Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 223 (2008) 137–151
www.elsevier.com/locate/entcs

A New Algorithm for Partitioned Symbolic Reachability Analysis

Kai Lampka[1]

Computer Engineering and Communication Networks Lab.
ETH Zurich, Switzerland

Abstract

Binary Decision Diagrams (BDDs) and their multi-terminal extensions have shown to be very helpful for the quantitative verification of systems. Many different approaches have been proposed for deriving symbolic state graph (SG) representations from high-level model descriptions, where compositionality has shown to be crucial for the efficiency of the schemes. Since the symbolic composition schemes deliver the potential SG of a high-level model, one must execute a reachability analysis on the level of the symbolic structures. This step is the main resource of CPU-time and peak memory consumption when it comes to symbolic SG generation. In this work a new operator for zero-suppressed BDDs and their multi-terminal extensions for carrying out (partitioned) symbolic reachability analysis is presented. This algorithm not only replaces standard BDD-based schemes, it even makes symbolic composition as found in contemporary symbolic model checkers such as Prism and Caspa obsolete.

Keywords: Binary Decision Diagrams and algorithms, symbolic reachability analysis, quantitative verification of systems

1 Introduction

In our work we focus on the *quantitative* verification of systems, where symbolic techniques, i.e. techniques based on Decision Diagrams are still state-of-the-art and employed in probabilistic model checkers. Decision diagrams (DDs) are directed acyclic graphs for representing finite functions. Multi-terminal Binary Decision Diagrams (MTBDDs) [1] are among the most efficient techniques for the state graph (SG) based quantitative analysis of large and complex systems. For obtaining a compact and readable description of systems to be analyzed, one commonly employs a (Markovian) description techniques, such as a Stochastic Process Algebra or Generalized Stochastic Petri Nets, among many others, rather than directly specifying the system's behavior by a SG. Many different approaches have been proposed for deriving symbolic representations of SGs from their high-level descriptions. Roughly

[1] Email: lampka@tik.ee.ethz.ch

doi:10.1016/j.entcs.2008.12.036

speaking, the proposed schemes can be divided into the classes of monolithic - and compositional approaches. Applying a compositional scheme means that the SG of the overall model is constructed from smaller components, commonly from symbolic representations of the SGs of submodels or partitions (submodel- or partition-local SGs). Compositionality turned out to be crucial, since (a) it reduces the run-time, as not all sequences of independent activities have to be extracted explicitly and (b) it induces regularity on the symbolic structures and thus reduces the peak memory consumption. However, symbolic composition is commonly based on symbolic cross-product computation, it therefore delivers the potential SG underlying a high-level model. For restricting the potential transition system to the set of reachable states and transitions, ones must execute a (symbolic) reachability analysis, commonly carried out on the level of the symbolic structures, which represent the potential transition system of the overall model. These symbolic reachability schemes are the main source of CPU time - and memory consumption when it comes to the construction of symbolic SG representations.

1.1 Contribution and related work

Based on Bryant's well known `Apply` algorithm [3] different symbolic algorithms have been proposed. Following these traditions this paper introduces a new operator for carrying out partitioned symbolic reachability analysis. In its final version this operator makes symbolic composition as found in contemporary compositional symbolic SG generation methods obsolete, –at least as far as the insertion of identity structures is concerned. The presented approach is tailored for compositional SG generation methods and zero-suppressed BDDs (ZBDDs) and their multi-terminal extension (ZDDs) [9,12]. But, it can easily be adapted to standard BDDs [2,3] and their multi-terminal derivatives [1]. In total it might find therefore its application in tools like the (compositional) symbolic model checkers Caspa [8] and Prism [14].

1.1.1 Multi-step reachability schemes

Following [4] earlier work [11,9] (re-) developed a scheme for partitioned symbolic reachability analysis. Like standard breadth-first-search (bfs) symbolic reachability analysis, this scheme can be considered as symbolic **multi-step** approach, which means that it sequentially executes a number of operators for computing the one-step reachability set with respect to a transition function. But contrary to the standard approach, it executes the symbolically represented transition functions in an activity-wise manner, rather than all at once, This strategy enables one to employ an *early-update strategy* on the set of states to be explored in the next step, leading to a *quasi-depth-first-search* (q-dfs) scheme rather than implementing a pure bfs - or dfs scheme. A very similar approach, customized for k-bounded Petri nets and a fully symbolic SG generation technique [2] has already been introduced as chaining in [15]. However, contrary to [11,9] and to the approach presented here, the technique

[2] Fully symbolic means that the high-level model description technique possesses a symbolic execution semantics.

of [15] is monolithic and therefore most likely to be hampered by large peak memory requirements.

1.1.2 Single-step approaches

The author of [19] gives a highly detailed overview on techniques related to symbolic reachability analysis and computation of *relational products* when one employs BDDs for representing transition relations. The workings [5] and [13] also present algorithms for computing relational products, where in case of the former BDDs and in case of the latter ZBDDs are addressed. Similar to these contributions our new algorithm combines conjunction and existential quantification into a single algorithm. However, the here presented work differs with respect to the state graph generation method and thus leads to a different algorithm. Like many other symbolic verification tools we also emphasize the usage of high-level modelling methods for describing systems, thus in our approach we assume compositionally constructed transition relations. In such a context one commonly inserts identity structures on the position of the sub-model-independent state variables before combining the individual transition relations and executing a symbolic reachability analysis. The algorithm as introduced here, makes this unnecessary, since it advises an identity semantic when recursing on such variables.

1.2 Organization

Sec. 2 introduces the basic setting and makes the reader familiar with Decision Diagrams (DDs). Sec. 3 introduces the new symbolic reachability algorithm. Its practical feasibility is investigated in Sec. 4, where standard benchmarking models as known from the literature are analyzed. Sec. 5 concludes the paper.

2 Background Theory

2.1 Model world

Powerful methods as known from the functional analysis of systems, have been extended to the Markovian case. In the following it is assumed that the reader has basic familiarity with high-level (Markov) model description techniques, such as Generalized Stochastic Petri Net (GSPNs), or Stochastic Process Algebra (SPA) to name only few of them. It is assumed that each high-level model M consists of a finite ordered set of discrete variables commonly denoted as state variables (SVs) with $\mathfrak{s}_i \in \mathfrak{S}$, and a finite set of activities ($\mathcal{A}ct$). Each $\mathfrak{s}_i$ records the number of tokens in a place, the state of the program or process counter, the values of the process parameters, etc.. By executing activities, one at a time, the model evolves from one state to another, where each SV $\mathfrak{s}_i$ takes an arbitrary value from $\mathbb{N}$, and each transition is equipped with the activity's label and its (exponential) execution rate. This may allow to map a high-level model to a finite transition system or state graph (SG), where this process is commonly denoted as SG generation. A SG consists of a (finite) set of states ($\mathbb{S}$), and a transition function. A transition function

is a mapping $\Delta : \mathbb{S} \mapsto 2^{\mathbb{S}}$, yielding a predecessor/successor relation $\rightarrow \subseteq \mathbb{S} \times \mathbb{S}$ on the set of states. If each directed edge is labeled with a symbol $l \in \mathcal{A}ct$ one speaks of a labeled transition system (*LTS*), yielding the relation $\rightarrow \subseteq \mathbb{S} \times \mathcal{A}ct \times \mathbb{S}$. According to the above discussion transitions are not only equipped with labels, but also with rates $r \in \mathbb{R}_0^+$. This gives one a stochastic *LTS* (*SLTS*) $\rightarrow \subseteq \mathbb{S} \times \mathcal{A}ct \times \mathbb{R}_0^+ \times \mathbb{S}$. From a *SLTS* S a Continuous Time Markov Chain (CTMC) can be derived in a straight forward manner. For exemplification one may refer to part (A) and (B) of Fig. 1 which show a simple SPN and its SLTS.

Compositionality has turned out to be crucial for the effective employment of symbolic SG generation techniques. Therefore it is assumed that high-level models are somehow compositionally structured, where analogously to contemporary compositional, symbolic SG generation schemes composition is assumed to be achieved via *activity synchronization*, which is the joint execution of dedicated activities among the model's partitions or via a *joining of SVs*, which is the merging of submodels via the sharing of dedicated SVs. [3] Compositionality allows one to group activities and SVs, obtaining a set of partition- or submodel-local dependent SVs ($\mathfrak{S}_P^D$) and a set of partition- or submodel-local independent SV ($\mathfrak{S}_P^I$) for each partition or submodel P of a high-level model $\mathcal{M}$.

2.2 Zero-suppressed MTBDDs (ZDDs)

Let $\mathbb{B} = \{0, 1\}$ be the set of Booleans, $\mathbb{N} = \{0, 1, 2, \ldots\}$ the set of naturals, and $\mathbb{R}$ the set of reals and let $\mathbb{D}$ be a finite set of function values (here $\mathbb{D} \subset \mathbb{R}$). Let $\mathcal{V}$ be some global (finite) set of Boolean variables on which a strict total ordering π is defined. The set of variables $\mathcal{F} := \{\mathsf{v}_1, \ldots, \mathsf{v}_n\} \subseteq \mathcal{V}$ employed in a Boolean function f is denoted as the set of function or input variables of f. Variable v_i is essential for a Boolean function if and only if at least for one assignment to the variables of f it holds that $f(\mathsf{v}_1, \ldots, \mathsf{v}_{i-1}, 0, \mathsf{v}_{i+1}, \ldots, \mathsf{v}_n) \neq f(\mathsf{v}_1, \ldots, \mathsf{v}_{i-1}, 1, \mathsf{v}_{i+1}, \ldots, \mathsf{v}_n)$. Otherwise the variable v_i is not essential. A non-essential variable is also commonly denoted as *don't-care* (*dnc*) variable. In the following n-ary pseudo-Boolean functions are considered, i.e. functions of the type $f : \mathbb{B}^n \mapsto \mathbb{D}$.

A reduced ordered ZDD is a tuple Z=($\mathcal{K}_{NT}$,$\mathcal{K}_T$,$\mathcal{V}_\mathsf{Z}$,π,$\mathtt{var}$,$\mathtt{then}$,$\mathtt{else}$,$\mathtt{value}$,*root*) where

(1) $\mathcal{K}_{NT}$ is the set of non-terminal - or inner nodes and $\mathcal{K}_T$ the set of terminal nodes, where $|\mathcal{K}_T| \geq 1$ and $\mathcal{K}_{NT} \cap \mathcal{K}_T = \emptyset$.

(2) $\mathcal{V}_\mathsf{Z} = \{\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n\} (\subseteq \mathcal{V})$ is a finite (possibly empty) set of Boolean variables, and $\mathtt{t} \notin \mathcal{V}_\mathsf{Z}$ is a pseudo-variable, labelling the terminal nodes. [4] Since the elements of $\mathcal{V}_\mathsf{Z}$ are ordered, we will also often employ a vector notation, e.g. $\vec{\mathsf{x}}$.

(3) π is a strict total ordering on the elements of $\mathcal{V}_\mathsf{Z} \cup \{\mathtt{t}\}$, where $\forall \mathsf{x}_i \in \mathcal{V}_\mathsf{Z} : \mathsf{x}_i < \mathtt{t}$.

(4) $\mathtt{var} : \mathcal{K}_{NT} \cup \mathcal{K}_T \mapsto \mathcal{V}_\mathsf{Z} \cup \{\mathtt{t}\}$ such that $\forall k \in \mathcal{K}_{NT} \cup \mathcal{K}_T : \mathtt{var}(k) = \mathtt{t} \Leftrightarrow \ k \in \mathcal{K}_T$.

(5) $\mathtt{then} : \mathcal{K}_{NT} \mapsto \mathcal{K}_{NT} \cup \mathcal{K}_T$ such that $\forall n \in \mathcal{K}_{NT} : \mathtt{var}(n) < \mathtt{var}(\mathtt{then}(n))$.

(6) $\mathtt{else} : \mathcal{K}_{NT} \mapsto \mathcal{K}_{NT} \cup \mathcal{K}_T$ such that $\forall n \in \mathcal{K}_{NT} : \mathtt{var}(n) < \mathtt{var}(\mathtt{else}(n))$.

(7) $\mathtt{value} : \mathcal{K}_T \mapsto \mathbb{D}$, where $\mathbb{D} \subset \mathbb{R}$.

(8) $\mathtt{root} \in \mathcal{K}_{NT} \cup \mathcal{K}_T$.

and the following reduction rules apply:

[3] We differ between partitions and submodels, since in case of the former the sets of SVs among the modules are disjoint, where in case of submodels this might not be the case.

[4] This (pure technical) extension allows one to include the terminal nodes into the ordering on the elements of $\mathcal{V}_\mathsf{Z}$.

(1) Isomorphism-free rule: There are no isomorphic nodes; i.e.
$\forall n, m \in \mathcal{K}_{NT}$:
$n \neq m \rightarrow (\mathtt{var}(n) \neq \mathtt{var}(m) \vee \mathtt{then}(n) \neq \mathtt{then}(m)$
$\vee \mathtt{else}(n) \neq \mathtt{else}(m))$ and
$\forall n, m \in \mathcal{K}_{T}$:
$n \neq m \rightarrow (\mathtt{value}(n) \neq \mathtt{value}(m))$.

(2) Zero-suppressing (*0-sup.*) reduction rule:
$\nexists n \in \mathcal{K}_{NT} : \mathtt{then}(n) \in \mathcal{K}_{T} \wedge \mathtt{value}(\mathtt{then}(n)) = 0$.

A combination of the Shannon expansion for Boolean functions [17] and the application of the *0-sup.* rule gives now that a ZDD's graph and a set of Boolean variables (together !) uniquely represents a Boolean function [9,12]. Therefore the notation $\mathsf{Z}<\mathcal{V}_{\mathsf{Z}}, \pi>$ will be employed, if sets of function variables and their ordering is from concern. In case $\mathbb{D} = \mathbb{B}$ the ZDD is a ZBDD, where also the notation 0-1 ZDD will be employed.

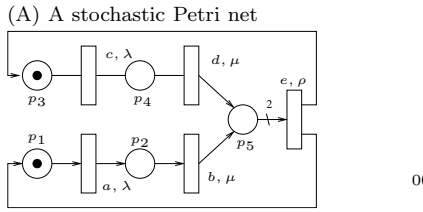
Within shared BDD-environments ZDD-nodes lose their uniqueness as soon as the represented functions are defined on different sets of input variables. To solve this problem, [9,12] introduced the concept of partially shared ZDDs (pZDDs) and algorithms for manipulating them. The basic idea of this approach is as follows: When working with pZDDs, i.e. with ZDDs having different set of input variables, one also iterates over the input variables of the operand pZDDs. This allows one to assign a specific semantics to each visited but skipped variable on the current path. The most important algorithms as far as it is from concern here are the followings:

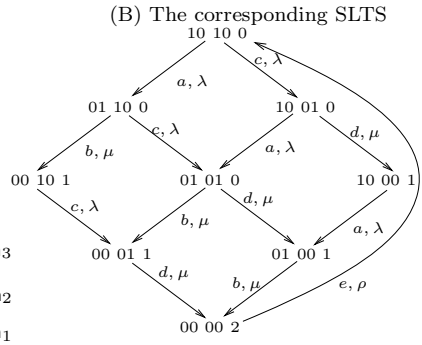
(1) Relabeling: The operation $\mathsf{Z}\{\vec{\mathsf{x}} \leftarrow \vec{\mathsf{y}}\}$ constructs a pZDD Y representing the function f_{Z} in case variable x_i is substituted by variable y_i, where $\forall \mathsf{y}_i \in \mathsf{Y}$: $\mathsf{y}_i \notin \mathsf{Z}$ must hold.

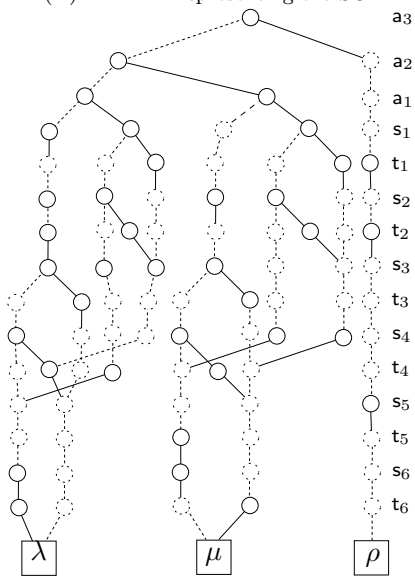
(2) The generic *p*`ZApply`-algorithm: A symbolic representation of a function $f := g \ \mathtt{op} \ h$ for `op` being a binary operator, e.g. $\mathtt{op} \in \{\wedge, \vee, *, \times, \ldots\}$ and for two functions g and h, not necessarily defined on the same set of function variables, can be computed by executing the generic *p*`ZApply`-algorithm. This algorithm takes the binary operator `op`, the respective operand pZDDs (i.e. their root nodes) and their sets of function variables $\mathcal{G}$ and $\mathcal{H}$ as input. The basic idea of the *p*`ZApply`-algorithm is that for a given pair of ZDDs and their sets of variables $\mathcal{G}$ and $\mathcal{H}$, a recursion for each variable $\mathsf{v} \in (\mathcal{G} \cup \mathcal{H})$ is executed. The recursive behavior depends on the type of the current variable, i.e. whether the current variable is a *0-sup.* input variable or a (skipped) non-function variable.

(3) The `Abstract`-algorithm: This algorithm implements the abstraction of a function from a variable `v`, i.e. the algorithm constructs a representation of the function $h := f|_{\mathsf{v}=0} \mathtt{op} \, f|_{\mathsf{v}=1}$, so that `v` is not a function variable for function h anymore. For $\mathtt{op} = \vee$ the `Abstract`-algorithm implements the existential -, and for $\mathtt{op} = \wedge$ it implements the universal quantification. It is straight forward to extend the `Abstract`-algorithm to the case of abstracting from sets of variables.

For simplicity we also allow Boolean operators to be applied to pZDDs, where 0-1 pZDDs are deliver as results.







(C) Binary encodings of the SLTS

l	$\vec{a}$ $a_1a_2a_3$	$\vec{s}$ $s_1s_2s_3s_4s_5s_6$	$\vec{t}$ $t_1t_2t_3t_4t_5t_6$	f_M
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

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

2.3 Symbolic SG representation

For symbolically representing a transition system T by a ZDD $\mathsf{Z}_T < \vec{a}, \vec{s}, \vec{t} >$ the following setting is defined: The variables of $\vec{a}$ (a-variables) of the ZDD hold the values of the binary encoded activity labels, variables of $\vec{s}$ (s-variables) the ones of the binary encoded source states, and variables of $\vec{t}$ (t-variables) the ones of the binary encoded target states of the transitions.

As common we define the following interleaved order on the variables: $a_m \prec \ldots \prec a_1 \prec s_1 \prec t_1 \prec \ldots \prec s_n \prec t_n$. For exemplification the reader may please refer to Fig. 1. The Boolean encodings of the transitions of the *SLTS* are given in table C. The 5 integer SVs are encoded by 6 pairs of Boolean variables (s_i, t_i). Part (D) shows the corresponding ZDD M, where the *0-sup.*-nodes are printed in dotted lines, –we did this for illustration purpose, in the actual graph of the ZDD these nodes are not present!– The rates of the transitions are stored in the terminal nodes. In the ZDD, a dashed (solid) line indicates the value assignment 0 (1) to the corresponding Boolean variable on the respective path.

3 New operator fo symbolic image computation

3.1 Preliminaries

Compositionality is crucial for the efficiency of a symbolic SG generation scheme. This is not only because it influences the efficiency of generating and encoding transitions, but also because it significantly influences the speed and memory consumption of symbolic reachability analysis. A compositional procedure, as found in tools like Caspa [8], Prism [14] or Möbius and its symbolic engine [11,9], one either directly exploits the hierarchic structure of compositionally constructed overall models or somehow decompose the latter. In any case one ends up with a set of symbolic representations (BDDs or ZBDDs), each representing a set of (submodel/partition-) local transitions. What follows next is the application of a (symbolic) composition scheme, which implements a synchronization over activity labels in a process algebra like style and/or implements the merging of state variables in the style of stochastic activity networks [16]. Details on the symbolic realization of these composition methods can be found in [6,18,7,10] among others. Independent of the employed procedure one may reveal the following similarities of the different schemes:

- The symbolic representation of a (submodel/partition-) local transition function takes only those SVs as input variables, which are in the dependency set of the activities encapsulated in the resp. submodel or partition, i.e. one solely encodes here the values of submodel- or partition-local (dependent) SVs $\mathfrak{S}_{\mathcal{P}}^{D}$.
- The symbolic structure $\tilde{\mathsf{Z}}_T <\vec{\mathsf{s}}, \vec{\mathsf{t}}, \pi>$, representing the potential transition system of the overall model is commonly obtained by cross-product computation of some local transition systems and some identity structures. The identity structures model the behavior on the positions of those SVs $\mathfrak{S}_{\mathcal{P}}^{I}$ which are not effected (independent) of the resp. submodel or partition.
- For restricting $\tilde{\mathsf{Z}}_T$ to the set of reachable transitions one commonly executes a symbolic reachability analysis. As known from the literature (cf. Sec. 1.1) this step should be organized in a partitioned manner.

To enforce a compositional setting the overall model is now assumed to be partitioned in an activity-wise manner, i.e. each of the high-level model's activities ($l \in \mathcal{A}ct$) is encapsulated in its own submodel, –any other partitioning would also be acceptable. This allows one to construct an individual (activity-)local transition function for each of these (activity-local) submodels, represented by a ZDD $\mathsf{Z}_l <\mathcal{V}_l^{\mathsf{D}}>$ (see [11,9] for details on this construction step). In such a setting $\mathcal{V}_l^{\mathsf{D}}$ and $\mathcal{V}_l^{\mathsf{I}}$ are those sets of Boolean variables, which encode the dependent or independent SVs of submodel l. The cross-product $\tilde{\mathsf{Z}}_l := \mathsf{Z}_l<\mathcal{V}_l^{\mathsf{D}}, \pi> \times \mathbb{1}(\mathcal{V}_l^{\mathsf{I}})$ delivers than the potential transition system as induced by submodel l, where $\mathbb{1}(\mathcal{V}_l^{\mathsf{I}})$ encodes the identity structures as mentioned above. [5] For carrying out symbolic reachability activity-labels

[5] Alternatively one could derive $\tilde{\mathsf{Z}}_l$ from $\tilde{\mathsf{Z}}_T$ as follows: $\tilde{\mathsf{Z}}_l := \tilde{\mathsf{Z}}_T \times \mathsf{A} <\vec{\mathsf{a}}>$, where in the above setting $\mathsf{A} <\vec{\mathsf{a}}>$ encodes activity label l, –or in case of a coarser partitioning all activity labels of a submodel l. Consequently the activity-wise partitioning as employed here is not mandatory, any other fragmentation is suitable, as long as one ends up with local transition functions and their sets of dependent and independent SVs, their Boolean counterparts resp..

are irrelevant, so that they can safely be removed. This is achieved by computing an existential quantification over all a-variables and over all local symbolic structures $\tilde{\mathsf{Z}}_l$. In such a setting and if the set of (unexplored) states is represented by a ZDD $\mathsf{Z}_{unex}{<}\vec{\mathsf{s}},\vec{\pi}{>}$ the one-step reachability set with respect to activity l can be computed by the following code fragment:

(0) $\mathsf{Z}_{tmp} := p\texttt{ZApply}(\wedge, \mathsf{Z}_{unex}, \tilde{\mathsf{Z}}_l)$
(1) $\mathsf{Z}_{tmp} := \texttt{ZAbstract}(\mathsf{Z}_{tmp}, \vec{\mathsf{s}}, \vee)$
(2) $\mathsf{Z}_{unex} := \mathsf{Z}_{tmp}\{\vec{\mathsf{s}} \leftarrow \vec{\mathsf{t}}\}$

In the above pseudo-code one extracts at first all transitions emanating from states contained in Z_{unex} (line (0)). Subsequently one eliminates the source states (line (1)) and re-labels the t-variables with t-variables (line (2)), which delivers the newly reached states encoded as source states. The first version of our new operator `ExecuteActivity()` implements these steps in a single operator.

3.2 The new scheme

When traversing the symbolic structures representing the unexplored states (Z_{unex}) and the potential, local transition functions ($\tilde{\mathsf{Z}}_l$) the new algorithm `Execute-Activity` executes a recursion for each variable $\in \mathcal{V}(= \vec{\mathsf{s}} \cup \vec{\mathsf{t}})$. Let v_c be the variable of the current recursion: if v_c is not skipped within the ZDDs Z_{unex} and $\tilde{\mathsf{Z}}_l$, the standard `Apply`-recursion rule is executed [3], otherwise the following case distinctions apply:

(1) Handling of structure Z_{unex}: Let variable v_c be a s-variable, i.e. it encodes a bit position of the source state. Since v_c is skipped, it must be a 0-assigned variable, due to the *0-sup.*-reduction rule. Concerning the recursive behavior the `else`-child to recurse on in the next step is the current node within Z_{unex} itself, whereas the `then`-child to recurse further with must be the terminal *0-node* (= semantics of a *0-sup.* node). In case the current variable v_c is a t-variable, it is non-decisive for ZDD Z_{unex}, since it holds a bit value of the binary encoded target state. In such cases a *don't-care* semantics must be applied, i.e. one recurses with the current node of Z_{unex} into the `else`- and `then`-branch.

(2) The handling of $\tilde{\mathsf{Z}}_l$ is straight-forward: in case v_c is skipped it must be *0-sup.* and the *0-sup.* recursion rule applies.

Part A of Fig. 2 shows ZDD Z_{unex} which represents the initial state of the SPN of Fig. 1.A, and ZDD $\tilde{\mathsf{Z}}_l$ representing the potential and local transition function as induced by high-level activity a of the SPN. In part C we depicted ZDD Z'_{unex} which represents the image of Z_{unex} with respect to transition function $\tilde{\mathsf{Z}}_l$. –One may already note that within $\tilde{\mathsf{Z}}_l$ positions referring to variables which are independent of the execution of activity a, here $\{\mathsf{s}_3, .., \mathsf{t}_6\}$, are filled with identity structures.– Fig. 2.B depicts the call tree of the new operator when recursing on the first 5 variables, where the parameter-lists of the individual function calls are also given. This parameter list contains the current node of the structure representing the set

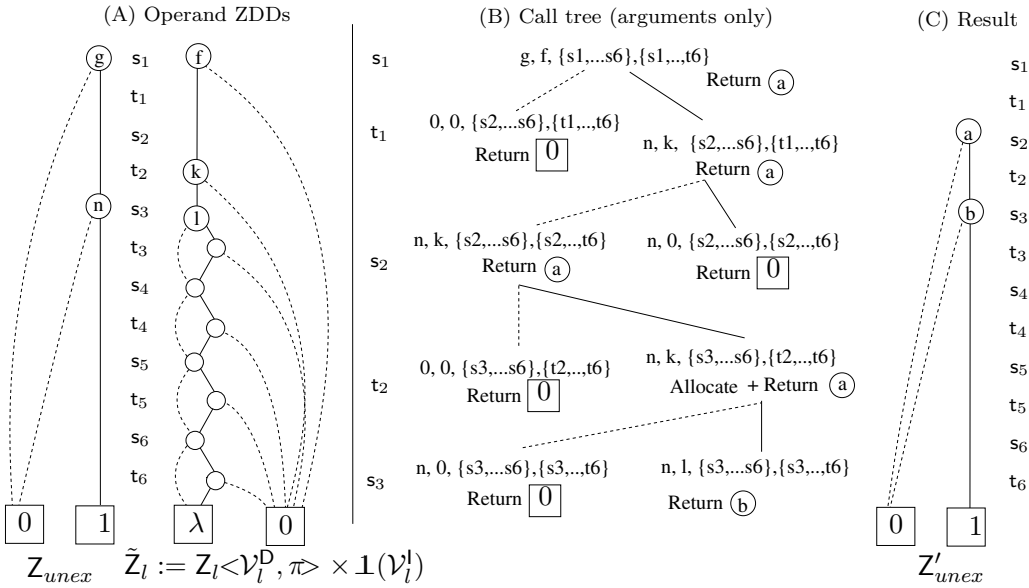


Figure 2. ZDD-traversal for computing the one-step reachability set

of source states (Z_{unex}), the current node of the ZDD representing the potential transition function ($\tilde{Z}_l$) as well as the individual sets of function variables of these ZDDs. The first `else`-branch recursion is than called with the terminal *0-node*, whereas the `then`-branch is executed with node n and k (cf. Fig. 2.A and B). The next variable to be visited within this `then`-branch is variable $\mathtt{t}_1$, where a *dnc*-semantics in case of Z_{unex} and a *0-sup.*-semantics in case of $\tilde{Z}_l$ applies. I.e. within the new `else`-branch one recurses with n and k, whereas in case of the new `then`-branch a recursion with node n and the terminal 0-node is started. In cases where a terminal 0-node is encountered the recursion can terminate by returning the terminal 0-node as result. In all other cases the recursion basically continues until the terminal non-zero nodes of Z_{unex} and $\tilde{Z}_l$ are reached (see line 1 - 5 of Algo. `ExecuteActivity`, Fig. 3). When returning from the recursion one either allocates a node or abstracts from the (current) variable. I.e. when the recursion returns at a `s`-variable the 1- and 0-successor of the potential node must be merged, since the operator must abstract from source states. In case the recursion returns at a `t`-level a node for the *preceding* `s`-variable is allocated. This behavior can be found in the call-tree of Fig. 2.B at level 4, i.e. at the level of variable $\mathtt{t}_2$: There node a is allocated (labelled with variable $\mathtt{s}_2$) and past back as result of the computation. The caller of this recursive step computes then p`ZApply`$(+, 0, a)$ $(= a)$ and returns it as result to its own caller. This functionality is encoded within line 55-60 of algorithm `ExecuteActivity` as illustrated in Fig. 3. However before we go into detail of its pseudo-code, another recursion-rule shall be covered. This rule will make the insertion of identity structures as found in contemporary

ExecuteActivity
Parameters: $Node: g, f, VarSet: \mathcal{G}, \mathcal{F}, \mathcal{V}, Bool: skipFlag$

```
(0)     Node res;

/* Check terminal condition */
(1)     IF g = 0-node ∨ f = 0-node
            THEN RETURN 0-node;
(2)     ELSE IF v_c = ∅
            THEN RETURN 1-node;
(3)     ELSE IF skipFlag THEN
(4)         IF g, f ∈ K_T
            THEN RETURN 1-node;
(5)     ELSE IF g = f ∧ F = V
            THEN RETURN 1-node;

/* Check for pre-computed results */
(6)     IF v_c ∈ s⃗ ∨ v_c ∈ F THEN
(7)         res := CacheLookup(f, F, g, G);
(8)     IF res ≠ ϵ RETURN res;

/* Prepare recursive step */
(9)     Node f1, f0, g1, g0, T, E;
(10)    var     v_g := min(G),
                v_f := min(F), v_c := min(V);

/* Obvious both variables contained */
(11)    IF v_f = v_c ∧ v_g = v_c THEN
(12) (      f1 := then(f);
(13)        f0 := else(f);
(14)        g1 := then(g);
(15)        g0 := else(g);

/* only var skipped in g, */
/* thus v_c is func.var for f */
(16)    ELSE IF v_f = v_c THEN
(17)        f1 := then(f);
(18)        f0 := else(f);
(19)        g0 := g;
/* is t-var and dnc in g or s-var and 0-sup. */
(20)        IF v_c ∈ t⃗ THEN
(21)            g1 := g;
(22)        ELSE g1 := 0-node:

/* var skipped within f => must be s-var */
(23)    ELSE IF v_g = v_c THEN
(24)        g1 := then(g);
(25)        g0 := else(g);
(26)        f0 := f;
/* skipped but non-func. s-var */
(27)        IF v_f ∉ F THEN
(28)            f1 := f;
/* fast fwd in else-branch, */
/* disabled for then-branch */
(29)            v_k := succ(v_c);
(30)            skipFlag := false

/* skipped var is assumed to be 0-sup. */
/* and func. var */
(31)        ELSE f1 := 0-node;

/* obviously level skipped in both graphs */
(32)    ELSE

/* Fast Fwd to node with smallest var */
(33)        IF skipFlag THEN
(34)            v_c := min(v_g, v_f);
(35)            WHILE v_c < max(V) DO V := V \ max(V); END
(36)            WHILE v_c < max(F) DO F := F \ max(F); END
(37)            WHILE v_c < max(G) DO G := G \ max(G); END
(38)            RETURN ExecuteActivity(g, G, f, F, V, true);
/* default: skipped var is non-func. s-var */
(39)        f0 := f;
(40)        f1 := f;
(41)        g0 := g;
/* default: skipped level in g is 0-assigned */
(42)        g1 := 0-node;
/* is t-var thus dnc in g */
/* and assume 0-sup. t-var in f */
(43)        IF v_c ∈ t⃗ THEN
(44)            g1 := g;
(45)            f0 := 0-node;
/* fast fwd in else-branch if non-func. s-var */
(46)        ELSE
(47)            v_k := succ(v_c);
/* skipped but func. var */
(48)        IF v_c ∈ s⃗ THEN
(49)            f1 := 0-node;
(50)            f0 := f;
(51)            v_k := v_c;

/* Remove variables from sets */
(52)    F' := F \ {v_c}), F'' := F \ {v_c, v_k});
(53)    G' := G \ {v_c}), G'' := G \ {v_c, v_k});
(54)    V' := V \ {v_c}), V'' := V \ {v_c, v_k});

/* go into recursion */
(55)    T := ExecuteActivity(g1, G'', f1, F'', V'', skipFlag));
(56)    E := ExecuteActivity(g0, G', f0, F', V', skipFlag));

/* allocate node if t-var, abstract if s var */
(57)    IF v_c ∈ t⃗ THEN
(58)        res := getZMTBDDNode(pred(v_c), T, E)));
(59)    ELSE
(60)        res := pZApply(+, T, E);

/* Cache pre-computes */
(61)    IF v_c ∈ s⃗ ∨ v_c ∈ F THEN
(62)        CacheInsert(res, f, F, g, G);

(63)    RETURN res;
```

Figure 3. New algorithm for computing one-step reachability set

composition schemes and as illustrated above unnecessary. Instead of operating on $\tilde{\mathsf{Z}}_l$ this new rule allows one to directly employ the local transition systems, their symbolic counterparts resp., when computing the one-step reachability set of a set of source states and with respect to a submodel l. The main idea of doing so is straightforward: when traversing Z_l, its non-function variables are handled according to an identity-semantics. –In the example of Fig. 2 this is the case for $\tilde{\mathsf{Z}}_l$ when hitting variables $\{\mathsf{s}_3, \ldots, \mathsf{t}_6\}$.

Let $\mathcal{V}$ be the set of all function variables for encoding the overall model's transition system ($\mathcal{V} := \vec{\mathsf{s}} \cup \vec{\mathsf{t}}$). Let $\mathcal{G} := \vec{s}$ and $\mathcal{F} \subseteq \mathcal{V}$ be the set of function variables for the symbolic structures Z_{unex} and Z_l (not $\tilde{\mathsf{Z}}_l$!). Let the variable of the current recursion be denoted v_c, the one-step reachability operator must cover the following additional case when accessing a non-function variable ($\mathsf{v}_c \notin \mathcal{F}$) within the symbolic

encoded transition function Z_l:

> If v_c is a `s`-variable one simply recurse with the current node into the `else`- and `then`-branch, where a node must be allocated for v_c only if there is a node allocated within Z_{unex}, which gives way for skipping variables (see discussion below). In case v_c is a `t`-variable a more complex behavior depending on the current recursion applies: (i) Within the `then`-branch of the recursion the `else`-child of the current node is the terminal *0-node* and the `then`-child is the current node itself. (ii) Within the `else`-branch of the recursion one is enabled to skip one recursive call, since the node to be allocated here would be eliminated due to the *0-sup.*-reduction rule anyway.

The whole functionality of the new operator is implemented by algorithm `Execute-Activity` as illustrated in Fig. 3. This algorithm takes the root nodes of the symbolic structure Z_{unex} and Z_l as arguments (here g and f), their sets of function variables (here $\mathcal{G}$ and $\mathcal{F}$) and the set of all function variables $\mathcal{V}$, –the Boolean parameter *skipFlag* is irrelevant for the time being. In line 11 - 51 the different recursions are prepared by setting the parameters of future function calls accordingly, where the recursion is actually executed in line 55 and 56. Once the algorithm returns from a recursion one allocates a node or abstracts from the current variable v_c (line 57 -60, as discussed above). One may also note that the caching of pre-computed results is intricate, since in case of non-function variables solely results for `s`-variables can be stored or fetched, otherwise one terminates with wrong results (line 61 together with line 6). Another important feature of the algorithm is the skipping of variables of $\mathcal{V}$, if on the current path no node labelled with the current variable appears. Such variables can be ignored as long as the last visited variable was $\notin \vec{\mathsf{s}}$, was not a non-function variable for Z_l ($\mathsf{v}_c \notin \mathcal{F}$) and did not refer to the last node visited in Z_{unex}. Otherwise the recursion must stop at the resp. variable, so that a node can be allocate on the respective path. The status bit for handling such cases is set in line 30, where the skipping of the variables is implemented in line 33-38. For computing now the set of all reachable states algorithm `ExecuteActivity` must be repetitively executed until a fixed point is reached. This can be done by replacing the three steps of multiplication, abstraction and relabelling (see code fragment above) as found in symbolic reachability algorithms by our new algorithm, where the commonly executed insertion of identity structures is also obsolete.

Finally it is also worth noting that in order to implement algorithm `Execute-Activity` for BDDs and standard Multi-terminal Binary Decision Diagrams [1] one need to adapt the terminal conditions, the conditions for cache look-ups and insertion, and assign the current node g (f) resp., to the `then`-child $g1$ ($f1$) instead of the *0-node* (line 22,31,42,45). Also it must be taken care of the node allocating function, so that it implements the correct node elimination rule (line 58).

For exemplification one may refer to Fig. 4, where contrary to previous illustrations we depicted now true ZDDs. Z_{unex} represents state $(1,0,0,1,0)$ and $(0,1,0,1,0)$ of the SPN of Fig. 1. Let Z_l encode now the transitions as induced by submodel S consisting of activity c and d and their pre- and post-sets of SVs. This gives that

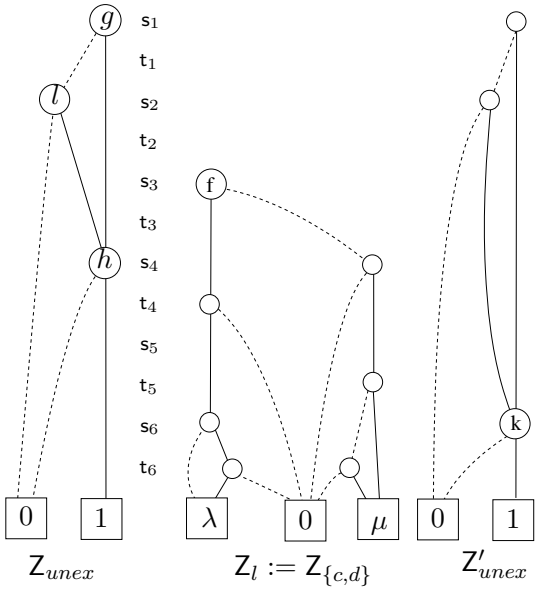


Figure 4. Source state, submodel-local transition function and one-step reachability set

N	$\#states$	$\#trans.$	$\lvert\mathcal{V}\rvert$	Standard scheme		Partitioned RA		1-step RA	
				t sec.	# nod.	t sec.	# nod.	t sec.	# nod.
Fault-tolerant Multi-processor System (FTMP)									
6	$9.9082E+15$	$1.7463E+17$	390	828.42	2,181,962	8.32	2,986,912	5.96	1,919,094
8	$1.7189E+21$	$4.0150E+22$	520	50604.99	5,203,296	22.75	7,594,248	16.34	4,821,764
Courier Protocol (CP)									
10	$2.4967E+9$	$1.7673E+10$	166	290.53	9,753,753	26.95	7,539,173	23.84	5,584,394
15	$4.5538E+10$	$3.4397E+11$	166	3937.39	72,332,345	391.16	45,941,998	677.99	42,531,553
Kanban Manufacturing System (Kanban)									
10	$1.0059E+9$	$1.2032E+10$	128	73.60	6,480,273	11.51	5,669,753	10.56	5,210,304
12	$5.5199E+9$	$6.8884E+10$	128	311.35	16,289,378	29.97	14,189,631	27.30	13,383,554
Flexibe Manufacturing System (FMS)									
15	$7.2428E+8$	$7.3780E+9$	124	193.62	1,809,266	6.93	2,100,353	5.09	1,333,056
20	$8.8313E+9$	$9.4968E+10$	150	1057.28	4,293,513	26.50	4,682,673	18.60	3,042,104
Cyclic Server System (Polling)									
21	$6.6060E7$	$7.4868E8$	168	0.18	72,040	0.10	63,769	0.10	51,220
25	$1.2583E9$	$16.7772E9$	200	0.17	101,178	0.21	89,723	0.13	72,016

Table 1
Run-times and peak memory consumptions of BDD-based symbolic reachability analysis

the variables $\{\mathtt{s}_1, \ldots, \mathtt{t}_2\}$ are non-function variables for Z_l, whereas $\{\mathtt{s}_3, \ldots, \mathtt{t}_6\}$ are its input variables. Within the first recursive call the `else`-branch takes node l and f as argument, and the `then`-branch takes node h and f as argument. As one can see, it is often also not necessary to stop for each variable in $\mathcal{V}$, since when returning from the recursion sometimes no node will be allocated at the respective level, e.g. between node k of ZDD Z'_{unex} and the nodes at level $\mathtt{s}_2$ and $\mathtt{s}_1$ no nodes are allocated, which gives way for optimization of the algorithm as illustrated above.

4 Emperical Evaluation

In previous work the Möbius modeling tool was extended with a ZDD-based symbolic engine [11]. As standard for semi-symbolic methods this engine also executes explicit SG exploration and encoding, but most likely for a limited number of transitions only. The large majority of transitions is generated by executing a symbolic composition scheme. which requires also the execution of a symbolic reachability scheme for identifying non-reachable transitions within the symbolic structures. As for pure symbolic schemes, the major CPU time - and memory consumption is therefore here also imposed by the symbolic reachability analysis. Consequently this framework is highly suited for benchmarking the new algorithm. Table 1 shows the different run-times when analyzing standard benchmarking models from the literature, where we employed a standard bfs symbolic reachability analysis scheme (stand. scheme), the partitioned quasi-dfs. scheme (partitioned RA) as introduced in [11] and our new algorithm (1-step RA). We give the model scaling parameter N, the columns of $\#states$ and $\#trans.$ report the size of the model's underlying CTMCs and $\mathcal{V}$ refers to the number of Boolean variables employed for encoding each system transition. For benchmarking the new algorithm the run-times of the schemes are given in sec. (t), as well as the peak memory consumption, where we recorded the max. number of nodes allocated $(\#nod.)$. As indicated by the data of Table 1, the new algorithm improves the run-time and lowers the peak memory consumption in almost all cases, –up to this end we can not really explain the behavior of the CP model for $N = 15$.– The data makes clear that the partitioning of the transition relations has the largest impact on the run-time and memory reductions. But nevertheless, combining the different steps of symbolic reachability analysis within a single BDD-operator and making the explicit insertion of identity structures obsolete improves the situation further, where the new algorithm in particular works well for models with very large state descriptors as demonstrated by the FTMP model. Finally one may note that in case we do not construct the transition rate matrix of the overall model, which is required for solving the model's underlying CTMC, peak memory consumption could even be reduced further up to a factor of 0.5 in case of the FTMP model. However, this seems to be insignificant for the run-time of the scheme, since this clearly stems from reachability analysis and not from computing the overall transition rate matrix which can be constructed by evaluating $\sum_{l \in \mathcal{A}ct} \mathsf{Z}_l \times \mathbb{1}(\mathcal{V}_l^{\mathsf{I}}) \times \mathsf{Z}_{reach}$.

5 Conclusion

In this paper we presented a new scheme for carrying out symbolic reachability analysis. The newly introduced algorithm computes the one-step reachability set for a set of states and with respect to a (local) transition function within a single BDD-operation. Contrary to existing approaches the here presented algorithm makes symbolic composition unnecessary, which is achieved by defining an identity semantics within the (local) transition functions on the positions of non-function variables. As demonstrated by the collected run-time data the new algorithm may

reduce the run-time and memory requirement of contemporary BDD-based schemes. The approach can be easily adapted to the case of standard BDDs and their multi-terminal derivatives. It could find therefore its application in contemporary quantitative symbolic model checkers such as Caspa [8] or Prism [14]. However, the success of tools such as Prism [14], Caspa [8] (both based on MTBDDs) and Möbius with its ZDD-based engine, is largely due to the efficiency of the employed symbolic data structures.In the context of high-level model descriptions, a model's state commonly consists of many state counters, each referring to the state of a local process, to the current value of a specific process parameter, to the number of tokens in a specific place of a Petri net, etc.. When making use of BDD-based structures in such a setting, each state counter is encoded in binary form by n bits, leading to a large number of bit positions filled with zeroes and to a possible small number of encodings of reachable states with respect to all possible 2^n state labelling. In such a setting MTBDDs and especially ZDDs have shown to be very helpful, as long as their space complexity is restricted. In such an area the here proposed algorithm seems to be a useful innovation. However if other BDD-based approaches it will also fail in cases of highly populated DDs, due to its recursive nature and due to the finiteness of operator caches.

References

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

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

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

[4] 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.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. The MIT Press, Cambridge, MA (USA), 1999.

[6] 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 S. Graf and M. Schwartzbach, editors, *Proc. of the 6'th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00), Berlin (Germany)*, LNCS 1785, pages 395–410, Berlin, 2000. Springer.

[7] 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.

[8] 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.

[9] K. Lampka. *A symbolic approach to the state graph based analysis of high-level Markov reward models*. PhD thesis, University of Erlangen-Nuremberg, Erlangen (Germany), 2007.

[10] 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.

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

[12] K. Lampka, M. Siegle, J. Ossowskis, and C. Baier. Partially-shared zero-suppressed Multi-Terminal BDDs: Concept, Algorithms and Applications, 2008. Article submitted for publication, a preliminary version can be downloaded as technical report from ftp.tik.ee.ethz.ch/pub/publications/TIK-Report-289.pdf.

[13] O. Lhoták, S. Curial, and J.N. Amaral. Using ZBDDs in Points-to Analysis. In *Proc. of the 20th International Workshop on Languages and Compilers for Parallel Computing*, October 2007.

[14] PRISM. http://www.cs.bham.ac.uk/∼dxp/prism/.

[15] Oriol Roig, Jordi Cortadella, and Enric Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri nets. In *16th International Conference on the Application and Theory of Petri Nets*, volume 815, pages 374–391, 1995.

[16] W.H. Sanders. *Construction and solution of performability models based on stochastic activity networks*. PhD thesis, University of Michigan, 1988.

[17] C.S. Shannon. *Eine symbolische Analyse von Relaisschaltkreisen*. Verlag Brinkmann + Bose, 2000. The article originally appeared with the title: *A Symbolic Analysis of Switching Circuits* in Transactions AIEE, 57 (1938), 713.

[18] M. Siegle. Advances in model representation. In Luca de Alfaro and Stephen Gilmore, editors, *Process Algebra and Probabilistic Methods*, LNCS 2165, pages 1–22. Springer, September 2001. Proc. of the Joint Int. Workshop, PAPM-PROBMIV 2001, Aachen (Germany).

[19] F. Somenzi. Binary decision diagrams. *Calculational System Design*, 173:303–366, 1999.