reduces run-time, where other symbolic SG representation methods, e.g. like the ones implemented in the tools CASPA and PRSIM, may benefit from.

Kurzfassung

In dem vorliegenden Bericht wird eine neue Methode zur Erstellung symbolischer Darstellungen von großen stochastischen beschrifteten Transitionssystemen vorgestellt. Im Gegensatz zu den bekannten Techniken kann der hier diskutierte Ansatz auf allgemeine stochastische Modellbeschreibungen angewandt werden, ohne daß diese von besonderer, d.h. kompositioneller Struktur sein müssen. Zur symbolischen Zustandsraumdarstellung werden "zero-suppressed" Binaere Entscheidungsdiagramme verwendet, welche wir um mehrere Terminalknoten erweitern. Die symbolische Zustandsraumdarstellung wird gewonnen, indem das zu untersuchende Modell entsprechend der zugrundeliegenden Semantik der Modellbeschreibungsmethode interpretiert bzw. ausgeführt wird. Während dieser expliziten Zustandsraumexploration wird für jede einzelne im Modell spezifizierte Aktivität ein eigenes symbolisch repräsentiertes Transitionssystem "on-the-fly" erzeugt. Die so gewonnenen Aktivitäts-lokalen Strukturen werden dann via Komposition zusammengefaßt, so daß man eine kompakte symbolische Zustandsraumdarstellung des potentiellen Gesamttransitionssystems erhält. Für die dann durchzuführende symbolische Erreichbarkeitsanalyse wird hier ein neues, sequentielles und aktivitäts-orientiertes Verfahren vorgeschlagen, das mit geringeren Laufzeiten als der bisherige Standardalgorithmus aufwartet. Vergleicht man die hier präsentierte Methode mit den bisher publizierten Ansätzen, dann wird folgendes in dem vorliegendem Bericht demonstriert: (a) Der Ansatz läßt sich für eine allgemeine Klasse von stochastischen Modellbeschreibungen verwenden, (b) Der "partial-order-reduction" ähnliche Explorierungsansatz vermeidet die explizite Generierung von verschränkten Sequenzen unabhängiger Aktivitäten, was zu einem erheblichen Geschwindigkeitsvorteil führt. (c) Das Kompositionsschema, gemeinsam mit der neuen Datenstruktur, führt zu extrem kompakten symbolischen Zustandsraumdarstellungen. Darüberhinaus verlangt das Kompositionsschema keine Produkteigenschaft der zu komponierenden Modellbestandteile, wie es bei den bekannten Kronecker-basierten Verfahren der Fall ist. (d) Der neue Algorithmus zur symbolischen Erreichbarkeitsanalyse reduziert die Laufzeit, auch anderer symbolischen Zustandsraumrepräsentationsverfahren, wie sie bspw. in den Werkzeugen CASPA und PRISM realisiert sind.

1 Introduction

Considering the wide proliferation of distributed hardware and software systems, it becomes increasingly important to ensure that such systems work correctly and that they meet high performance and dependability requirements. Stochastic models, e.g. stochastic Petri nets or stochastic process algebra specifications, have shown to be powerful tools for describing and analyzing such concurrent systems. We consider high-level specifications of distributed systems, from which a low-level representation is derived, such as stochastic labelled transition systems (SLTS) or (labelled) Markov chains. This state graph (SG) provides the basis for analysis, be it numerical analysis, model checking or combinations thereof.

Unfortunately, the interleaving semantics can easily lead to a growth of the SG which is exponential in the number of independent activities, a phenomenon commonly known as the "state space explosion" problem. In this paper, we present a new symbolic method for constructing and representing the SG for a general class of models which do not have to possess any particular structure. The symbolic representation is obtained by evaluating the semantics of the high-level model and constructing a separate symbolic set of transitions for each model activity in an on-the-fly fashion, where zero-suppressed multi-terminal binary decision diagrams (ZDDs) are used as the basic data structure. The "activity-local" structures are composed in order to obtain a compact symbolic representation of the SG of the overall system. Our algorithm is a round-based scheme, where exploration, encoding, composition and symbolic reachability analysis are performed until a fixed point is reached. Results obtained from an implementation of our method in the context of the Möbius modelling framework [DCC+02] show that our method is both run-time efficient and memory efficient and therefore enables the analysis of systems whose size would otherwise render them intractable.

1.1 Related work

In the context of stochastic modeling, 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 as published in the literature will be given.

Published symbolic approaches range from the individual generation of each successor state and its symbolic encoding [DKK02] to compositional generation procedures, where operators for symbolic submodel composition are provided [HMKS99, CM99, Sie02, AKN+00]. At the top level, we distinguish between monolithic and compositional approaches, where the latter are based on SG exploration of the overall models subunits as well as on operators for the symbolic composition of these local SGs. or even avoid the representation of the overall transition system by employing Kronecker-matrix-operations [CM99]. In contrast monolithic approaches do not take advantage of any structure as possibly inherited by the high-level model, the SG is generated in one step by exploring all enabled activities in each state, which may lead to tremendous run-time overhead or peak memory sizes. We further distinguish between fully symbolic approaches and hybrid approaches, where hybrid characterizes a combination of explicit exploration and sym-

bolic encoding. Fully symbolic methods require a symbolic realization of the next-state function, which is directly derived from the high-level model description. Thus the latter methods are highly efficient, since they avoid any explicit SG exploration, but they are limited to the case of the respective model description method, e.g. like R-TIPP [KS02] (a stochastic Process Algebra as employed in the tool CASPA [KSW04], simple k-bounded Petri nets, or the input language of [Par02, Pri].

1. **Monolithic approaches**
   These methods do in principle not consider any particular structure of the high level model, but either suffer from long run-times or depend on the model description method.

   (a) *Hybrid*: In [DKK02] the reachability set of a stochastic Petri net is generated by successively firing the enabled transitions, one at a time. Each detected state vector is encoded as a binary decision diagram (BDD) and inserted via disjunction into the decision diagram (DD) representing the set of states reached so far. Due to its sequential nature this approach suffers from long run-times. Besides this the memory savings achieved are due to the use of *P-invariants*, whose computation require that the S-PN is of a certain kind.

   (b) *Fully symbolic*: The method presented in [PRCB94] gives a symbolic transition function for each activity[1] as defined in a *non-stochastic, 1-bounded* PN. It generates the set of all reachable markings by introducing the standard breadth-first search (bfs.) algorithm for symbolic reachability analysis. Even though this approach is highly efficient, its applicability is limited to the case of PNs, where this approach was latter extended to the case of *k-bounded* weighted PNs with *inhibitor arcs* [PRC97].

2. **Compositional approaches**
   Compositionality is known to be crucial for the success of symbolic methods, since it reduces run-time and space complexitiy. Runtime is reduced, since only sequences of activities at the level of sub-units, sub-models resp. are extracted explicitly, so that the explicit generation of all shuffeled execution sequences of independent activities is avoided. The reduction of space complexity is gained from regularity of the symbolic structures as induced by the composition schemes [EFT93, HMKS99, HKN+03]. Consequently compositional approaches, i.e. *all* of the approaches listed below, require therfore an adequat compositional structure of the high-level model, where furthermore the SGs of the submodels in isolation need to be finite. However the partitioning of flat models into independent subunits with local SGs of adequate sizes is still an open question.

   (a) *Hybrid*: If the high-level model is partitioned into submodels, it may be possible to generate the SG of each individual submodel in a conventional, explicit manner. The submodel SGs are then encoded as DDs and afterwards composed by a symbolic composition scheme, where the composition may take either of the two following forms:

[1] In contrast to standard PN notations, the term activity is emphasized here, so that transitions will in the following always address the low-level counterparts of activities when SG generation has taken place.

i. Synchronization over a set of activities, either by employing a Kronecker structure to compute the elements of the overall generator matrix [CM99], or by applying a symbolic version of the synchronization operator to generate a symbolic representation of the overall transition matrix [Sie98, Sie02].

ii. Composition via state variable sharing, and application of a symbolic "Join"-operator [LS02].

(b) *Fully symbolic*: In this case, the modular high-level specification is translated directly to a DD-based representation, where submodel encodings are composed by symbolic synchronization operators [AKN+00, Par02, KS02].

Many of the approaches listed above are limited to cases where an upper bound for the value of each state variable (SV) is known *a priori*. This restricts their applicability to cases where the bounds are specified in the model [KS02, Par02], where the local SG can be generated in isolation [CM99, Sie98, Sie02, LS02], or where bounds can be computed, e.g. by means of invariant analysis [PRCB94, DKK02]. In order to overcome this restriction, recently developed methods generate the local SGs in an interleaved fashion [CMS03, DKS03], but the application of these methods is problematic in case of flat models where a partitioning into adequate submodels is not obvious. As a further problem, concurrency taking place within one of the submodels is not detected, i.e. shuffled sequences of independent activities are fully expanded at the submodel level. These considerations result in two focal aims for our new scheme:

1. The individual treatment of states (both their exploration and encoding) should be avoided as much as possible.

2. The scheme should be applicable to both, structured and flat models.

Our activity-local scheme, whose basic idea we had described briefly in [LS03] (but for a limited class of models and using standard symbolic reachability analysis), achieves these goals by maintaining compositionality at the lowest level, i.e. at the level of individual activities. Due to the nature of Bryant's [Bry86] `Apply`-algorithm, the activity-local structures do not need to fulfill any *product-form* requirement as is the case for Kronecker-based schemes. Thus the activity-local approach does not require any particular structure of the high-level model.

In order to extend the saturation technique of [CMS03] to a general class of models, [Min04] describes a kind of `Apply`-algorithm for building the cross-product of two matrix diagrams. This algorithm allows [Min04] to employ the same composition scheme in the context of matrix diagrams, as 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,where the latter follows an activity-wise startegy. Thus similiar to the approach of [BCL91], one executes partitions of the overall transition system sequentially, rather than executing them all at once. Furthermore it enables one to employ *greedy chaining* on the set of states to be explored in the next step. As we recently noticed a similiar startegy, however in case of k-bounded non-stochastic Petri nets is also proposed in [PRC97].

1.2 Organization of the paper

Sec. 2 introduces the model world and discusses the encoding of labelled Markov chains by ZDDs. Sec. 3 explains our new algorithms and discusses their features. Empirical results are presented in Sec. 4, and Sec. 5 concludes the paper.

2 Background

2.1 Static properties of high-level model descriptions

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. As a consequence, each state of the model is given as a vector $\vec{s} \in \mathcal{S} \subseteq \mathbb{N}^{|S|}$. Concerning the high-level model description by means of Petri nets or process algebras, 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, denoted $\mathcal{A}ct$. SVs and activities are connected through a connection relation $\mathcal{C}on \subseteq (S \times \mathcal{A}ct) \cup (\mathcal{A}ct \times S)$. Thus the execution of an activity $l \in \mathcal{A}ct$ depends on a certain set of SVs (the enabling set), and when it is executed it changes the values of a certain other set of SVs (the set of affected SVs). In the style of Petri nets we denote the set of enabling SVs as *pre*-set $\triangleright l := \{s_i \in S | (s_i, l) \in \mathcal{C}on\}$, and the set of affected SVs as *post*-set $l\triangleleft := \{s_i \in S | (l, s_i) \in \mathcal{C}on\}$. The union of these sets will be denoted as the set of dependent SVs of activity l, $\triangleright l\triangleleft := \triangleright l \cup l\triangleleft$. For each activity $l \in \mathcal{A}ct$, we define a projection function χ^{D_l}: $\mathbb{N}^{|S|} \longrightarrow \mathbb{N}^{|\triangleright l\triangleleft|}$ which yields the sub-vector consisting of the dependent SVs only. We use the shorthand notation $\vec{s}_{D_l} := \chi^{D_l}(\vec{s})$, 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. $(k, l) \in \mathcal{A}ct^{\mathcal{D}} \Leftrightarrow \triangleright k\triangleleft \cap \triangleright l\triangleleft \neq \emptyset$. Now the set of dependent activities for each activity l can be defined as $\mathcal{A}^{D_l} := \{k \in \mathcal{A}ct \mid (l, k) \in \mathcal{A}ct^{\mathcal{D}}\}$. Note that according to this definition we have $l \in \mathcal{A}^{D_l}$. Each time activity l is executed, the activity-local markings for the activities $\in \mathcal{A}^{D_l}$ may have changed as well, so that new transitions might be obtainable by executing these activities. We will make use of this set in our scheme.

2.2 Dynamic properties of high-level models

When an activity takes place, the model evolves from one state to another. The transition function $\delta : \mathcal{S} \times \mathcal{A}ct \longrightarrow \mathcal{S}$ depends on the model description method. Concerning the target state of a transition, we use the superscript of a state descriptor or SV to indicate the sequence of activities leading to that state, thus we write $\vec{s}^{\,l} := \delta(\vec{s}, l)$. If activity l is enabled in state $\vec{s}$ we write $\vec{s}[\triangleright l$. We also define the partial rate function $\eta : \mathcal{S} \times \mathcal{A}ct \times \mathcal{S} \longrightarrow \mathbb{R}^{>0}$, which yields the rate at which the model moves from source to target state when a specific activity l occurs. The rate $\eta(\vec{s}, l, \vec{s}')$ is undefined if $\delta(\vec{s}, l) \neq \vec{s}'$. During SG exploration, δ and η define the successor-state relation as a set of quadruples $T \subseteq (\mathcal{S} \times \mathcal{A}ct \times \mathbb{R}^{>0} \times \mathcal{S})$, which is the set of transitions of a stochastic labelled transition

system.

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:

$$T^l := \{(\vec{s}_{D_l}, l, \lambda, \vec{s}^{\,l}_{D_l}) \mid \vec{s}_{D_l} = \chi^{D_l}(\vec{s}) \wedge \vec{s}^{\,l}_{D_l} = \chi^{D_l}(\vec{s}^{\,l}) \wedge (\vec{s}, l, \lambda, \vec{s}^{\,l}) \in T\} \tag{1}$$

Note that, due to the abstraction from the independent SVs, an element of T^l might correspond to more than one element of T. Concerning two activities l and k, we define the following partitioning of the set of SVs:

$$\begin{array}{ll} D^{1,l}_{k,l} := \triangleright l \triangleleft \cap \overline{\triangleright k \triangleleft} & D^{1,k}_{k,l} := \overline{\triangleright l \triangleleft} \cap \triangleright k \triangleleft \\ D^{2}_{k,l} := \triangleright l \triangleleft \cap \triangleright k \triangleleft & D^{3}_{k,l} := \overline{\triangleright l \triangleleft} \cap \overline{\triangleright k \triangleleft} \end{array} \tag{2}$$

The pairwise intersection of the above sets is empty and their union is the set of all SVs S. After a suitable reordering of the state descriptor we can write $\vec{s} = (\vec{s}_{1,l}, \vec{s}_{1,k}, \vec{s}_2, \vec{s}_3)$. We can then distinguish the following cases concerning the execution sequences $\rho = l\,k$ and $\omega = k\,l$:

$$\begin{array}{l} \vec{s} \xrightarrow{\rho} \vec{s}^{\,lk}\colon\ \vec{s} \xrightarrow{l} (\vec{s}^{\,l}_{1,l}, \vec{s}_{1,k}, \vec{s}^{\,l}_2, \vec{s}_3) \xrightarrow{k} (\vec{s}^{\,l}_{1,l}, \vec{s}^{\,k}_{1,k}, \vec{s}^{\,lk}_2, \vec{s}_3) \\ \vec{s} \xrightarrow{\omega} \vec{s}^{\,kl}\colon\ \vec{s} \xrightarrow{k} (\vec{s}_{1,l}, \vec{s}^{\,k}_{1,k}, \vec{s}^{\,k}_2, \vec{s}_3) \xrightarrow{l} (\vec{s}^{\,l}_{1,l}, \vec{s}^{\,k}_{1,k}, \vec{s}^{\,kl}_2, \vec{s}_3) \end{array}$$

In case $(l,k) \notin \mathcal{A}ct^{\mathcal{D}}$ we have $D^2_{k,l} = \emptyset$ and thus

$$\begin{array}{ll} \text{if } \vec{s}\lceil \triangleright k \text{ then } \vec{s}^{\,l}\lceil \triangleright k & \text{(Prop. Ia)} \\ \text{if } \vec{s}\lceil \triangleright l \text{ then } \vec{s}^{\,k}\lceil \triangleright l & \text{(Prop. Ib)} \\ \vec{s}^{\,lk} = \vec{s}^{\,kl} & \text{(Prop. II)} \end{array} \tag{3}$$

Thus, the order of the independent activities k and l is without significance (diamond property, Prop. II). It is obvious that one may execute these activities independently on a given source state $\vec{s} = (\vec{s}_{1,l}, \vec{s}_{1,k}, \vec{s}_2, \vec{s}_3)$, where the target state of the sequential execution of either kl or lk can be obtained by combining the dependent sub-vectors $\vec{s}^{\,l}_{1,l}$ and $\vec{s}^{\,k}_{1,k}$. It is clear that the above properties also hold for sequences of more than two activities which are pairwise independent. This yields a well-known equivalence relation on the set of sequences of transitions, where two sequences ω and ρ are considered equivalent if and only if they can be obtained from each other by swapping adjacent independent transitions. Each equivalence class is commonly denoted as a trace [God95].

2.3 Symbolic encodings of state graphs

Binary decision diagrams (BDDs) are a popular data structure for symbolic SG representation. In the context of stochastic modelling, the most prominent decision-diagram based data structures are *multi-terminal* or *algebraic BDDs* (ADDs) [FMY97], *multi-valued decision diagrams* (MDDs) [KVBSV98] and *matrix diagrams* [Min01].

2.3.1 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 $\mathcal{E}_{Act}$ using n_{Act} bits. This gives us the following binary encoding scheme:

$$(\vec{s} \xrightarrow{l} \vec{s}^{\,l}) \mapsto \left(\mathcal{E}_{Act}(l), \mathcal{E}_1(s_1), .., \mathcal{E}_n(s_n), \mathcal{E}_1(s_1^l), .., \mathcal{E}_n(s_n^l)\right)$$

The rate λ is not encoded in binary form, it will be stored in a terminal node of the ADD.

2.3.2 Zero-suppressed multi-terminal binary DDs (ZDDs)

In a reduced ordered BDD, isomorphic subgraphs have been merged and *don't care* nodes[2] are skipped. Zero-suppressed BDDs (Z-BDDs) [Min93] are derivatives of BDDs for representing sparse sets efficiently. In Z-BDDs, instead of eliminating don't-care nodes, one eliminates those non-terminal nodes whose 1-successor is the terminal 0-node. We extend Z-BDDs to the multi-terminal case, i.e. a ZDD is like a multi-terminal BDD, but instead of eliminating don't care nodes we eliminate those nodes whose 1-successor is the terminal 0-node. 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].

2.3.3 ZDD-based representation of SGs

A transition of a labelled transition system can be encoded by a Boolean vector. Each bit position of the vector corresponds to a Boolean variable of the ZDD representing the overall 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 the 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_{Act} 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, which is a commonly accepted heuristics for obtaining small BDD sizes. For convenience we will use the somewhat sloppy notation $\vec{s} \in \mathsf{Z}$ to denote the check whether the encoding of a certain state $\vec{s}$ is contained in the ZDD Z either as a source or as a target state.

2.3.4 Unknown bounds for SVs

The values K_i are in general not known a priori 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 declaring a new Boolean variable for Z, i.e. *without* (!) changing the structure of the DD. Thus it is not necessary to know the maximum value K_i of SV variables s_i in advance, and the introduction of new bits does not slow down the generation process.

[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 [Som98], but we extend each DD by the set of variables on which it depends. This allows us to implement an `Apply`-algorithm for partially shared ZDDs.

2.3.5 Example

In order to demonstrate the process of symbolic encoding and the advantages of ZDDs, we will complete this section by discussing a small example. Part (A) and (B) of Fig. 1 show a simple SPN and its underlying SLTS[4]. The Boolean encodings of the transitions of the SLTS are specified in table (C), where activity labels are encoded by a-bits, source states by s-bits and target states by t-bits.[5] Part (D) shows the corresponding ADD M, where the Boolean variables encoding the bits of source and target states 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[6].

3 Symbolic Activity-local State Graph Generation

3.1 Main ideas

The main idea of our approach is the explicit exploration of parts of the SG, where a detected transition is encoded symbolically and inserted into an "activity-local" ZDD. The modular or hierarchical structure of the model is without any significance for this scheme, we only need to know the set of dependent SVs for each activity. Each activity l has its own ZDD which depends only on those Boolean variables which encode the dependent SVs of l. After the generation of the activity-local ZDDs, the symbolic representation of the overall SG is obtained by composing the activity-local ZDDs and carrying out a symbolic reachability analysis. Several rounds of generation and composition may be needed to construct the overall SG.

Let us assume, that at the end of an exploration phase we have $|\mathcal{A}ct|$ ZDDs Z_l each of which encodes the corresponding relation T^l as defined in eq. 1. We define the sets of dependent Boolean source and target variables, as well as the sets of their independent counterparts:

$$\mathsf{D}_l := \{\vec{\mathsf{s}}^i, \vec{\mathsf{t}}^i | s_i \in \triangleright l \triangleleft\} \quad \mathsf{I}_l := \{\vec{\mathsf{s}}^i, \vec{\mathsf{t}}^i | s_i \in \overline{\triangleright l \triangleleft}\} \tag{4}$$

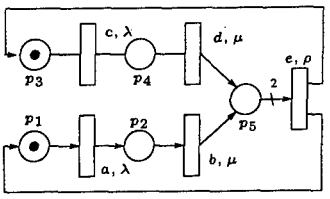
In this equation, $\vec{\mathsf{s}}^i$ and $\vec{\mathsf{t}}^i$ denote those Boolean variables which encode the value of the SV s_i in the source and target state of a transition. The activity-local ZDD for activity l depends only on the set D_l. Before composition can take place, Z_l needs to be supplemented by the set of independent Boolean variables I_l, yielding the symbolic representation of the set of *potential* transitions induced by activity l. When activity

[4]For the moment, the bold, regular and dashed arrows of the SLTS have the same meaning, we will discuss the difference between them in Sec. 3.3.

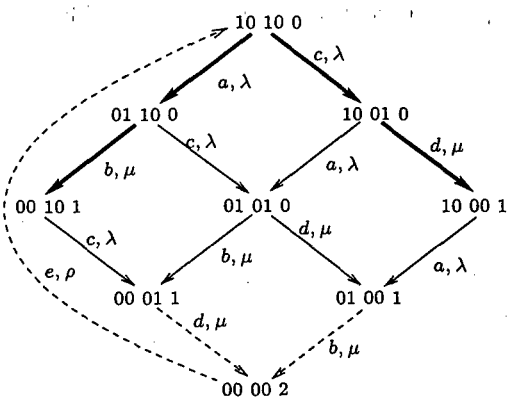
[5]The 5 integer state variables are encoded by 6 bits, since only the last one (the marking of place p_5) can take a value other than 0 or 1.

[6]In this example, the ZDD reduction rule can be applied in a straight-forward manner, since incidently in M no node is skipped.

(A) A stochastic Petri net



(B) The corresponding 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

(D) ADD representing the SLTS

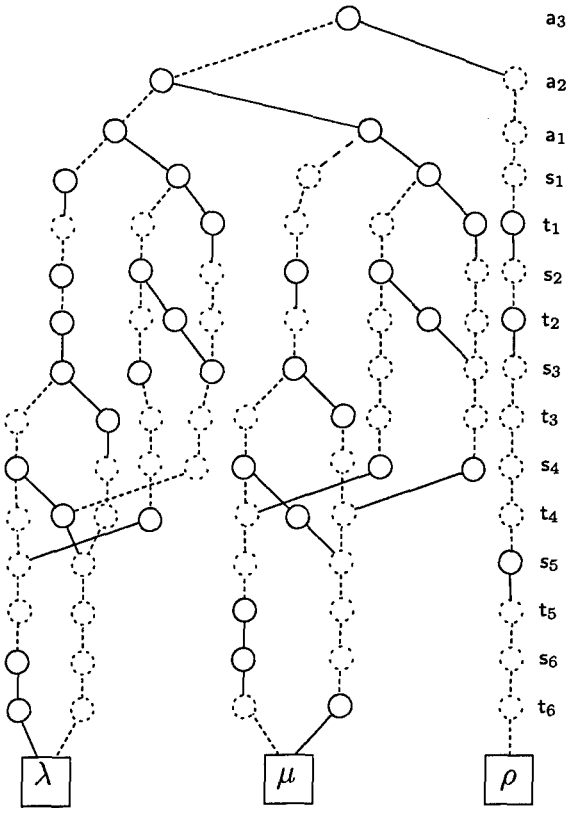


Figure 1: From a SPN to the symbolic representation of its underlying SLTS

l takes place, the SVs $s_i \in \overline{\triangleright l \triangleleft}$ do not change their values, they stay stable, which is expressed by the pairwise identity over the Boolean variables contained in I_l:

$$\mathsf{Stab}_l(\vec{\mathsf{s}}_{I_l}, \vec{\mathsf{t}}_{I_l}) := \bigwedge_{s_i \in \overline{\triangleright l \triangleleft}} \bigwedge_{j=1}^{n_i} (\mathsf{s}_j^i \leftrightarrow \mathsf{t}_j^i,)$$

During composition, the activity-local ZDDs are combined in order to obtain the transition relation of the overall model:

$$\mathsf{Z}_T := \sum_{l \in \mathcal{A}ct} \mathsf{Z}_l \cdot \mathsf{Stab}_l \cdot \mathsf{A}_l \tag{5}$$

Hereby A_l represents the binarily encoded activity label l. The ZDD Z_T thus constructed encodes a set of potential transitions of the overall model. Therefore, at this point it is necessary to perform symbolic reachability analysis.

For generating the sets of activity-local transitions T^l we follow a *selective* breadth-first-search strategy, i.e. for a detected state $\vec{s}^l$ which was reached by firing action l we generate the set of successor states by applying the transition function δ for each dependent activity $k \in \mathcal{A}_{\vec{s}^l}^{D_l}$, where

$$\mathcal{A}_{\vec{s}^l}^{D_l} := \{k \in \mathcal{A}^{D_l} \mid \vec{s}_{D_k}^l \notin \mathsf{E}_k \ \wedge \ \vec{s}^l \, [\triangleright \, k\} \tag{6}$$

In eq. 6, $\vec{s}_{D_k}^l \notin \mathsf{E}_k$ states that activity k was not yet tested on the activity-dependent marking of state $\vec{s}^l$. The Z-BDD E_k is introduced here for convenience. It encodes those activity-local markings on which activity k was already tested (successfully or not)[7]. Consequently, E_k is initialized with the model's initial state $\vec{s}^\epsilon$. For initializing the activity-local SG generation procedure we define $\mathcal{A}_{\vec{s}^\epsilon}^{D_\epsilon}$, which is the set of activities enabled in the initial state.

3.2 SG Generation Scheme

The SG generation is realized with the help of two complementary procedures, EncodeTransitions and ExploreStates (shown in Fig. 2.A and 2.B), which we discuss in the sequel. In line 2 of algorithm EncodeTransitions a transition is read from the $TransBuffer$, and in lines 3 - 7 the set $\mathcal{A}_{\vec{s}^l}^{D_l}$ of activities enabled in the successor state is determined. The list of state-activity tuples to be explored further is inserted into the $StateBuffer$ in line 9, and finally the activity-local encoding of the current transition is inserted into ZDD Z_l. The complementary exploration routine ExploreStates for executing the set of activities $\mathcal{A}_{\vec{s}}^{D_l}$ on a state $\vec{s}^l$ works as shown in Fig. 2.B. In line 2, a state together with a list of activities to be checked is read from the $TransBuffer$. For each activity from that list, the successor state $\vec{s}^{lk}$ and the corresponding rate λ are computed (lines 4 and 5). The transition thus found is inserted into the $TransBuffer$ (line 7), provided it is not a self-loop (line 6)[8]. By executing procedures ExploreStates and EncodeTransitions in an alternating fashion, the algorithm will reach a point where EncodeTransitions has been executed and the $StateBuffer$ is still empty. This means that the algorithm has visited all states

[7] One could also test if $\vec{s}_{D_k}^l \in \mathsf{Z}_k$, either as source or target state. Repeated tests of states would only induce a small run-time overhead.

[8] Self-loops can safely be omitted since they do not influence transient or steady-state probabilities.

(A) Encoding and insertion of transitions into Z_l

```
(0)   EncodeTransitions()
(1)     while (TransBuffer ≠ empty) do begin
(2)       (s⃗, l, λ, s⃗^l) ⟵ TransBuffer
(3)       A^{D_l}_{s⃗^l} := ∅
(4)       for each k ∈ A^{D_l} do begin
(5)         if s⃗^l_{D_k} ∉ E_k ∧ s⃗^l [▷ k then A^{D_l}_{s⃗^l} := A^{D_l}_{s⃗^l} ∪ {k}
(6)         E_k := E_k ∪ s⃗^l_{D_k}
(7)       end
(8)       if A^{D_l}_{s⃗^l} ≠ ∅ then
(9)         StateBuffer ⟵ (s⃗^l, A^{D_l}_{s⃗^l})
(10)      Z_l := Z_l + E(s⃗_{D_l}, λ, s⃗^l_{D_l})
(11)    end
```

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

```
(0)   ExploreStates()
(1)     while (StateBuffer ≠ empty) do begin
(2)       (s⃗^l, A^{D_l}_{s⃗^l}) ⟵ StateBuffer
(3)       for each k ∈ A^{D_l}_{s⃗^l} do begin
(4)         s⃗^{lk} := δ(s⃗^l, k)
(5)         λ := η(s⃗^l, k, s⃗^{lk})
(6)         if s⃗^l ≠ s⃗^{lk} then
(7)           TransBuffer ⟵ (s⃗^l, k, λ, s⃗^{lk})
(8)     end
```

Figure 2: Algorithms for explicit exploration and encoding

Symbolic composition, symbolic reachability analysis
and refill of $StateBuffer$

```
(0)  InitiateNewRound()
(1)    Z_R := ReachabilityAnalysis()
(2)    for each k ∈ Act do begin
(3)      Temp := Z_R \ E_k
(4)      while Temp ≠ ∅ do begin
(5)        s⃗ ⟵ Temp
(6)        if s⃗[▷k then StateBuffer ⟵ (s⃗, {k})
(7)        Temp := Temp \ {E(s⃗)}
(8)      end
(9)    end
```

Figure 3: Algorithm for re-initiating a new round of explicit exploration and encoding

reachable from the initial state(s) through sequences of dependent activities. However so far we have not considered the combined execution of independent activities which may trigger new model behavior. This is important, since the activity-local scheme does not generate states of the latter type explicitly, they are obtained by symbolic composition, i.e. applying eq. 5. The whole functionality of testing such states is encapsulated in algorithm InitiateNewRound, where symbolic composition and reachability analysis is realized by procedure ReachabilityAnalysis, its realization will be discussed below (see Sec. 3.4).

In lines 2 - 9 of algorithm InitiateNewRound (Fig. 2) one 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 6) in order to re-initialize the $StateBuffer$ for a new round of explicit SG exploration and symbolic encoding. A fixed point in SG generation is reached when the $StateBuffer$ is still empty after the execution of InitiateNewRound. After the final call of InitiateNewRound, $Z := Z_T \cdot Z_R$ gives one the symbolic representation of the reachable SG of the overall model as ZDD Z.

The top-level algorithm for our activity-local SG generation and encoding strategy is shown in Fig. 4.D. In lines 1 - 3, the $StateBuffer$, the $TransBuffer$ and the Z-BDDs E_k are initialized. In the inner loop (lines 5 - 8) procedures ExploreStates and EncodeTransitions are called in an alternating fashion. The re-initialization, performed by procedure InitiateNewRound is called in line 9, before a new round of the outer loop (lines 4 - 10) is started. The final set of all reachable transitions is computed in line 11.

3.3 Comments on the activity-local generation scheme

In this section we will reason about the correctness and completeness of our activity-local approach, i.e. we will discuss the correctness of the symbolic activity-local composition scheme and we will discuss the "partial" character of its explicit exploration part.

```
(0)  ExploreStateGraph()
(1)    StateBuffer ⟵ (s⃗^ε, {A^{D_ε}_{s⃗^ε}})
(2)    TransBuffer = ∅
(3)    for each k ∈ Act do begin E_k := E(s⃗^ε_{D_k}) end
(4)    do begin
(5)      do begin
(6)        ExploreStates()
(7)        EncodeTransitions()
(8)      end until StateBuffer = ∅
(9)      InitiateNewRound()
(10)   end until StateBuffer = ∅
(11)   Z_T := ( Σ_{l∈Act} Z_l · Stab_l · A_l) · Z_R'
```

Figure 4: Main algorithm of activity-local SG generation

3.3.1 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 (i.e. the execution of l may influence the enabledness of k)
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 D_k is identical to that of $\vec{s}^l$ (i.e. $\vec{s}^l_{D_k} = \vec{t}_{D_k}$).

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 D_l. The SVs outside the set D_l may take arbitrary values, but they must remain stable upon execution of activity l, which is expressed by the multiplication with Stab_l. This has the effect that a single detected transition is encoded as a possibly huge set of potential transitions. By performing a symbolic reachability analysis, this set of potential transitions is reduced to the transitions which are actually reachable from the initial state, yielding only legal transitions.

3.3.2 Completeness of the generation scheme

According to the diamond property [God95] (c.f. Sec. 2.2) for two independent activities l and k (here $(l,k) \notin \mathcal{A}ct^{\mathcal{D}}$), the order of their execution is without significance. Consequently 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 than 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 [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 as mentioned above. This is exactly the functionality of the algorithms presented in Fig. 2.

3.3.3 Example

We consider again the example depicted in Fig. 1, where we will especially illustrate now the reasons why in the general case the activity-local generation scheme may need more than one round of exploration. One 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 called by algorithm InitiateNewRound. We will now explain why the transitions drawn as dashed arrows in the figure are not generated during the first round of exploration, i.e. the reason why more than one round of explicit exploration is required. 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, however, the algorithm will detect the fact 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 the $StateBuffer$ at this point, and this dashed transition (as well as the other two dashed transitions) will be explored in the second round.

3.4 Symbolic reachability analysis

We now discuss two variants of a reachability algorithm as required by algorithm InitiateNewRound (line 1 of algorithm of Fig. 4.C). – In line 1 of the algorithm of Fig. 5.A we first compute the ZBDD Z_T, which represents the set of *potential* transitions, (for simplicity, activity-labels are omitted and rates are dropped). Furthermore the algorithm employs another three ZBDDs: The ZBDD Z_U for representing the set of *unexplored states*, the 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 binary encoding of the initial state $\vec{s}^{\epsilon}$, where the function $\mathcal{M}$ constructs the respective ZBDD (lines 2 and 3). The standard breadth-first-search (bfs) symbolic reachability analysis is realized by the `do-until` 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). From the level of

(A) Quasi-parallel symbolic reachability analysis
bfs. traversal as proposed by [PRCB94, Sie02]

```
(0)  ReachabilityAnalysis()
(1)    Z_T := ∑_{l∈Act} Z_l · Stab_l
(2)    Z_R := M(t⃗, E(s⃗^ε))
(3)    Z_U := M(s⃗, E(s⃗^ε))
(4)    do begin
(5)       Z_tmp := Abstract(Z_T ∧ Z_U, s⃗, ∨) \ Z_R;
(6)       Z_R = Z_R ∨ Z_tmp;
(7)       Z_U := Z_tmp{s⃗ ← t⃗};
(8)    end until Z_U = ∅
(9)    Z_R := Z_R{s⃗ ← t⃗};
```

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

```
(0)  ReachabilityAnalysis()
(1)    Z_R := ∅;
(2)    Z_U := M(s⃗, E(s⃗^ε));
(3)    for each k ∈ Act do begin Z̃_k := Z_k · Stab_k end
(4)    do begin
(5)       Z_R := Z_R ∨ Z_U
(6)       for each k ∈ Act do begin
(7)          Z_tmp := Abstract(Z̃_k ∧ Z_U, s⃗, ∨) \ Z_R;
(8)          Z_U := Z_U ∨ Z_tmp{s⃗ ← t⃗};
(10)      end
(11)      Z_U := Z_U \ Z_R
(12)   end until Z_U = ∅
```

Figure 5: Pseudo-code of symbolic reachability analysis variants

the reachability algorithm this step is set-oriented and parallel, since Z_U may represent more than one state, and one obtains all successor states. We propose now the following improvements:

1. replace the "*parallel*" scheme of line 5 Fig. 5.A by an *activity-wise* scheme (lines 6 - 10 Fig. 5.B).

2. update the set of unexplored states as soon as possible (line 8 Fig. 5.B).

If Z_U of Fig. 5.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. 5.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 iter-

ations (*#iter*) of the main (outer) `do-until loop`. In Sec. 4 where the empirical results of the two reachability algorithms are presented, we will refer to this reduction by the ratio r_{iter}. The order of execution the symbolic encoded state-to-state transition function thus influences the generation speed. Consequently one may refine or coarsen the set of states explored in each for-loop, e.g. bookkeeping the sets of unexplored states for each activity individually. Depending on the employed high-level model, doing so influences the run-time behavior significantly. E.g. in case of the FTMP model the most efficient strategy is the strategy to explore all states reached so far (do not remove already explored states form Z_U), however in case of the FMS model such a strategy may almost double the run-time.

As we discovert recently the authors of [PRC97] also develop the idea of symbolic Petri net traversal by applying transition chaining. By applying the symbolic encoded activity transition functions individually one is there also enabled to directly insert the states reached next into the set of unexplored states (line 7 and 8, algorithm 5.B), but there without removing the already reached ones. However our experiments showed us, that this so called *greedy chaining* technique, even though it reduces the number of iterations of the outer `do-until loop` and thus calls to the `Apply`– and `Abstract`-algorithms, plays often a miner role only. Consequently it seems that the sequential employment of a somehow partiotioned set of symbolic transition functions, which was to the best of our knowledge already suggested by [BCL91], is the main source of runtime reduction. I.e. the sequential handling of small DD-structures is more efficient as handling big DD structures once-at a time, where the update strategy of the set of unexplored states Z_U plays a minor role only, which of course depend on the employed model.

4 Empirical Evaluation

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. Consequently, such a framework is highly suited for implementing the activity-local approach. Furthermore, this offers the opportunity to compare our method to the compositional MDD- and Kronecker-based approach of [DKS03], where submodel SGs are generated in an interleaved fashion and symbolically encoded on-the-fly.

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 our symbolic engine and Möbius (algorithm of Fig. 2.B).
2. The symbolic engine (mainly algorithm (A) and (C) of Fig. 2 and one algorithm of Fig. 5).
3. A ZDD-library (based on the CUDD-package [Som98]), which contains the algorithms for manipulating partially shared ZDDs and implements a C++ wrapper for them.

The experiments carried out with our implementation, as well as the ones executed with CASPA [KSW04] and PRISM [Pri], were run on a Pentium IV 3 GHz system with 1 GByte of RAM and a Linux OS. All run-time results were averaged from 100 runs.

4.1 Comparison of the ADD and ZDD data structures

Table 1 illustrates the difference between the ADD- and ZDD-based encoding schemes with respect to their space and time complexity for five different models taken from the literature. The first column gives the model scaling parameter, the second gives the total number of Boolean variables required for encoding all SVs. In order to make a fair comparison, we encoded each SV by a minimum number of bits. In practice such an allocation strategy for ADD-variables is not feasible, due to the lack of a priori knowledge of the maximum value K_i taken by SV s_i, but a brute-force strategy, where one allocates as many ADD variables as possible, significantly increases memory space and run-time. In case of ZDDs, pre-allocation of Boolean variables is unnecessary, since skipped variables are interpreted as being 0-assigned. In order to give the reader an impression of the dimensions of the employed DD-structures, Table 1 gives the number of nodes required for representing the set of reachable states (encoded by Z-BDD 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. We also collected the number of cache hits and misses (concerning the DD "computed table"), in order to give an impression of the number of calls to the recursive `Apply` and `Abstract` algorithms. Column t_g contains the generation time in seconds. In order to simplify the comparison, on the right-hand side of the table we provide the ratios of memory consumption for Z_R, Z_T and the peak number of nodes, where figures are normed with respect to the ZDD-based version. The last two columns in Table 1 give the ratio of the cache hit rates (r_{chr}) and the ratio of the construction times r_{time}, were in both case the ZDD-variant was once again considered of being of unit 1.

As illustrated by the various case studies, the use of ZDDs reduces memory consumption. As a consequence of smaller DD sizes, run-time and cache hit rate both improve. The Tandem Queuing Network model, which we specified as a SPN consisting of 3 places, constitutes a very interesting case study. Two of the places 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 requirements of the ADD-based scheme are to be favored. If N is a power of two, the enumeration scheme is much sparser and a different picture has to be drawn. Surprisingly, the ZDD-based scheme maintains its run-time advantage in both cases, which seems to be a consequence of the fact that using ZDDs one does not need to allocate nodes for 0-assigned variables. Note that the Tandem Queuing Network model is a worst-case scenario for the activity-local scheme concerning the number of transitions to be explicitly explored and binarily encoded.

From a certain size on, the FMS model has smaller run-time using ADDs than using ZDDs, even though the final ADD-structures are much larger than their ZDD counterparts. We

N	n	ZDD-based scheme						ADD-based scheme				
		# nodes			caching		t_g in secs.	ratios[a]				
								mem. space			r_{chr}	r_{time}
		Z_R	Z_T	peak	*hits*	*misses*		r_R	r_T	r_{peak}		
Fault-tolerant Multiprocessor (FTMP) [SM92]												
2	132	256	5792	$7.483e^4$	$4.633e^5$	$1.844e^5$	0.277	2.01	2.38	2.36	0.39	1.81
3	196	610	16225	$2.546e^5$	$1.725e^6$	$7.128e^5$	1.179	1.99	2.40	2.31	0.43	1.59
4	262	1044	30892	$6.201e^5$	$4.174e^6$	$1.777e^6$	3.268	2.00	2.41	2.31	0.22	1.80
5	326	1556	49845	$9.742e^5$	$8.646e^6$	$3.723e^6$	7.257	1.99	2.41	2.90	0.43	1.53
6	390	2146	73002	$2.278e^6$	$1.599e^6$	$6.955e^5$	14.082	1.99	2.41	2.20	0.39	1.44
Courier Protocol (CP) [WL91]												
3	122	198	2452	$2.328e^5$	$3.175e^6$	$1.518e^6$	1.997	1.90	2.41	2.49	0.44	1.81
4	144	271	3490	$4.168e^5$	$5.540e^6$	$2.878e^6$	3.781	2.11	2.65	2.80	0.41	2.03
5	144	353	4715	$7.303e^5$	$8.421e^6$	$4.426e^6$	5.871	1.93	2.43	2.67	0.39	1.97
6	144	433	5941	$1.212e^6$	$1.221e^7$	$6.489e^6$	8.778	1.82	2.29	2.59	0.49	2.06
7	144	515	7184	$1.943e^6$	$1.728e^7$	$9.310e^6$	13.493	1.74	2.2	2.49	0.51	1.78
8	166	603	8487	$2.964e^6$	$2.598e^7$	$1.477e^7$	20.128	1.98	2.48	2.86	0.44	2.00
Kanban System [CT96]												
3	64	137	2509	$2.969e^4$	$1.527e^5$	$8.092e^4$	0.088	1.58	1.94	5.94	0.55	1.76
4	96	239	4645	$6.775e^4$	$1.065e^6$	$6.381e^5$	0.275	2.00	2.35	2.58	0.47	2.15
5	96	366	7434	$1.323e^5$	$1.065e^6$	$6.381e^5$	0.674	1.73	2.08	2.33	0.48	1.88
6	96	519	10856	$2.363e^5$	$2.200e^6$	$1.280e^6$	1.457	1.58	1.93	2.16	0.58	1.71
7	96	697	14875	$3.933e^5$	$4.258e^6$	$2.416e^6$	2.929	1.46	1.81	2.03	0.59	1.84
Flexible Manufacturing System (FMS) [CT93]												
3	68	668	13131	$5.969e^4$	$6.108e^5$	$2.924e^5$	0.305	1.95	2.40	2.41	0.39	1.52
4	90	1654	38124	$1.836e^5$	$2.769e^6$	$1.304e^6$	1.386	2.19	2.71	2.66	0.34	1.34
5	94	3377	91448	$4.524e^5$	$9.388e^6$	$4.007e^6$	4.430	2.04	2.56	2.60	0.44	1.13
6	96	5974	$1.905e^5$	$9.714e^5$	$2.852e^7$	$1.106e^7$	14.326	1.91	2.39	2.49	0.53	0.83
7	98	9738	$3.497e^5$	$1.875e^6$	$8.323e^7$	$3.097e^7$	41.785	1.90	2.37	2.40	0.54	0.62
8	118	15179	$6.149e^5$	$3.383e^6$	$2.560e^8$	$9.291e^7$	126.821	2.12	2.64	2.62	0.47	0.55
Tandem Queueing Network [HMKS99]												
63	26	19	173	$2.297e^4$	$7.148e^5$	$1.927e^5$	0.443	0.42	1.40	1.68	0.84	1.23
64	30	16	174	$2.129e^4$	$7.826e^5$	$2.273e^5$	0.469	1.06	1.68	2.23	0.88	1.45
127	30	22	204	$5.922e^4$	$3.406e^6$	$8.841e^5$	2.338	0.41	1.40	1.96	0.80	1.14
128	34	18	201	$5.398e^4$	$3.202e^6$	$9.113e^5$	2.376	1.06	1.68	2.60	0.94	1.33
255	34	25	235	$2.354e^5$	$1.424e^7$	$3.655e^6$	12.353	0.40	1.40	1.59	0.58	1.11
256	38	20	228	$2.647e^5$	$1.531e^7$	$4.202e^6$	12.729	1.05	1.68	1.72	0.73	1.18
511	38	28	266	$8.576e^5$	$6.549e^7$	$1.616e^7$	61.146	0.39	1.40	1.51	0.70	1.05
512	42	22	255	$8.143e^5$	$7.067e^7$	$1.836e^7$	62.907	1.05	1.68	1.95	0.66	1.17

[a]The figures of the ZDD-based version were considered as 100%, i.e. values above 1 for $r_R, r_T, r_{peak}, r_{time}$ and below 1 for r_{chr} indicate that the ZDD-based version is superior to the ADD-based version.

Table 1: Empirical comparison of ADDs and ZDDs for various case studies

(A) Möbius [DCC+02]

N	r_{time}	r_{c2ut}	r_{c2ct}	r_{peak}
FTMP				
2	0.711	0.764	0.698	0.366
3	2.668	3.571	3.429	0.280
4	4.868	6.754	5.714	0.248
5	11.947	22.065	17.24	0.294
6	73.533	10.808	10.158	0.217
CP				
3	2.626	2.915	3.602	0.243
4	2.272	2.437	3.756	0.291
5	2.420	2.564	3.085	0.313
6	2.670	2.988	3.494	0.411
7	3.484	4.387	4.912	0.498
8	6.018	7.197	8.005	0.649

N	r_{time}	r_{c2ut}	r_{c2ct}	r_{peak}
Kanban				
3	0.898	1.029	1.119	0.413
4	2.047	2.367	2.672	0.462
5	3.792	4.419	5.112	0.506
6	8.259	9.678	11.352	0.560
7	14.667	17.221	20.033	0.602
FMS				
3	0.738	0.736	0.838	0.567
4	1.005	1.003	1.12	0.545
5	1.295	1.228	1.346	0.528
6	1.283	1.362	1.419	0.510
7	1.220	0.8	1.347	0.492
8	1.085	0.653	1.236	0.481

(B) CASPA [KSW04]

r_{time}	r_{peak}	r_{iter}
Kanban		
1.264	1.332	4.300
1.403	1.900	4.385
1.848	2.609	4.438
2.073	3.269	4.475
2.470	4.143	4.500
FMS		
1.148	0.966	4.167
1.221	1.138	4.125
1.575	1.359	4.100
1.702	1.581	4.083
1.839	1.784	4.071
2.303	2.109	4.063

Table 2: Empirical comparison of the two variants of symbolic reachability analysis

give these figures in order to illustrate another important effect, the influence of the variable ordering on the DD sizes and thus the time for manipulating them. Under a different variable ordering, the ZDD-based representation delivers much better run-times (cf. col. 10 and 14 of Table 3.C).

4.2 Reachability analysis

The number of explicitly explored and encoded transitions under the activity-local scheme is very low (see e.g. col. 3 and 4 of Table 3.A). Consequently it is not very surprising that under the activity-local scheme, similar to the fully symbolic approaches, most of the execution time is consumed by symbolic reachability analysis. The portion of time spent for this symbolic reachability analysis differs, of course, for different models. For instance, for the Kanban and FTMP model one only spends about 70% on reachability analysis, whereas for the FMS and CP model 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 *CacheLookup* function accesses the computed table in order to fetch results from previous recursions of the `Apply`- or `Abstract`-algorithm. Table 2 compares standard bfs- with our new quasi-dfs reachability algorithm. The data is based on the different run-times, the number of calls to *UniqueInter* (c2ut) and *CacheLookup* (c2ct), and on the peak memory requirements (*peak*). In order to simplify the comparison, we only give ratios by norming everything to the figures of the new variant. Table 2.A shows the figures for the ZDD-based implementation as realized within Möbius. While the new variant consumes more peak memory, it involves much fewer calls to *UniqueInter* and *CacheLookup*, which makes it substantially faster. Table 2.B shows results obtained from a realization of the new reachability scheme within the tool CASPA,

which is based on ADDs. Even though the number of iteration of the outer `do-until` (Fig. 5) loop is reduced by a factor of about 4 (r_{iter} in Table 2.B), the quasi-dfs scheme only halves the run-time. This might be a consequence of the very compact encodings of each state by CASPA, since CASPA employs a dense enumeration scheme of submodel states, leading to much "flatter" DD-structures, i.e. DDs with fewer Boolean variables, than other implementations, e.g. PRISM or our Möbius implementation. Thus it is not surprising, that even under CASPA the new scheme for symbolic reachability analysis becomes more advantageous the larger the generated DDs are, indicated by the growing figures of column one and two of Table 2.B.

The algorithm of Fig. 5.B leaves room for variation, e.g. one could update the set of unexplored states outside the inner `for` loop, one could use all reached states for exploration, rather than only the newly reached ones, etc.. Surprisingly we experienced that the activity-wise refinement is the main source of run-time reduction. An early updating for Z_U, as realized by our scheme of Fig. 5.B, often plays a minor role only, which of course depends on the employed high-level model. Therefore we conclude, that one should avoid operating directly on large DD structures. It is much better to explicitly sequentialize the operations and operate on smaller structures.

4.3 Assessment of the activity-local scheme

In order to make a fair comparison, we used the same two Möbius model specifications as in [DKS03], namely the scalable Fault-tolerant multiprocessor model (FTMP) [SM92] and the Courier protocol (CP) [WL91].[9] To the best of our knowledge, these models are currently the only ones where run-time data for the MDD-based approach under Möbius is available. The results of [DKS03] were obtained on an AMD Athlon 2400 with 1.5 GByte of RAM, whereas our own experiments were run on a Pentium 4 with 3 GHz and 1 GByte of RAM. Table 3.A shows the basic figures for the two models. For simplicity we once again provide only ratios for run-time[10] and memory consumption, where we normed everything to the figures of the the activity-local scheme. Our activity-local approach is significantly faster than the MDD-based approach, especially in case of the FTMP example. This shows that our partial-order style strategy of exploring only paths of dependent activities pays off, especially for models without strongly modular structure (cf. col. 3 and 4 of Table 3.A and col. 3 of Table 3.C). Furthermore the memory requirement for storing the set of reachable states is better as well (r_{mem4R}), except in case of the FTMP model with scaling parameter $N = 6$.

The size of DDs, and thus the effectiveness of the symbolic manipulations, is strongly influenced by the ordering of the Boolean variables. Given that symbolic reachability analysis is the dominant factor of run-time, which is also the case for the BDD-based approaches as realized in the tools PRISM and CASPA, and given that the variable ordering might even depend on the model specification itself [KSW04], it is clear that a comparison of different BDD-based tools needs to be conducted with great care.

[9]The FTMP model as specified under Möbius is a worst-case scenario for methods based on composition, since it has very little submodel-local behavior. It therefore nicely illustrates the advantages of our approach, which does not require any particular model structure.

[10]The timing information in [DKS03] includes time for state lumping, but since this is below 0.3% of the overall time we can safely neglect it.

(A) Comparison to the MDD-based scheme of [DKS03]

N	# states	# trans.	# $trans_e$	$r_{\text{mem}4R}$	r_{time}
CP					
3	$2.3812e^6$	$1.3104e^7$	94	1.52	0.46
4	$9.7102e^6$	$5.7005e^7$	142	2.53	1.15
5	$3.2405e^7$	$1.9988e^8$	206	4.17	3.53
6	$9.3302e^7$	$5.9818e^8$	289	6.70	9.55
7	$2.3965e^8$	$1.5858e^9$	394	10.61	25.72
8	$5.6182e^8$	$3.8166e^9$	524	16.03	51.67
FTMP					
2	256932	$1.6978e^6$	688	1.14	2.78
3	$1.2408e^8$	$1.1513e^9$	1548	1.08	17.05
4	$5.5039e^{10}$	$6.6113e^{11}$	2752	1.06	57.22
5	$2.3549e^{13}$	$3.4847e^{14}$	4300	1.05	65.36
6	$9.9082e^{15}$	$1.7463e^{17}$	6192	0.36	830.35

(B) Run-time data produced by PRISM [Pri]

N	# states	# trans.	# peak	# $iter_R$	t_g in secs.
Kanban					
3	$5.84e^4$	$4.464e^5$	11938	43	0.11993
4	$4.5448e^5$	$3.9799e^6$	35264	57	0.37980
5	$2.5464e^6$	$2.446e^7$	59731	71	0.77923
6	$1.1261e^7$	$1.1571e^8$	93464	85	1.49230
7	$4.1645e^7$	$4.5046e^8$	135514	99	2.38614
8	$1.3387e^8$	$1.5079e^9$	246750	113	4.99712
FMS					
3	$6.52e^3$	$3.7394e^4$	26574	25	0.23584
4	$3.591e^4$	$2.3712e^5$	76043	33	0.73170
5	$1.5271e^5$	$1.1115e^6$	134145	41	1.56822
6	$5.3777e^5$	$4.2057e^6$	226441	49	2.96269
7	$1.6394e^6$	$1.3553e^7$	348540	57	6.02259
8	$4.4595e^6$	$3.8534e^7$	679426	65	23.08034

(C) Comparing the activity-local approach to PRISM [Pri]

			# ADD-nodes for		ADD + bfs-reach.		ADD + quasi-dfs reach.			ZDD + quasi-dfs reach.			
N	n	# $trans_e$	M_R	M_T	r_{peak}	r_{time}	r_{peak}	r_{iter}	r_{time}	r_R	r_T	r_{peak}	r_{time}
Kanban													
3	64	252	131	2474	12.514	3.15	3.87	0.37	0.95	0.56	0.47	1.78	0.44
4	96	740	260	4898	16.058	4.81	3.52	0.35	1.09	0.44	0.39	1.28	0.41
5	96	1,860	320	6308	19.485	6.14	3.49	0.34	1.10	0.50	0.44	1.39	0.47
6	96	4,116	388	7876	22.047	8.85	3.51	0.33	1.07	0.55	0.47	1.51	0.53
7	96	8,232	457	9521	24.510	9.64	3.60	0.32	1.20	0.59	0.50	1.64	0.65
8	128	15,194	730	14698	27.193	15.59	3.73	0.32	1.44	0.47	0.41	1.35	0.62
FMS													
3	68 80	100	289	6,622	3.257	0.80	2.23	0.52	0.50	0.38	0.26	0.92	0.24
4	90 110	180	577	16,227	4.960	1.30	2.00	0.48	0.49	0.32	0.21	0.73	0.21
5	94 110	290	798	26,662	7.348	1.63	2.01	0.44	0.44	0.37	0.21	0.79	0.20
6	96 110	434	1038	40,274	9.487	2.26	1.91	0.43	0.39	0.40	0.22	0.80	0.19
7	98 110	616	1297	56,853	12.750	2.08	1.82	0.40	0.29	0.44	0.22	0.82	0.17
8	118 140	840	2021	96,647	15.736	1.24	1.63	0.40	0.14	0.36	0.18	0.64	0.09

Table 3: Comparison of the activity-local scheme and other symbolic SG representation approaches

The probabilistic model checker PRISM [Pri] implements a fully symbolic compositional SG generation scheme. We decided to employ PRISM for the remaining two case studies (Kanban and FMS), since – similar to our own implementation – it allows the user to specify the SV ordering and it encodes each SV by a Boolean vector. Table 3.B gives the basic data of the models, as well as some statistics obtained from PRISM: $iter_R$ refers to the number of iterations needed for symbolic reachability analysis (basically algorithm (A) in Fig. 5, line 4 - 8), and t_g refers to the CPU time for the whole process of reachable SG generation. The number of nodes required for encoding the set of states and transitions is given in Table 3.C (col. 4 + 5). For the Kanban model our implementation encodes the model in 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 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. 2 of Table 3.C for details), leading to smaller DD structures, whose sizes are given in col. 3 and 4 of Table 3.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:

1. In the first setting we combined the activity-local scheme with ADD-based SG representation, where the standard bfs. symbolic reachability analysis was employed.
2. In the second setting we replaced the standard scheme for symbolic reachability analysis by our new quasi-dfs symbolic exploration scheme.
3. In the final setting we switched to the ZDD-based SG representation.

The figures of the different settings are shown in column 6 to 14 of Table 3.C, where we normalized all data to the figures produced by PRISM.

Form Table 3.C one can conclude, that the explicit handling of transitions induces a non-neglible run-time overhead, albeit this number ($trans_e$) is reduced to a small fraction of the overall number of transitions to be symbolically represented (cf. #trans. and $\#trans_e$ in Tables 3.A-3.C). However this drawback is justified by two aspects:

1. The activity-local approach – in comparison with the fully symbolic ones – is not restricted to any specific model description method, which is of great importance for tools relying on a multi-formalism paradigm.
2. Unstructured monolithic models, such as the FTMP-model, can still be analyzed efficiently, where submodel-oriented compositional approaches may fail.

As shown by the last 7 columns of Table 3.C, our new algorithm for symbolic reachability analysis 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 FMS model, which was the largest model concerning memory requirement, consumed only 10.5 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.

5 Summary and Future Work

The work presented here consists of the following three main components:

1. We introduced and empirically evaluated ZDDs, which proved to be an excellent data structure for symbolic SG generation and representation.
2. We proposed a new algorithm for symbolic reachability analysis, organized as a sequential quasi-dfs scheme, and demonstrated its significant run-time savings.
3. We presented the activity-local SG generation scheme for generating the symbolic next-state functions by explicit SG exploration.

The scheme does not only yield compact symbolic representations, but also has the advantage that the SG only needs to be explicitly explored partially. Consequently the scheme leads to substantial run-time savings, especially in cases where the high-level model does not have a compositional structure or a fully symbolic method is not applicable. Besides this, ZDD-based SG representation, as well as the new scheme for symbolic reachability analysis can easily be integrated into existing BDD-based tools such as PRISM and CASPA, in order to improve run-time and reduce memory space.

Since we develop our implementations in the context of Möbuis, we are currently working on an efficient symbolic realization of the "Replicate" feature and on the symbolic treatment of reward variables. Another important step of our work is the development of efficient numerical analysis algorithms. Here we are almost finished with adapting the approach of [Par02] to the ZDD data structure, where the first results are very promising.

6 Acknowledgment

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

References

[AKN+00] L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic Model Checking for Probabilistic Processes using MTBDDs and the Kronecker Representation. In *TACAS'2000, LNCS 1785*, pages 395–410, 2000.

[BCL91] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, 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 *Application and Theory of Petri Nets 1999*, LNCS 1639, pages 6–25, 1999.

[CMS03] G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Tools and Algorithms for the Construction and Analysis of Systems (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 stocastic Petri nets, 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 the 12th Int. Conf. on Modelling Techniques and Tools (TOOLS 2002)*, pages 188–199. LNCS 2324, 2002.

[DKS03] S. Derisavi, P. Kemper, and W. H. Sanders. Symbolic State-space Exploration and Numerical Analysis of State-sharing Composed Models. In *Proc. Fourth Int. Conf. on Numerical Solution of Markov Chains*, pages 167–189, 2003.

[EFT93] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. *Distributed Computing*, 6(3):155–164, 1993.

[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.

[HKN+03] H. Hermanns, M. Kwiatkowska, G. Norman, D. Parker, and M. Siegle. On the use of MTBDDs for performability analysis and verification of stochastic systems. *Journal of Logic and Algebraic Programming*, 56(1-2):23–67, 2003.

[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 3rd Int. Workshop on Numerical Solution of Markov Chains*, pages 188–207. Prensas Universitarias de Zaragoza, 1999.

[KS02] M. Kuntz and M. Siegle. Deriving Symbolic Representations from Stochastic Process Algebras. In *Process Algebra and Probabilistic Methods (PAPM-PROBMIV'02)*, LNCS 2399, pages 1–22, 2002.

[KSW04] M. Kuntz, M. Siegle, and E. Werner. Symbolic Performance and Dependability Evaluation with the Tool CASPA. In *European Performance Engineering Workshop*, 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 the 2nd 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 *Sixth Int. Workshop on Performability Modeling of Computer and Communication Systems (PMCCS6)*, pages 15–18, 2003.

[Min93] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proc. of the 30th Design Automation Conference*, pages 272–277, Dallas (Texas), USA, June 1993. ACM Press.

[Min01] A. Miner. Efficient solution of GSPNs using matrix diagrams. In *Petri Nets and Performance models (PNPM)*, pages 101–110. IEEE Computer Society Press, 2001.

[Min04] A. Miner. Saturation for a general class of models. In *Proc. of the First Int. Conf. on the Quantitative Evaluation of Systems (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, 1997. Technical Report of Departament Arquitectura de Computadors (UPC) DAC/UPC Report No. 97/8.

[PRCB94] E. Pastor, O. Roig, J. Cortadella, and R.M. Badia. Petri Net Analysis Using Boolean Manipulation. In R. Valette, editor, *Proc. of the 15'th Int. Conf. on Application and Theory of Petri Nets (APN'94), Zaragosa, Spain*, 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 *Fourth Int. Workshop on Performability Modeling of Computer and Communication Systems (PMCCS4)*, 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.

[Som98] F. Somenzi. CUDD: Colorado University Decision Diagram Package, Release 2.3.0. User's Manual and Programmer's Manual, September 1998.

[WL91] M. Woodside and Y. Li. Performance Petri net analysis of communications protocol software by delay-equivalent aggregation. In *In Proc. of the 4th Int. Workshop on Petri Nets and Performance Models (PNPM)*, pages 64–73, 1991.