Technical report: TIK-Report-289

Partially-shared Zero-Suppressed Multi-Terminal BDDs: Concept, Algorithms and Applications

Kai Lampka · Markus Siegle
Jörn Ossowski · Christel Baier

Abstract Multi-Terminal Binary Decision Diagrams (MTBDDs) are a well accepted technique for the state graph (SG) based quantitative analysis of large and complex systems specified by means of high-level model description techniques. However, this type of Decision Diagram (DD) is not always the best choice, since finite functions with small satisfaction sets, and where the fulfilling assignments possess many 0-assigned positions, may yield relatively large MTBDD based representations. Therefore, this article introduces *zero-suppressed* MTBDDs and proves that they are canonical representations of multi-valued functions on finite (input) sets. For manipulating DDs of this new type, possibly defined over different sets of function variables, the concept of *partially-shared zero-suppressed* MTBDDs and respective algorithms are developed. The efficiency of this new approach is demonstrated by comparing it to the well-known standard type of MTBDDs, where both types of DDs have been implemented by us within the C++-based DD-package Jinc. The benchmarking takes place in the context of Markovian analysis and probabilistic model checking of systems. In total, the presented work extends existing approaches, since it not only allows one to directly employ (multi-terminal) zero-suppressed DDs in the field of quantitative verification, but also clearly demonstrates their efficiency.

Keywords Binary Decision Diagrams and their algorithms, quantitative verification of systems, symbolic data structures for performance analysis

Kai Lampka
ETH Zuerich, Computer Engineering and Communication Networks Lab.
E-mail: lampka@tik.ee.ethz.ch

Jörn Ossowski
University Dresden, Inst. for Theoretical Comp. Sc.
E-mail: mail@jossowski.de

Markus Siegle
Univ. of the German Federal Armed Forces Munich, Inst. for Comp. Eng.
E-mail: markus.siegle@unibw.de

Christel Baier
Technical University Dresden, Inst. for Theoretical Comp. Sc.
E-mail: baier@tcs.inf.tu-dresden.de

1 Introduction

1.1 Motivation

Decision diagrams (DDs) are directed acyclic graphs for representing finite functions. Multi-terminal Binary Decision Diagrams (MTBDDs) are among the most efficient techniques for the state graph (SG) based quantitative analysis of large and complex systems, commonly described by Markovian extensions of well known description techniques, e.g. Stochastic Process Algebra [7] or Generalized Stochastic Petri Nets [3], among many others. The success of modelling and evaluation tools such as the probabilistic symbolic model checker Prism [19], the stochastic process algebra tool Caspa [10] (both based on MTBDDs) or the modelling and analysis tool Smart [24] (based on multi-valued DDs) is largely due to the efficiency of the employed symbolic data structures. In the context of such 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 MTBDDs 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 labellings. In such a setting, MTBDDs are not the best choice, since finite functions with small satisfaction sets, and where the fulfilling assignments possess many 0-assigned positions, may yield relatively large MTBDD based representations. The *0-suppressing* (*0-sup.*) reduction rule as introduced in [14] has the potential to improve such situations, since contrary to MTBDDs one does not allocate nodes for 0-assigned bit-positions. Thus, this reduction rule helps to reduce memory space and thus computation time when generating and manipulating symbolic representations of SGs underlying high-level model specifications, thereby ultimately enabling the analysis of larger systems. However, there is a significant problem attached to the usage of Minato's *0-suppressing* reduction rule.

When considering *0-sup.* BDDs defined on different sets of variables, the sharing and manipulation of their graphs turns out to be more complex than in case of standard BDDs or *0-sup.* BDDs with a global set of function variables. The Shanon expansion [22] requires that for deducing the function represented by a node of a *0-sup.* BDD, the set of function variables must be known since skipped function variables are assumed to be *0-sup.*, whereas non-function variables are assumed to have a *don't-care* semantics. Consequently, in the presence of multi-rooted DDs [21], as provided by standard implementations such as Cudd [25] or the recently developed package Jinc [8, 18], the nodes of the *0-sup.* DDs lose their uniqueness as soon as the represented functions are defined on different sets of variables. *This is why in a wide range of applications standard implementations of* 0-sup*. DDs can not be employed directly.* As an example, one may think of state graph based symbolic quantitative verification of systems, which is an important part of our research. Since the *0-sup.* DD representing the (stochastic labelled) transition relation and the *0-sup.* DD representing the set of states have not all variables in common, –a function for the transition relation over **s** and **t** variables and a function for the source and target states (or, for example, a vector of rewards associated with the states), each defined either over the variables of **s** or **t**,– standard implementations of *0-sup.* DDs and their algorithms fail to even execute a symbolic reachability analysis, if not somehow adapted. To solve this problem, this work de-

velops the concept of partially shared *0-sup.* DDs and presents generic algorithms for efficiently manipulating DDs of that kind.

1.2 Contributions

For the quantitative analysis of systems, this paper extends zero-suppressed Binary Decision Diagrams (ZBDDs) [14] to the multi-terminal case, thereby obtaining zero-suppressed Multi-terminal Binary Decision Diagrams (ZMTBDDs). It is shown that this new type of decision diagram (DD) is a canonical representation for multi-valued functions on finite (input) sets. But when applying Minato's 0-suppressing (*0-sup.*) reduction rule, the sharing of the graphs of the DDs is not trivial anymore. For deducing the function represented by a ZMTBDD's graph correctly, the set of Boolean input - or function variables must be known. Thus within (fully) shared DD-environments, e.g. as provided by Cudd, and in case of different sets of function variables, ZMTBDD-nodes lose their uniqueness, if they are meant to be defined on differing sets of function variables. To solve this problem in an efficient way, this paper develops the concept of *partially shared* ZMTBDDs (pZMTBDDs), and also introduces new algorithms for manipulating them. Most importantly, a new variant of Bryant's well known `Apply`-algorithm [4] will be discussed. The newly obtained *p*`ZApply`-algorithm not only allows one to implement pZMTBDDs within standard shared DD-environments, such as Cudd or Jinc, but also supports the application of non-zero-preserving operators, i.e. of operators `op` where $0\ \texttt{op}\ 0 \neq 0$ holds (such as *nand* and *nor*).

For evaluating the efficiency of the presented approach, the paper compares pZMTBDDs to the well-known MTBDDs, where as a matter of fairness we have implemented both types of DDs within the same DD packages, namely within the C++-based DD-package Jinc. As demonstrated by various case studies, ZMTBDDs turn out to be superior to MTBDDs in terms of memory - and thus run-time efficiency, when it comes to the stochastic performance evaluation and/or probabilistic model checking of large and complex systems.

1.3 Related work

Throughout the last decade, many derivatives of decision diagrams (DDs) have been developed. For representing stochastic transition relations, the most prominent types are *multi-terminal* Binary Decision Diagrams (MTBDDs) [1,6,23] and (multi-terminal) *Multi-valued Decision Diagrams* (MDDs) [9,5]. All these data types are extensions of Binary Decision Diagrams [13,2] for which Bryant [4] designed algorithms for efficiently manipulating them and keeping them reduced. Variants of these symbolic algorithms have found their way into contemporary DD-packages providing multi-rooted DDs defined on a global set of function variables. To the best of our knowledge, previous works [14,15,26] only considered *0-sup.* BDD for representing (pure) Boolean functions, where all *0-sup.* BDDs had to be defined on the same set of function variables (!) to make their manipulating algorithms work properly.

The concept of partially shared DD as developed in this work is not the only solution to the problem of differing sets of function variables, orthogonal procedures exist.

Since the meaning of nodes in a shared *0-sup.* DD relies on a fixed variable set, one could either work with different shared *0-sup.* DDs for each relevant variable set and transformation algorithms, or work with a single shared pseudo-reduced *0-sup.* DD (where don't care nodes are allocated at the levels of non-function variables and reducedness is understood in a level-wise fashion). We did not implement these approaches, since it must be expected that both alternatives would lead to rather inefficient solutions. For the first alternative, we expect that the transformation algorithms are very time-consuming. For the second alternative, we would lose all advantages of suppressing zeroes, since pseudo-reduced ZDDs agree with pseudo-reduced BDDs. Since the idea of partially shared DDs also applies to Minato's standard *0-sup.* BDDs, this article clearly extends previous works and thus ultimately allows one to replace standard types of reduced ordered DDs by zero-suppressed ones in a wide range of applications.

1.4 Organization

Sec. 2 presents the background material and repeats some usefull concepts with respect to Boolean functions and their representation. Sec. 3 introduces our new type of DD and discusses its canonicity aspects. Sec. 4 introduces the concept of partially shared *0-sup.* DD and presents generic algorithms for efficiently manipulating them. Sec. 5 describes our implementation of pZMTBDDs within Jinc, as well as the benchmarking experiments carried out. Sec. 6 concludes the paper.

2 Background material

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.

Canonical representations: Two Boolean functions are *equivalent*, if and only if their function values coincide for all inputs. A representation of a Boolean function is called *canonical* if each function f has exactly one representation of this type. A representation is denoted as *strongly canonical* if two equivalent functions have the same representation, no matter what their sets of (input or function) variables are. If for identical sets of essential variables and different sets of (input or function) variables the representation of two equivalent function is not the same, we denote them as *weakly canonical*. In this sense, the canonical disjunctive normal form (CDNF) is a weakly canonical representation.

For example, consider the two functions $f_1 := \mathsf{x}_1\mathsf{x}_2 + \mathsf{x}_1(1 - \mathsf{x}_2)$ and $f_2 := \mathsf{x}_1$. Obviously these functions are equivalent. However, since the functions possess different sets of variables, their CDNFs differ.

Co-factors and expansion: Let $f : \mathbb{B}^n \rightarrow \mathbb{B}$ be a Boolean function and let $\mathcal{F}$ be the set of its variables. Function f can be expanded with respect to its variables. If one expands only one variable, e.g. v_i, one ends up with the two co-factors of f with respect to v_i, namely (a) the one-co-factor $f|_{\mathsf{v}_i:=1} := f(\mathsf{v}_1, \ldots, \mathsf{v}_{i-1}, 1, \mathsf{v}_{i+1}, \ldots \mathsf{v}_n)$ or (b) the zero-co-factor $f|_{\mathsf{v}_i:=0} := f(\mathsf{v}_1, \ldots, \mathsf{v}_{i-1}, 0, \mathsf{v}_{i+1}, \ldots, \mathsf{v}_n)$. If the variable to be expanded is clear from the context, we will use the simplified notation f_1 and f_0 for referring to the respective co-factors. For all $\mathsf{v}_i \in \mathcal{F}$ it holds:

$$f := \mathsf{v}_i f(\mathsf{v}_1, \ldots, \mathsf{v}_{i-1}, 1, \mathsf{v}_{i+1}, \ldots, \mathsf{v}_n) + (1 - \mathsf{v}_i) f(\mathsf{v}_1, \ldots, \mathsf{v}_{i-1}, 0, \mathsf{v}_{i+1}, \ldots, \mathsf{v}_n)$$

This so-called *Shannon-expansion*, introduced in 1938 by Shannon in the context of switching functions [22], can be recursively applied until all n variables are made constant. The expansion can be applied for an arbitrary subset $\mathcal{F}' \subseteq \mathcal{F}$, where the notation $f|_{\mathbf{v}':=\mathbf{b}}$ refers to the sub-function derived from function f by assigning the values contained in the Boolean vector $\mathbf{b}$ to the variables in $\mathcal{F}'$.

*Don't care semantics for variables (*dnc*-semantics):* A variable v_k is a *dnc* variable if and only if its one- and zero-co-factors are identical. Let function $g := f|_{\mathbf{v}':=\mathbf{b}}$ and let v_k be a *dnc* variable of g. Applying the Shannon expansion to g with respect to v_k one obtains:

$$\begin{aligned} g &= (1 - \mathsf{v}_k) g_0 + \mathsf{v}_k g_1 \quad \text{since } \mathsf{v}_k \text{ is } \textit{dnc} \text{ we have } g' = g_0 = g_1 \text{ and thus:} \\ g &= ((1 - \mathsf{v}_k) + \mathsf{v}_k) g' \\ &= g' \end{aligned} \tag{1}$$

Variable v_k is therefore a non-essential variable for function g. Thus one does not need to consider variables of such kind when expanding function g, a subfunction of f. As a result it is also clear that within a BDD no nodes for such variables need to be allocated within the graph representing g (c.f. discussion below). At this point it is important to note that nevertheless v_k may still be essential for function f.

*0-suppressing semantics for variables (*0-sup.*-semantics):* A variable v_k is denoted a *0-sup.* if and only if its one-co-factor is the constant zero-function ($f_1 = 0$). Applying the Shannon expansion to a function f and a *0-sup.* variable v_k yields:

$$\begin{aligned} f &= (1 - \mathsf{v})_k f_0 + \mathsf{v}_k f_1 \qquad \text{where } f_1 = 0 \text{ since } \mathsf{v}_k \text{ is 0-suppressed} \\ &= (1 - \mathsf{v}_k) f_0 \end{aligned} \tag{2}$$

In contrast to the *dnc*-case, it is obvious that variable v_k cannot be ignored.

Binary Decision Diagrams and derivatives: A Binary Decision Diagram (BDD) is a directed acyclic graph for representing boolean functions [13, 2, 4]. It consists of a set of inner nodes and a set of terminal nodes, where the inner nodes are labelled by boolean variables and terminal nodes carry values from $\{0, 1\}$. Each inner node possesses two children, a 0- and a 1-child, connected to the respective parent node by an incoming 0- or 1-edge. If the variables labelling the inner nodes appear in the same order on every path one speaks of an ordered BDD. An ordered BDD is reduced (in the standard, *dnc*-sense), if no isomorphic subgraphs exist and no *dnc*-nodes exist (where a non-terminal node of a BDD is a *dnc*-node if its 1- and 0-edge point to the same child). Since the associated variable of such a node is a *dnc*-variables and thus not essential for

the respective function f, *dnc*-nodes can safely be omitted (*dnc* reduction) [2,4]. Reduced ordered BDDs (ROBDDs) are known to be a strongly canonical representation of Boolean functions. Within a single BDD-environment, each allocated BDD-node represents therefore a unique Boolean function [21].

A node referring to a *0-sup.*-variable is denoted *0-sup.* node, its outgoing 1-edge points to the terminal *0-node*. Reducing ordered (isomorphism-free) BDDs by eliminating *0-sup.*-nodes rather than *dnc*-nodes leads to ZBDDs [14]. A ZBDD is a weakly canonical representation of a Boolean function [21].

3 Data Structure

We consider n-ary pseudo-Boolean functions, i.e. functions of the type $f : \mathbb{B}^n \mapsto \mathbb{D}$. When shifting the co-domain of standard reduced ordered BDDs from $\mathbb{B}$ to $\mathbb{D}$ one obtains MTBDDs. Analogously, by extending reduced ordinary ZBDDs we obtain ZMTBDDs.

Definition 1 An ordered ZMTBDD is a tuple $\mathsf{A} = (\mathcal{K}_{NT}, \mathcal{K}_T, \mathcal{F}, \mathtt{var}, \mathtt{then}, \mathtt{else}, \mathtt{value}, \mathtt{root})$ where

(1) $\mathcal{K}_{NT}$ is the set of non-terminal (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) $\mathcal{F} = \{\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n\} \subseteq \mathcal{V}$ is a finite (possibly empty) set of Boolean variables. $\mathsf{t} \notin \mathcal{V}$ is a pseudo-variable, labelling the terminal nodes and solely used for technical reasons. π is a strict total ordering on the elements of $\mathcal{F} \cup \{\mathsf{t}\}$ such that $\forall \mathsf{x}_i \in \mathcal{F} : \mathsf{x}_i <_\pi \mathsf{t}$.
(3) $\mathtt{var} : \mathcal{K}_{NT} \cup \mathcal{K}_T \mapsto \mathcal{F} \cup \{\mathsf{t}\}$ such that $\forall k \in \mathcal{K}_{NT} \cup \mathcal{K}_T : \mathtt{var}(k) = \mathsf{t} \leftrightarrow k \in \mathcal{K}_T$.
(4) $\mathtt{then} : \mathcal{K}_{NT} \mapsto \mathcal{K}_{NT} \cup \mathcal{K}_T$ such that $\forall n \in \mathcal{K}_{NT} : \mathtt{var}(n) <_\pi \mathtt{var}(\mathtt{then}(n))$.
(5) $\mathtt{else} : \mathcal{K}_{NT} \mapsto \mathcal{K}_{NT} \cup \mathcal{K}_T$ such that $\forall n \in \mathcal{K}_{NT} : \mathtt{var}(n) <_\pi \mathtt{var}(\mathtt{else}(n))$.
(6) $\mathtt{value} : \mathcal{K}_T \mapsto \mathbb{D}$, where $\mathbb{D} \subset \mathbb{R}$.
(7) $\mathtt{root} \in \mathcal{K}_{NT} \cup \mathcal{K}_T$.

A ZMTBDD is called reduced if the following conditions apply:

(1) (Ismorphism 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) (0-suppressing rule) There is no inner node whose `then`-successor is the terminal *0-node*; i.e. $\nexists n \in \mathcal{K}_{NT} : \mathtt{then}(n) \in \mathcal{K}_T \wedge \mathtt{value}(\mathtt{then}(n)) = 0$.

Let $k, l \in \mathcal{K}_{NT} \cup \mathcal{K}_T$ and let $\mathcal{F} \subseteq \mathcal{V}$. We now introduce a notation for the set of Boolean variables which are from $\mathcal{F}$ but have a smaller or greater order than $\mathtt{var}(k)$:

$$\mathcal{F}_k^{\text{before}} := \{\mathsf{x}_i \in \mathcal{F} | \mathsf{x}_i <_\pi \mathtt{var}(k)\}$$
$$\mathcal{F}_k^{\text{after}} := \{\mathsf{x}_i \in \mathcal{F} | \mathsf{x}_i \ _\pi> \mathtt{var}(k)\}$$

With the help of this notation, the semantics of a ZMTBDD node *with respect to a set of Boolean variables* can be defined as follows:

Definition 2 The pseudo-Boolean function $f(n, \mathcal{F})$ represented by ZMTBDD-node $n \in \mathcal{K}_{NT} \cup \mathcal{K}_T$ and variable set $\mathcal{F} \subseteq \mathcal{V}$ is recursively defined as follows:
If $n \in \mathcal{K}_T$ then

$$f(n, \mathcal{F}) := \left(\prod_{\mathsf{x}_i \in \mathcal{F}} (1 - \mathsf{x}_i) \right) * \mathtt{value}(n)$$

Else, if $n \in \mathcal{K}_{NT}$ and $\mathtt{var}(n) \notin \mathcal{F}$, then $f(n, \mathcal{F})$ is undefined.
Else (if $n \in \mathcal{K}_{NT}$ and $\mathtt{var}(n) \in \mathcal{F}$)

$$f(n, \mathcal{F}) := \prod_{\mathsf{x}_i \in \mathcal{F}_n^{\text{before}}} (1 - \mathsf{x}_i) \\ * \left[\mathtt{var}(n) * f(\mathtt{then}(n), \mathcal{F}_n^{\text{after}}) + (1 - \mathtt{var}(n)) * f(\mathtt{else}(n), \mathcal{F}_n^{\text{after}}) \right]$$

The last defining equation of Def. 2 is a combination of the Shannon expansion for Boolean functions [22] and the application of the zero-suppressing rule. According to Def. 2, the Boolean function represented by the combination of a ZMTBDD node and a set of Boolean variables is uniquely determined and finally gives that ZMTBDDs are canonical representations of pseudo Boolean functions, which is shown now.

Weak canonicity of ZMTBDDs is based on the following two theorems and their proofs.

Theorem 1 (Existence) *Let $\mathcal{F} = \{\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n\} \subseteq \mathcal{V}$ be a set of boolean variables. For each pseudo-Boolean function f defined on $\mathcal{F}$ and given strict total ordering π on $\mathcal{V}$ there exists a ZMTBDD-based representation.*

Theorem 2 (Canonicity) *Let $\mathcal{F} = \{\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n\} \subseteq \mathcal{V}$ be a set of boolean variables with total strict ordering π on $\mathcal{V}$. Let B and C be reduced ZMTBDDs defined on their variable sets $\mathcal{B} = \mathcal{C} = \mathcal{F}$, representing the functions f_{B} and f_{C}, respectively. If $f_{\mathsf{B}} = f_{\mathsf{C}}$, then ZMTBDDs B and C are isomorphic.*

Proof of Th. 1 (existence): For simplicity, the proof is for the Boolean case only. Its extension to the pseudo-Boolean case is straight-forward. The proof is by induction on the number of variables $n = |\mathcal{F}|$. Without loss of generality we assume the ordering $\mathsf{x}_1 <_\pi \mathsf{x}_2 <_\pi \ldots <_\pi \mathsf{x}_n <_\pi \mathsf{t}$.

Base case: For the case $n = 0$ (i.e. $\mathcal{F} = \emptyset$), assume that f is represented by a non-terminal. This leads to a contradiction, since the function `var` for that non-terminal would be undefined. Thus, since f is a constant, it can only be represented by the terminal carrying the value of f.

Induction step: Assume that the conjecture holds for all $(n-1)$-ary Boolean functions defined on $\{\mathsf{x}_2, \ldots, \mathsf{x}_n\}$. We show that then the conjecture also holds for the n-ary Boolean function f defined on $\{\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n\}$. We expand f as

$$f(\mathsf{x}_1, \ldots, \mathsf{x}_n) = \mathsf{x}_1 \cdot f_1(\mathsf{x}_2, \ldots, \mathsf{x}_n) + (1 - \mathsf{x}_1) \cdot f_0(\mathsf{x}_2, \ldots, \mathsf{x}_n)$$

where $f_1(\mathsf{x}_2, \ldots, \mathsf{x}_n) = f(1, \mathsf{x}_2, \ldots, \mathsf{x}_n)$ and $f_0(\mathsf{x}_2, \ldots, \mathsf{x}_n) = f(0, \mathsf{x}_2, \ldots, \mathsf{x}_n)$. According to the induction hypothesis, both f_1 and f_0 have ZBDD representations.

Case 1: If $f_1 = f_0 \neq 0$ then f_1 and f_0 are both represented by $(k, \{\mathsf{x}_2, \ldots, \mathsf{x}_n\})$, where $k \in \mathcal{K}_{NT} \cup \{e\}$, $e \in \mathcal{K}_T$ and $\mathtt{value}(e) = 1$ (i.e. k can be the terminal one-node). In this case f can be represented by $(m, \{\mathsf{x}_1, \ldots, \mathsf{x}_n\})$ as shown in Fig. 1(a).

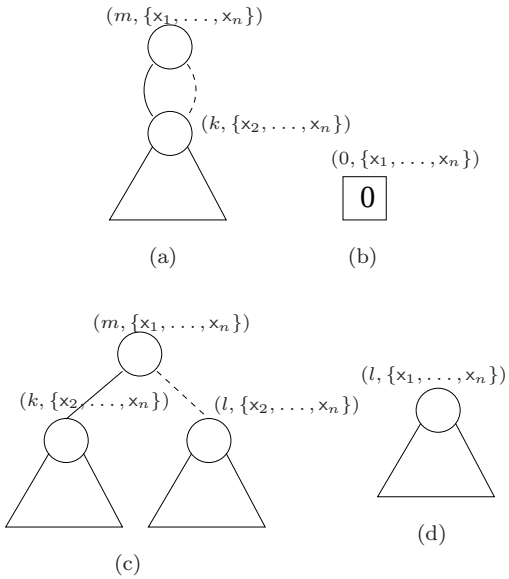

Figure 1 Constructing ZDDs

Case 2: If $f_1 = f_0 = 0$ then f_1 and f_0 are both represented by $(k, \{\mathsf{x}_2, \ldots, \mathsf{x}_n\})$, where $k \in \mathcal{K}_T$ and $\mathtt{value}(k) = 0$ (the terminal zero-node). In this case f is also represented by the zero-node, i.e. by $(0, \{\mathsf{x}_1, \ldots, \mathsf{x}_n\})$ as shown in Fig. 1(b).

Case 3: If $f_1 \neq f_0$ and $f_1 \neq 0$ then x_1 is essential for f. If f_1 is represented by $(k, \{\mathsf{x}_2, \ldots, \mathsf{x}_n\})$ and f_0 is represented by $(l, \{\mathsf{x}_2, \ldots, \mathsf{x}_n\})$, then f can be represented by $(m, \{\mathsf{x}_1, \ldots, \mathsf{x}_n\})$ as shown in Fig. 1(c).

Case 4: If $f_1 \neq f_0$ and $f_1 = 0$ then x_1 is essential for f, but due to the zero-suppressing rule f is represented by the same node as f_0. If f_0 is represented by $(l, \{\mathsf{x}_2, \ldots, \mathsf{x}_n\})$, then f can only be represented by $(l, \{\mathsf{x}_1, \ldots, \mathsf{x}_n\})$ as shown in Fig. 1(d).

Proof of Th. 2 (canonicity): Consider the following two Boolean functions: $f_1(\mathsf{x}_1, \mathsf{x}_2) = \mathsf{x}_1\mathsf{x}_2 + \mathsf{x}_1(1 - \mathsf{x}_2)$ and $f_2(\mathsf{x}_1) = \mathsf{x}_1$, which are equivalent. We observe that the corresponding ZBDDs are not the same, as shown in Fig. 2 (a), since their variable sets are not identical. Therefore ZBDDs are a weakly canonical representation of Boolean functions. The proof is once again by induction on the number of variables $n = |\mathcal{F}|$. Without loss of generality we assume the ordering $\mathsf{x}_1 <_\pi \mathsf{x}_2 <_\pi \ldots <_\pi \mathsf{x}_m <_\pi \mathsf{t}$ with $m \geq n$.

Base case: For the case $n = 0$ (i.e. $\mathcal{F} = \emptyset$) $f_\mathsf{B} = f_\mathsf{C}$ is constant. Let c be the value of $f_\mathsf{B} = f_\mathsf{C}$. Then, the root of B as well as C is a terminal node labelled with c.

Induction step: We will now assume that the root nodes of B and C are non-terminal nodes, each labelled with variable x_k, where x_k is the first not zero assigned variable in $f_\mathsf{B} = f_\mathsf{C}$ according to the fixed ordering π. Let v be the root of B and w be the root of C. Let $v_0 = \mathtt{else}(v)$, $v_1 = \mathtt{then}(v)$, $w_0 = \mathtt{else}(w)$ and $w_1 = \mathtt{then}(w)$ (see Fig. 2 (b)). For $\xi \in \{0, 1\}$, as B and C are reduced, we get $f_{v_\xi} = f_{w_\xi}$ (with variable set $\{x_{k+1}, \ldots, x_n\}$). By induction step the sub-ZBDD with root nodes v_ξ, w_ξ and variable set $\{x_{k+1}, \ldots, x_n\}$ are isomorphic. Hence, B and C are isomorphic.

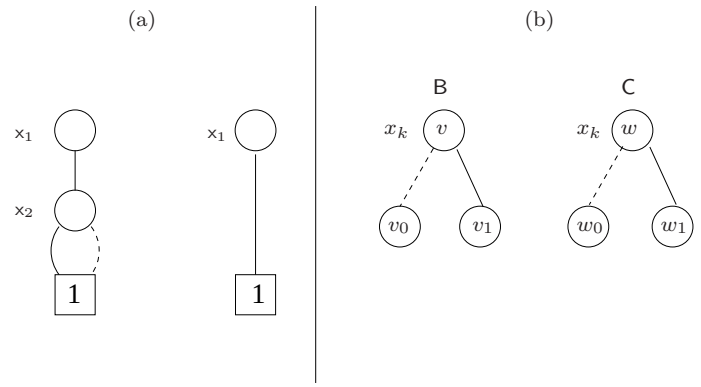

Figure 2 (a) Weak canonical representation (b) Illustrating the proof

In a nutshell, function variables skipped on a path within the ZDD leading to a terminal non-zero node are interpreted as being 0-assigned. Contrary to this, non-function variables are interpreted as *don't* care variables as commonly eliminated within standard reduced ordered BDDs [2,4]. In the following, only reduced ordered types of DDs are considered. Since we have defined $\mathbb{D} \subset \mathbb{R}$, we can combine the values of terminal nodes by arithmetic operators, but it should be emphasized that the concepts described in this paper carry over to more general domains. A ZBDD is a special case of a ZMTBDD, namely $\mathbb{D} = \mathbb{B}$, consequently the concept of *partially-shared* DDs and the algorithms introduced next also applies to them. For making the discussion as generic as possible we will therefore from now on speak of *0-sup.* DDs (ZDDs), addressing the multi-terminal as well as the Boolean type. A concrete case distinction is only made where necessary.

4 Partially shared ZDDs: Concept and Algorithms

Contemporary DD-packages provide strong canonical representations of (pseudo-) Boolean functions. Since nodes can then even be shared among different symbolically represented functions, these packages provide fully shared DD-environments, also known as multi-rooted DDs [21]. This concept is a major source of the efficiency when it comes to the manipulation of DDs, since memory requirements are reduced and the likelihood of finding a pre-computed result in the respective operator-cache is also increased. However, due to weak canonicity, in the presence of multi-rooted DDs, ZDD-nodes lose their uniqueness as soon as the represented functions are defined on different sets of variables. Up to now, this has truly limited the application of *0-sup.* DDs. As an example, one may think of state graph based symbolic quantitative verification of systems where, as pointed out in the beginning, standard ZDD implementations and their manipulating algorithms fail to even execute a symbolic reachability analysis, if not somehow adapted. To solve this problem, this work introduces now the concept of partially shared ZDDs (pZDDs) and presents generic algorithms for efficiently manipulating DDs of that kind. This allows to implement and manipulate *0-sup.* DDs within (standard) fully shared DD-environments, even though the functions to be represented may not have the same set of input variables.

4.1 Concept of partially shared ZDDs (pZDDs)

When working with pZDDs, i.e. with ZDDs having different sets of input variables, each node must be associated with a set of variables such that one can correctly deduce the represented function from the graph rooted in that node. Thus, two ZDD-nodes represent the same function, if not only their subgraphs are isomorphic but also their sets of variables are identical. Therefore, the notion of equality of pZDD nodes must be refined. ¿From now on, two nodes are considered as representing the same function *if and only if their sub-graphs are identical, as well as their sets of function variables!* This gives the following rules concerning node equality (where $\mathcal{N}, \mathcal{M} \subseteq \mathcal{V}$ are the variable sets associated with nodes n and m):

Definition 3 (1) Non-terminal case ($n, m \in \mathcal{K}_{NT}$):

$$n \equiv m \Leftrightarrow$$
$$\mathtt{var}(n) = \mathtt{var}(m), \mathtt{else}(n) = \mathtt{else}(m), \mathtt{then}(n) = \mathtt{then}(m) \text{ and } \mathcal{N} = \mathcal{M}$$

(2) Terminal case ($n, m \in \mathcal{K}_T$):

$$n \equiv m \Leftrightarrow \mathtt{value}(n) = \mathtt{value}(m) \text{ and } \mathcal{N} = \mathcal{M}$$

Each time an algorithm tests for node equality, e.g. for deciding whether the recursion can be terminated or when looking up pre-computed results in the operator's cache, the above rules are applicable. *Instead of actually storing a set of variables for each node (or at least a reference to such sets), we do this only for each pZDD object.* As a consequence, within a shared BDD-environment a pZDD object is now uniquely defined by its root node plus its set of (function) variables. When applying now operators on pZDDs, one not only recurses on the operand DDs, but also iterates over their sets of variables. *At any time a node is accessed, it can be therefore associated with a unique set of variables.* This strategy has the main advantage that it leads to memory and computation time savings, since a single graph represents now different functions and the sharing of graphs can be significantly increased.

For exemplification we refer to Fig. 3. Let the set of all variables defined in the shared DD-environment be denoted $\mathcal{V}^G := \{\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3\}$. If the graphs of Fig. 3 (a) and (b) are both interpreted as standard shared ZDDs, i.e. $\mathcal{V}^{\mathsf{Z}_1} = \mathcal{V}^G$, they represent different functions, namely the Boolean function $f_{\mathsf{Z}_1} := (1 - \mathsf{v}_1) + \mathsf{v}_1\mathsf{v}_3$ in case of Fig. 3 (a) and $f_{\mathsf{Z}_1} := (1 - \mathsf{v}_1)(1 - \mathsf{v}_2) + \mathsf{v}_1(1 - \mathsf{v}_2)\mathsf{v}_3$ in case of Fig. 3 (b). However, if the variables v_1 and v_3 are the only function variables for the function represented by node k_1, the graphs of Fig. 3 (a) and (b) are interpreted as the same function. In contrast, the ZDDs of Fig. 3 (b) and (c) have isomorphic graphs but are intended to represent different functions. By defining sets of variables for each node, where $\mathcal{V}^{n_1} = \mathcal{V}^G$, and $\mathcal{V}^{k_1} = \{\mathsf{v}_1, \mathsf{v}_3\}$, and interpreting each (sub-)graph over the respective set, the correct interpretation is achieved. This allows one to store ZDDs with different sets of function variables as multi-rooted graphs, as it is common within shared BDD-environments. This situation is illustrated in Fig. 3 (d) where Z_1 and Z_2 are represented by different roots in the same graph. In contrast to node n_1 and k_1, the nodes n_2 and k_2, as well as n_3 and k_3 can still be merged, since they have isomorphic sub-graphs and identical sets of function variables. As shown by this example, if one equips each node with a set of function variables, only a sharing of sub-graphs representing the same function is achieved. However, if one equips only pZDDs objects with sets of function variables, the sharing is significantly increased, as illustrated in Fig. 3 (e), but as a result nodes

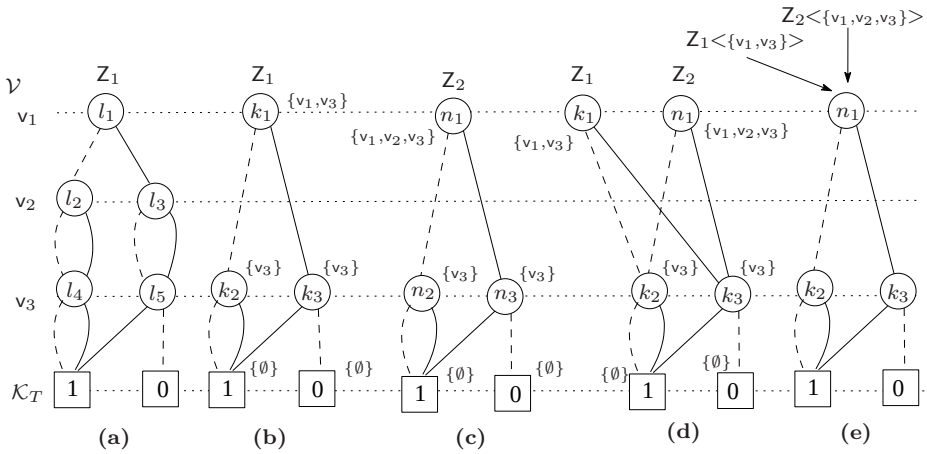

Figure 3 Allocating multi-rooted pZDDs within standard DD-environments

lose their uniqueness. *Therefore, when operating on pZDDs, one must always pass the set of function variables of the operand DDs as additional argument to the manipulating algorithms. While iterating jointly on the graphs and sets of function variables of the operand pZDDs, each node can be associated with a set of function variables.*

4.2 Applying binary operators to pZDDs

A symbolic representation of a function $f := g \,\mathtt{op}\, h$ can be computed by executing the generic *p*`ZApply`-algorithm. This algorithm takes a binary operator `op`, the respective operand DDs (i.e. their root nodes n and m) and their sets of function variables $\mathcal{N}$ and $\mathcal{M}$ as input. For keeping this algorithm as generic as possible, we also make use of so-called operator functions (`op`-functions), which steer the recursive behavior of the *p*`ZApply`-algorithm, such that it does not have to contain the operator-specific case distinctions itself. Contrary to existing work, the `op`-functions must here apply the new rule for node equality as introduced in Def. 3.

4.2.1 Operator function (`op`*-function)*

When working with reduced DDs, one may reach the terminal node of one operand DD earlier, while the partner DD still needs to be traversed further. In some such cases, it is possible to terminate the recursion of the traversing algorithms. The called `op`-function returns either a node, representing the result of $n \,\mathtt{op}\, m$, where n and m are the nodes passed as arguments, or it returns the empty node ϵ. In the latter case the ZDD manipulating algorithm must proceed with the recursion, since the conditions for termination are not satisfied. Since the concept of partially shared ZDDs also applies to partially shared ZBDDs, we first introduce operator functions for Boolean operators, which also turn out to be more complex than their arithmetic counterparts.

`op`-*functions implementing binary operators*

(1) op $= \wedge$: In case of node-equality, the $\wedge$-function may only terminate the recursion if the current sets of function-variables are identical. In case of reaching a terminal and non-terminal node, the situation is as follows: Due to the different semantics of skipped variables – a *dnc-semantics* for the remaining variables in case of reaching the terminal *0-node* and a *0-sup.-semantics* in case of the terminal *1-node* – one may only terminate the recursion, if either one of the nodes is the terminal *0-node*, or one of the input nodes is the terminal *1-node* with its associated set of variables being empty. In case both input nodes are the terminal *1-node*, the recursion can also be terminated, where the terminal *1-node* can be returned as result. Otherwise the recursion must proceed further.

(2) op $= \vee$: In case of node-equality the $\vee$-function behaves like the $\wedge$-function. But when reaching terminal nodes the situation differs. When reaching terminal *1-node*s, the recursion can only be terminated if also the set of variables matches. In case of reaching a terminal *0-node* the $\vee$-function can also terminate, but contrary to the $\wedge$-function not the terminal *0-node* but the other input node is delivered as result.

(3) op $= \setminus$ (difference): The $\setminus$ operator-function steers the *p*ZApply-algorithm in such a way, that the difference of two binary encoded sets is computed. I.e. the $\setminus$ operator function allows us to compute $f := g \wedge \neg h$ with a single (recursive) call to the *p*ZApply-algorithm, rather than first negating function h and than computing the conjunction of g and $\neg h$ as it is necessary in case of BDDs. For computing the complement of a function, one solely needs than to evaluate the expression *1-node*$\setminus h$ by calling *p*ZApply$(\setminus, \textit{1-node}, \mathcal{H}, h, \mathcal{H}, \ldots)$. Concerning the terminal case distinctions one may note that analogously to the above op-functions the $\setminus$ operator-function only terminates the recursion of the *p*ZApply-algorithm in case of node equality, if also the set of function variables matches. Contrary to this terminal *0-node*s terminate the recursion anyway, where either f or the terminal *0-node* are returned as result, depending on the fact whether g or f represented the constant 0-function. –We have found that function ZSetMinus is of great value during the symbolic computation of the set of reachable states of a high-level model.

op-*functions implementing arithmetic operators* The op-functions for op $\in \{*, \div, +, -\}$ can be implemented analogously to the Boolean ones. In case both input nodes are terminal nodes the respective op-function simply needs to return a terminal node labelled with (value(n)opvalue(m)), –as long as the operation is a valid arithmetic operation, otherwise op-function specific conditions apply. In case of the terminal *0-node* and *1-node* one must not necessarily descend to the terminal nodes within both graphs, a termination of the recursion by returning the partner node as result is often possible. E.g. in case of the op-functions implementing $+$ and $-$ one simply returns the partner node as result, as the terminal *0-node* is the neutral element for $+$ and $-$. When computing $\div$ and $*$ for two pZDDs and encountering a terminal *1-node* in one of the graphs, the recursion can also be terminated. Since 1 is the neutral element here, the partner node represents the result, where in case of the division the non-commutativity must be respected, as well as the error case that the divisor equals 0.

4.2.2 The generic pZApply-*algorithm*

The basic idea of the algorithm is that for a given pair of nodes (n, m) and their sets of variables $(\mathcal{N}, \mathcal{M})$, a recursion for each variable $\mathsf{v} \in (\mathcal{N} \cup \mathcal{M})$ is executed. I.e., while descending the operand pZDDs rooted in node n and m, the algorithm has to stop for

Algorithm 4.1 The generic *p*ZApply-algorithm

$p\texttt{ZApply}(op, n, \mathcal{N}, m, \mathcal{M})$
(0) *node* res, e, t;
var $\mathsf{v}_n := min(\mathcal{N}), \mathsf{v}_m := min(\mathcal{M}), \mathsf{v}_c := min(\mathcal{N} \cup \mathcal{M})$;

/* *Check terminal condition* */
(1) $res := \texttt{op}(n, \mathcal{N}, m, \mathcal{M})$;
(2) IF $res \neq \epsilon$ THEN RETURN res;

/* *Check* op*-cache if result is already known* */
(3) $res = \texttt{cacheLookup}(\texttt{op}, n, \mathcal{N}, m, \mathcal{M})$;
(4) IF $res \neq \epsilon$ THEN RETURN res;

/* *Remove variables from sets* */
(5) $\mathcal{N} := \mathcal{N} \setminus \{\mathsf{v}_c\})$;
(6) $\mathcal{M} := \mathcal{M} \setminus \{\mathsf{v}_c\})$;

/* *(A) No level is skipped* */
(7) IF $\texttt{var}(n) = \mathsf{v}_c$ && $\texttt{var}(m) = \mathsf{v}_c$ THEN
(8) $e := p\texttt{ZApply}(op, \texttt{else}(n), \mathcal{N}, \texttt{else}(m), \mathcal{M})$;
(9) $t := p\texttt{ZApply}(op, \texttt{then}(n), \mathcal{N}, \texttt{then}(m), \mathcal{M})$;

/* *(B) Skipped a level only in one of the pZDD* */
(10) ELSE IF $\texttt{var}(n) = \mathsf{v}_c$ THEN
(11) $e := p\texttt{ZApply}(op, \texttt{else}(n), \mathcal{N}, m, \mathcal{M})$;
(12) IF $\mathsf{v}_c = \mathsf{v}_m$ THEN
(13) $t := p\texttt{ZApply}(op, \texttt{then}(n), \mathcal{N}, \textit{0-node}, \mathcal{M})$;
(14) ELSE
(15) $t := p\texttt{ZApply}(op, \texttt{then}(n), \mathcal{N}, m, \mathcal{M})$;
(16) ELSE IF $\texttt{var}(m) = \mathsf{v}_c$ THEN
(17) $e := p\texttt{ZApply}(op, n, \mathcal{N}, \texttt{else}(m), \mathcal{M})$;
(18) IF $\mathsf{v}_c = \mathsf{v}_n$ THEN
(19) $t := p\texttt{ZApply}(op, \textit{0-node}, \mathcal{N}, \texttt{then}(m), \mathcal{M})$
(20) ELSE
(21) $t := p\texttt{ZApply}(op, n, \mathcal{N}, \texttt{then}(m), \mathcal{M})$;

/* *(C) Skipped a level in both pZDDs* */
(22) ELSE
(23) $e := p\texttt{ZApply}(op, n, \mathcal{N}, m, \mathcal{M})$;
(24) IF $\mathsf{v}_n = \mathsf{v}_c$ && $\mathsf{v}_m = \mathsf{v}_c$
(25) $t := p\texttt{ZApply}(op, \textit{0-node}, \mathcal{N}, \textit{0-node}, \mathcal{M})$;
(26) ELSE IF $\mathsf{v}_c = \mathsf{v}_n$
(27) $t := p\texttt{ZApply}(op, \textit{0-node}, \mathcal{N}, m, \mathcal{M})$;
(28) ELSE IF $\mathsf{v}_c = \mathsf{v}_m$
(29) $t := p\texttt{ZApply}(op, n, \mathcal{N}, \textit{0-node}, \mathcal{M})$;

/* *Allocate new node, respecting (ZDD) isomorphism and 0-sup. rule* */
(30) $res := \texttt{getUniqueZMTBDDNode}(\mathsf{v}_c, t, e\})$;

/* *Insert result into* op*-cache and terminate recursion* */
(31) $\texttt{cacheInsert}(\texttt{op}, n, \mathcal{N}, m, \mathcal{M}, res)$;
(32) RETURN res;

each such variable v, in order to trigger the required recursion. The behavior depends hereby on the fact, whether v is *0-sup.*, not essential or an ordinary variable within the current path. The pseudo-code of the generic *p*ZApply-algorithm is given as Algo. 4.1. As input parameters, the algorithm takes the binary operator to be executed, the root nodes (n, m) of the pZDDs to be combined, and their sets of (function) variables ($\mathcal{N}$, $\mathcal{M}$). In lines 1 and 2, the terminal condition is tested with the help of the respective operator function. If this is not successful, one checks the op-function specific computed table (op-cache), if the result is already known from a previous recursion (line 3-4). Note that the sets $\mathcal{N}$ and $\mathcal{M}$ must also be considered, since the sets of variables are not stored within the ZDD-nodes themselves. In case the lookup is not successful, the recursion must be entered:

The pseudo-code of lines 5 and 6 prepares the new sets of variables as required in the next recursion. The pseudo-code of lines 7-9 handles the ordinary branching in case no-skipping of variables appeared within the traversed graphs. The code of lines 10-21 covers the case that the current variable v_c is a skipped variable exclusively in one of the graphs. In such a case one executes at first line 11 or 17, for entering the else-branch of the recursion. Concerning the then-branch, the behavior is more complex. It depends on the circumstances, whether variable v_c is a function-variable of the respective pZDD or it is not. I.e. one either interprets v_c as *0-sup.*. - or as not essential variable within the respective graph. In case v_c is considered as being *0-sup.*, line 13 or alternatively line 19 is executed. In case v_c is considered as being not essential, one assumes a *dnc*-semantics and executes line 15 or alternatively line 21.
Lines 22-29 cover the case that the variable v_c is skipped within both graphs. For the else-branch, the current pair of nodes (n and m) is the pair of children nodes, since the 0-children of the fictitious nodes being skipped are the current node n and m themselves (line 23). Concerning the then-branch the following cases must be covered: (a) the variable is a variable for both graphs: here the standard *0-sup.*-branching rules apply, which means that in both cases the *0-node* is the then-child to be recursed on (line 25); (b) the variable is a non-function variable for one of the graphs and *0-sup.* for the other. Here the branching rules follows a *dnc*-rule in one case and a *0-sup.*-rule in the other case, which means that in the *dnc*-case one does not traverse any further, i.e. one passes the current node into the *then* recursion. Contrary to this, the case of a *0-sup.*-semantics yields a passing of the *0-node* into the *then* recursion (line 27 and 29). Finally when returning from the recursion, one either newly allocates a new ZDD-node representing $f^n \,\mathtt{op}\, f^m$ (line 30), or, in case it already exists, simply re-use the respective node as to be found within the list of allocated nodes with label v_c. This result is then inserted into the op-cache, where also the respective sets of variables must be provided. Now the algorithm can terminate by passing the obtained node as its result to the calling function.

It is not difficult to see that such an extensive treatment of all variables as done within the *p*ZApply-algorithm is sometimes unnecessary. In the following, we therefore identify special cases for which we describe improved variants of the *p*ZApply-algorithm.

4.2.3 Variants of the pZApply*-algorithm*

Fully shared ZDDs and zero-preserving op*-functions:* In case of pZDDs having identical sets of variables and zero-preserving op-functions ($0 \,\mathtt{op}\, 0 = 0$), the *p*ZApply-algorithm

can be simplified. This simplification yields an algorithm whose recursive behavior corresponds to the one of Minato's recursive ZBDD-algorithms [14,15]. The obtained variant will be denoted as *fs*`ZApply`-algorithm in the following – as we recently noticed this variant is also described in [26].
Like Bryant's original `Apply`-algorithm and Minato's original ZBDD-algorithms, this variant only recurses on variables for which a node is actually allocated. In case a variable is skipped in one operand, the *fs*`ZApply` algorithm follows a *0-sup.*-rule, which means when recursing into the `else`-branch the current node is the next node to be traversed, whereas the `then`-branch recursion takes the *0-node* as argument. The *fs*`ZApply`-algorithm can be derived from the *p*`ZApply`-algorithm by omitting line 5-6, 12, 14-15, 18, 20-21 and 22-29, by adapting the Boolean tests of line 7,10 and 16 accordingly, and modifying the function calls of line 1,3 and 30,31, such that the sets of variables are not needed any more. Since the *fs*`ZApply` algorithm solely recurses on variables where nodes are actually allocated within the current path, it can only be applied for operators which are zero-preserving. This stems from the fact that in case of paths leading to the terminal *0-node* skipped variables refer to *dnc*-nodes, which must be considered when replacing the function value 0 with a value $\neq 0$. This is also the reason why the computation of the complement of pZDDs is much more complex than in case of non-*0-sup.* DDs.

Non-shared ZDDs: In case two pZDDs have no variable in common, they can be manipulated by a specialized *p*`ZApply`-algorithm which we denote as *ns*`ZApply`-algorithm. The simplification is mainly based on the fact that certain case distinctions of the generic *p*`ZApply`-algorithm can be omitted. In contrast to the *fs*`ZApply`-algorithm, the *ns*`ZApply`-algorithm still requires to stop for each variable from $\mathcal{N} \cup \mathcal{M}$, no matter if it encounters a node for this variable on the current path or not. This variant is obtained by simply omitting line 7-9, 12-14, and 18-20 of the *p*`ZApply`-algorithm.

The p`ZAnd`*-algorithm:* Again, we consider the computation of $f := n \,\texttt{op}\, m$, where n and m are the root nodes of the operand DDs and $\mathcal{N}, \mathcal{M}$ are the respective sets of variables. The *p*`ZApply` and *ns*`ZApply`-algorithms execute two recursive calls for each variable $\mathsf{v} \in \mathcal{F} := \mathcal{N} \cup \mathcal{M}$. This is less efficient than the original `Apply`-style algorithms [4,14,15,26], since the latter algorithms only need to recurse on those variables for which nodes are actually allocated. In order to achieve the same efficiency, we now present another variant of the *p*`ZApply`-algorithm for the special case $\texttt{op} \in \{\wedge, *\}$, which we denote as *p*`ZAnd`-algorithm.
When skipping a variable, one assumes a *0-sup.*- or a *dnc* semantics, depending on whether the associated variable is a variable or not for the respective pZDD. In case of non- and partially shared ZDDs this led to many case distinctions. If one assumes now that a variable v is skipped in both pZDDs, the following scenarios appear:

(1) The omission results from different semantics, i.e. in case of the pZDD rooted in node n the current variable v is assumed to be *dnc* and in case of the pZDD rooted in node m it is assumed to be *0-sup.* – or the other way round. According to the Shannon-expansion it follows that $n = n_1 = n_0$ and $m_1 \overset{(!)}{=} 0$. This can now be employed for computing $f := n * m$ as follows:

$$\begin{aligned} f &= ((1-\mathsf{v})n_0 + \mathsf{v}\,n_1) * (\mathsf{v}\,m_1 + (1-\mathsf{v})m_0) \quad \text{with } m_1 = 0 \\ &= (1-\mathsf{v})n_0\,m_0 + 0 \quad \text{with } n_0 = n \text{ it follows: } f = (1-\mathsf{v})n\,m_0 \end{aligned}$$

Thus the representation of function f solely depends on the expansion of $(1 - \mathsf{v})\, n\, m_0$, where the pZDD rooted in node m_0 is the current node m itself, according to the *0-sup.*-reduction rule.

(2) The omission results from the same semantics:

(2.a) Under the *dnc*-semantics $\mathsf{v} \notin \mathcal{N} \cup \mathcal{M}$ and therefore nothing needs to be done.

(2.b) When the *0-sup.*-semantics is applicable, also nothing needs to be done, since one has:

$$f = \mathsf{v}(0 * 0) + (1 - \mathsf{v})(m_0 * n_0) = 0 + (1 - \mathsf{v})(m_0 * n_0),$$

which is the semantics of a node to be *0-sup.*. Consequently, one solely needs to traverse the 0-children of the two "fictitious *0-sup.*-nodes" being skipped, which are the current nodes m and n themselves.

The above conclusions allow one to significantly simplify the *p*`ZApply`-algorithm for $\mathtt{op} \in \{\wedge, *\}$, where the resulting *p*`ZAnd`-algorithm only stops for variables where actually nodes are allocated, rather than executing two recursive calls for each variable $\mathsf{v} \in \mathcal{N} \cup \mathcal{M}$. This allows one furthermore to omit the sets of variables, as it was required in case of the generic *p*`ZApply` and `nsZApply`-algorithm. Since the *p*`ZAnd`-algorithm implements the same behavior as the *fs*`ZApply` algorithm, i.e. the algorithm must solely recurse on variables for which actually nodes are allocated, it can be derived from the *fs*`ZApply` algorithm as discussed above.

4.3 The *p*`ZAbstract`-operator

The abstraction from a variable v is implemented by the *p*`ZAbstract`-algorithm, called with a respective `op`-function. The *p*`ZAbstract`-algorithm constructs a representation of the function $h := f|_{\mathsf{v}=0}\, \mathtt{op}\, f|_{\mathsf{v}=1}$, so that variable v is not essential for function h any more. When eliminating nodes labeled with the variable v or in case v is a *0-sup.* variable on the current path, it may occur that previously distinct (sub-) paths collapse. In such cases the respective *p*`ZApply`-algorithm with the binary Boolean operator `op` has to be called for computing $f|_{\mathsf{v}=0}\, \mathtt{op}\, f|_{\mathsf{v}=1}$, where in case v is *0-sup.* $f|_{\mathsf{v}=1} = 0$ holds. Depending on the operator `op` passed as an argument, the *p*`ZAbstract` implements different operations:

(1) In case $\mathtt{op} \in \{\vee, +\}$ it implements the existential quantification, where the algorithm can also be simplified; One only needs to take care of nodes labeled with variables to be abstracted from. I.e. contrary to a generic variant, the handling of *0-sup.* variables to be abstracted is then not necessary.

(2) In case $\mathtt{op} \in \{\wedge, *\}$ the *p*`ZAbstract`-algorithm implements the universal quantification. Contrary to the above setting, a *0-sup.* variable to be abstracted from must be considered on the current path, since $f|_{\mathsf{v}=0} * 0 = 0$. I.e. one can immediately terminate the current recursion and return the terminal 0-node as result.

It is straight forward to extend the *p*`ZAbstract`-algorithm to the case of abstracting from sets of variables instead of a singe variable.

The pseudo-code of the generic *p*`ZAbstract`-algorithm is given as Algo. 4.2. It takes the following arguments as input parameters:

(1) the binary operator `op` for steering the merging of collapsing paths,

Algorithm 4.2 The *p*ZAbstract-algorithm

```
pZAbstract(op, N^abs, n, N)
(0)   node  t, e, res;

/* Reached terminal nodes, end of recursion */
(1)   IF (N = ∅ || N^abs = ∅)
(2)       THEN res := n;

/* Check op-cache if result is already known */
(3)   res := cacheLookup(pZAbstract, op, N^abs, n, N);
(4)   IF res ≠ ε THEN RETURN res;

(5)   var v_i := min(N^abs), v_n := var(n);
(6)   N^abs := N^abs \ v_i;

/* Variable to be abstracted is located below v_i */
(7)   WHILE v_i > min(N) DO N := N \ min(N); END

/* Reached variable to be abstracted */
(8)   IF v_n ≥ v_i THEN

/* Variable to be abstracted is 0-sup. */
(9)       IF v_i ≠ v_n THEN
(10)          t := 0-node;
(11)          e := pZAbstract(op, N^abs, n, N);

/* Reached node carrying variable to be abstracted */
(12)      ELSE
(13)          t := pZAbstract(op, N^abs, then(n), N);
(14)          e := pZAbstract(op, N^abs, else(n), N);

/* Merge collapsing paths */
(15)      res := pZApply(op, t, N, e, N);


/* Reached node carrying variable not to be abstracted */
(16)  ELSE
(17)      t := pZAbstract(op, N^abs, then(n), N);
(18)      e := pZAbstract(op, N^abs, else(n), N);
(19)      res := getUniqueZMTBDDNode(v_n, t, e);

/* Insert result into pZAbstract-cache and terminate recursion */
(20)  cacheInsert(pZAbstract, op, N^abs, n, N, res);
(21)  RETURN res;
```

(2) the set of variables to be abstracted from ($\mathcal{V}^{abs}$),
(3) the root node of the pZDD to be manipulated (n), and
(4) the set $\mathcal{V}$ representing the set of variables of the pZDD to be manipulated.

In line 1-2 one tests if the terminal condition for terminating the recursion is satisfied. If this is the case, a respective node is returned, otherwise one tests at first if a result from a previous recursion is known (line 3-4). In case the cache-lookup does not deliver such a result, the recursion is entered, where three different cases must be covered (line 6-19):

(1) The pseudo-code of line 7 simply causes a skipping of levels referring to *0-sup.* variables not to be abstracted from, since the resulting pZDD does not need to contain here any node as well.

(2) The pseudo-code of line 8-15 covers the case, that the variable to be removed is *0-sup.* or appears in the current path.
(3) The pseudo-code of line 16-19 covers the case, that the algorithm reached a node referring to a variable not to be abstracted.

As one may note, in line 15 the *p*`ZApply` and not the `ZApply`-algorithm is called for merging the collapsing paths, even though the pZDD-graphs to be merged are defined on the same set of variables. This is correct, since in case of non-zero-preserving operators it might be necessary to allocate nodes for variables which were previously omitted. However, in case of zero-preserving operators, one may call here the `ZApply`-algorithm for merging collapsing paths, rather than calling the more generic *p*`ZApply`-algorithm (line 2 and 15).

5 Applications

5.1 Implementation

Jinc [8, 18] is an object-oriented BDD library written in C++. Its key features can be summarized as follows:

(1) Clean API to reduce errors while implementing symbolic algorithms and to make source code more readable.
(2) All data-structures needed for an efficient BDD library (such as unique tables, hash tables, variables, memory pool, ...) are implemented as templates and can be used for regular and weighted variants. This allows an easy handling, when implementing new types of DDs.
(3) Advanced techniques for memory management, where Jinc uses a memory pool in order to prevent memory fragmentation.
(4) For increasing the hits in the table of pre-computed results, the package uses a delayed garbage collection. Like in other BDD packages the reference count is used to identify nodes to be deleted, but contrary to other packages we solely mark the root nodes and delay the recursive marking of their subgraphs. This comes at the cost that the number of dead nodes cannot be used to start the garbage collection. Instead, the number of deleted functions (or root nodes) is used to decide if the garbage collection should be executed. The advantage of doing so is that deleting and reusing the DD can be performed in constant time. As a result, the computed tables are deleted less frequently which leads to an increased number of hits in the computed table.
(5) Insertion of variables at any position of the variable ordering.
(6) All reordering methods are based on the swap of two neighboring variables. This enables all reordering methods for a new DD type to be implemented as soon as the swap function is available.

When implementing pZDDs within Jinc, we decided to store the set of variables for each pZDD object as a cube set, represented by its own BDD. As a consequence, the algorithms operating on pZDDs not only traverse the respective graphs, but also have to traverse the BDDs representing the sets of function variables. This approach is efficient because it supports the existence test in linear time, i.e. it can be checked in linear time if a variable is inside a cube set. But in our implementation the variable

set for functions computed on the basis of the *p*`ZApply`-algorithm may not be minimal. The reason for this is the fact that we assign the union of the set of function variables of the operand pZDD-objects as set of function variables to the newly generated pZDD object. It might occur, that some of the function variables are not essential for the resulting function and could therefore be eliminated from the set of function variables. Thus implementing pZDDs and their algorithms was achieved in a direct way, since the union of two cube sets can be computed very efficiently by standard BDD operation, where the resulting BDD is not only passed to the pZDD manipulating algorithm as input parameter, but also serves as representation of the set of function variables of the resulting pZDD. –In case of co-factor and exists calculation, the set of variables of the ZDD to be generated can be obtained by applying a set-minus on the resp. sets of variables.– In addition to this, also the set of function variables of the operand pZDDs must be passed to the pZDD-manipulating algorithm, so that the appropriate case distinction for the next recursions can be made, i.e. mainly to decide if the recursive behavior is as usual, or if a *0-sup*.- or *dnc*-semantics for the current variable is applicable.

One may note that we previously implemented pZDDs within Cudd. However, integrating this implementation into the Prism tool [19], which makes use of Cudd, seems to be cumbersome, as Prism does not export variable sets for the symbolic structures to be generated during state graph generation. Therefore we decided to start from scratch, which solely required the implementation of pZDDs within Jinc and a replacement of Cudd in our symbolic analyzer for Markov models. Furthermore, this procedure allowed us also to directly benchmark pZDDs with the tool Promoc [20], as this (fully) symbolic probabilistic model checker is based on the Jinc package and the Prism input language and its symbolic semantics.

5.2 pZMTBDDs in the context of Markov reward models

In the past decade, DDs have been successfully employed for efficiently representing stochastic state graphs (SG). Many different approaches have been proposed for efficiently generating such symbolic representations from high-level model descriptions, such as Generalized Stochastic Petri Net [3], Stochastic Process Algebra [7], among others. Roughly 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 runtime, 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.

[11] introduced the activity-local SG generation for efficiently generating state graphs (SGs) or activity-labeled continuous time Markov chains (CTMCs) as underlying high-level model descriptions. For representing such low-level models, [11] employed pZMTBDDs, where as high-level descriptions Markovian extensions of well-known model description techniques as mentioned above were considered. For numerically computing the measures of interest specified on high-level models, the latter must be transformed

(A) Model features and data of MTBDD based analysis

N	*states*	*trans*	t_g in sec.	number of MTBDD nodes		
				$size(\mathsf{Z}_T)$	$size(\mathsf{Z}_R)$	sz_{pk}
Kanban						
6	1.1261 $E7$	1.1571 $E8$	1.1441	4.9664 $E4$	2.9280 $E3$	1.4937 $E6$
8	1.3387 $E8$	1.5079 $E9$	5.2123	1.2413 $E5$	6.9620 $E3$	5.9383 $E6$
10	1.0059 $E9$	1.2032 $E10$	16.3570	2.1054 $E5$	1.1244 $E4$	1.5159 $E7$
12	5.5199 $E9$	6.8884 $E10$	51.1752	3.2744 $E5$	1.6842 $E4$	2.1841 $E7$
FMS						
6	5.3777 $E5$	4.2057 $E6$	0.90006	9.0190 $E4$	4.559 $E4$	9.7921 $E5$
8	4.4595 $E6$	3.8534 $E7$	2.52816	2.2991 $E5$	1.0554 $E4$	2.5860 $E6$
10	2.5398 $E7$	2.3452 $E8$	5.27633	4.1550 $E5$	1.7502 $E4$	4.9215 $E6$
12	1.11415 $E8$	1.07892 $E9$	9.36458	6.7230 $E5$	2.6372 $E4$	8.2828 $E6$

(B) Data of pZMTBDD based analysis and ratios

N	t_g in sec.	number of ZDD nodes			ratios			
		$size(\mathsf{Z}_T)$	$size(\mathsf{Z}_R)$	sz_{pk}	r_{t_g}	r_{Z_T}	r_{Z_R}	r_{pk}
Kanban								
6	0.74005	3.7960 $E3$	2.6100 $E2$	6.0926 $E5$	1.55	13.08	11.22	2.45
8	2.80017	6.1420 $E3$	4.0200 $E2$	2.1885 $E6$	1.86	20.21	17.32	2.71
10	8.22451	9.0550 $E3$	5.5800 $E2$	6.2463 $E6$	1.99	23.25	20.15	2.43
12	23.12544	1.2459 $E4$	7.3900 $E2$	1.5347 $E7$	2.21	26.28	22.79	1.42
FMS								
6	0.37202	1.7406 $E4$	6.0600 $E2$	2.6896 $E5$	2.42	5.18	7.52	3.64
8	0.91606	3.7688 $E4$	7.3900 $E2$	5.7405 $E5$	2.76	6.10	14.28	4.50
10	1.69611	6.9753 $E4$	1.6270 $E3$	1.0259 $E6$	3.11	5.96	10.76	4.80
12	2.80418	1.1589 $E5$	7.3900 $E2$	1.6665 $E6$	3.34	5.80	11.30	4.97

Table 1 Data for the two benchmark models

into a continuous time Markov chain (CTMC), annotated with reward values. If a high-level model description technique does not possess a symbolic semantics, symbolic representations of annotated CTMCs can only be deduced from a high-level model description by explicitly executing the high-level model and encoding of the detected state-to-state transitions. For doing this in a memory and run-time efficient manner, the activity-local SG generation scheme exploits local information of high-level model constructs only. I.e. for keeping explicit SG generation and encoding of transitions as partial as possible, the scheme exploits a dependency relation on the activities and partitions the set of transitions into subsets, each containing the transition asscociated with a specific activitiy. The symbolic representations of the obtained activity-local transition systems depend hereby solely on the binary variables encoding those state counters which are connected to the respective activity (= dependent state variables (SVs)). A symbolic representation of the overall CTMC is constructed by applying a symbolic composition scheme on the previously generated activity-local structures, yielding the potential CTMC. For extracting the reachable states, one must execute symbolic reachability analysis. Since symbolic composition and symbolic reachability analysis are the most time consuming part ($70-99\%$ of the CPU time) of the activity-local scheme, this gives an adequate framework for benchmarking pZMTBDDs. We implemented the activity-local scheme within the Möbius modelling framework [17],

n	k	states	nodes		savings in %
			MTBDD	pZDD	
8	6	2529	18274	12437	31.94
8	7	12673	35303	24183	31.50
8	8	395	8703	5615	35.48
10	6	3941	31888	23823	25.29
10	7	19761	66085	45873	30.58
10	8	6931	43466	31134	28.37
12	6	833	20415	13638	33.20
12	7	28417	101605	64910	36.12
12	8	2815	42024	28462	32.27

n	k	run-time		savings in %	E[π]
		MTBDD	pZDD		
8	4	42	33	21.42	113
8	5	189	111	41.27	66
8	6	54	35	35.19	44
8	7	91	49	46.15	30
8	8	12	10	16.67	20

Table 2 Comparison of MTBDDs and pZDDs with the UMTS model

where our implementation allows us to use either MTBDDs or pZMTBDDs. This makes it possible not only to use the same number of variables with both data structures, but also to maintain the same variable ordering when constructing the symbolic representations. For the experiments, several benchmark models from the literature were analyzed. Here we present results for the Kanban model and the Flexible Manufacturing System (FMS) model, both included in the standard case studies of the Prism tool [19]. Tab. 1 (A) gives the number of states (*states*) and transitions (*trans*) of the respective CTMC. These characteristic figures depend on the model scaling parameter N, which is also given. In the right part of Tab. 1 (A) and in the left part of Tab. 1 (B) the time required for generating the symbolic structures (t_g), as well as the number of nodes within the symbolic structures are given, where Z_R represents the set of reachable states, Z_T the CTMC and sz_{pk} denotes the peak number of nodes allocated during the construction process. Note that the nodes of the cube set are also counted to the size of the pZMTBDDs. The right part of Tab. 1 (B) finally gives the respective ratios for comparing MTBDD and pZMTBDDs, where we normed everything to the figures of our new data structure. As illustrated by these figures, employing pZMTBDDs clearly reduces the run-time and space requirement.

Once the CTMC is generated, the next step in the analysis is the computation of transient or steady-state probabilities. In [12] we adapted the hybrid solution method [16] to ZMTBDDs for solving CTMCs in a run-time and memory efficient way. When applying a numerical solution method such as Jacobi, pseudo Gauss-Seidel or uniformization, the sparsity of ZMTBDDs pays off another time, leading to a clear reduction of CPU-time consumptions by a factor between 2 and 3. As a consequence, when employing ZMTBDDs instead of MTBDDs, performance measures for the FMS model and a scaling parameter $N = 12$ can be computed in $\approx 4h$ instead of $\approx 12h$ (for further details please refer to [12]).

5.3 pZDDs in the context of probabilistic model checking

The model that is used for this benchmark study is a simplification of the UMTS system. It represents the mechanism to request validation keys for different domains (like telephone or internet access). The model also identifies if a synchronization failure

occurs, i.e. if a used key is older than the stored key in the UMTS card. The system size depends on the number of slots n on the UMTS card, as well as on the number of keys k that are requested at a time. (Note, that the model size collapses if k is a divider of n.) We specified this model by employing the symbolic probabilistic model checker Promoc [20], which is based on the Prism input language and the Jinc package. For probabilistic model checking, a symbolic representation of a transition matrix is directly derived from the (Prism) model specification. Each matrix entry $m_{i,j}$ hereby defines the transition probability that the system moves from state i to state j. For investigating the stationary probability of states with a given property one needs to solve a system of linear equations and to calculate the expected number of requests ($E[\pi]$) to reach a synchronization failure. The benchmark results in Table 2 show that pZMTBDDs clearly outperform the regular MTBDDs in both size and run-time.

6 Conclusion

In this paper we extended ZBDDs [14] to the multi-terminal case, in order to employ the *0-sup.*-reduction rule in the context of symbolic, quantitative analysis of systems. For efficiently working with ZDDs defined on differing sets of function variables we introduced the concept of partially shared ZDDs and described the resp. algorithms for manipulating them. This not only allowed us to implement pZDDs within standard shared DD-environments, such as Cudd [25] or Jinc [8], but also supports the application of non-zero-preserving operators to them. The efficiency of the introduced approach was then demonstrated by analyzing various case studies, where pZDDs turned out to require less memory space and less CPU time if compared to the standard type of MTBDDs. The superior performance of pZDDs can be explained by the following reasons: (a) It is typical that matrices derived from high-level model descriptions are sparsely populated. (b) Many positions of the bit strings encoding system states (and thus referring to the indices of reachable states) carry the value 0, and ZDDs are very efficient at representing sets of such bit strings. (c) The concept of partially shared ZDDs avoids the insertion of *dnc*-nodes. It therefore keeps the symbolic structures compact and even allows to represent different functions by the same graph. The size reduction of the symbolic structures leads to run-time advantages, where it was found that the overhead imposed by the handling of sets of (function) variables is of minor concern.

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. G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, M. Ajmone Marsan, and M. Ajmone Marsan. *Modelling with Generalized Stochastic Petri Nets*. JOHN WILEY & SONS, 1995.
4. R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
5. G. Ciardo, G. Lüttgen, and A. S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *Form. Methods Syst. Des.*, 31(1):63–100, 2007.
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), LNCS 1785, pages 395–410, Berlin, 2000. Springer.
7. H. Hermanns, U. Herzog, and V. Mertsiotakis. Stochastic Process Algebras - Between LOTOS and Markov Chains. *Computer Networks and ISDN Systems, 30 (9-10)*, pages 901–924, 1998.
8. JINC BDD package. web page www.jossowski.de.
9. T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1-2):9–62, 1998.
10. M. Kuntz, M. Siegle, and E. Werner. Symbolic Performance and Dependability Evaluation with the Tool CASPA. In *Proc. of EPEW*, pages 293–307. Springer, LNCS 3236, 2004.
11. K. Lampka and M. Siegle. 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 and M. Siegle. Analysis of Markov Reward Models using Zero-supressed Multi-terminal decision diagramms. In *Proceedings of VALUETOOLS 2006 (CD-edition)*, October 2006.
13. C.Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell Systems Technical Journal*, 38:985–999, July 1959.
14. S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proc. of the 30th Design Automation Conference (DAC)*, pages 272–277, Dallas (Texas), USA, June 1993. ACM / IEEE.
15. S. Minato. Zero-suppressed BDDs and their applications. *International Journal on Software Tools for Technology Transfer*, 3(2):156–170, May 2001.
16. A. Miner and D. Parker. Symbolic Representations and Analysis of Large State Spaces. In Ch. Baier, B. Haverkort, H. Hermanns, J.-P. Katoen, and M. Siegle, editors, *Validation of Stochastic Systems*, LNCS 2925, pages 296–338, Dagstuhl (Germany), 2004. Springer.
17. Möbius web page. www.moebius.crhc.uiuc.edu.
18. Joern Ossowski and Christel Baier. A uniform framework for weighted decision diagrams and its implementation. Accepted for publication in STTT.
19. PRISM web page. www.prismmodelchecker.org.
20. PROMOC modelling tool. web page www.jossowski.de.
21. T. Sasao and M. Fujita, editors. *Representations of Discrete Functions*, volume 1. Kluwer Academic Publishers, Dordrecht The Netherlands, 1996.
22. 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.
23. M. Siegle. Advances in model representation. In Luca de Alfaro and Stephen Gilmore, editors, *Proc. of the Joint Int. Workshop, PAPM-PROBMIV 2001, Aachen (Germany)*, LNCS 2165, pages 1–22. Springer, September 2001.
24. SMART web page. www.cs.ucr.edu/∼ciardo/SMART.
25. F. Somenzi. Cudd: CU decision diagram package release, 1998.
26. Ingo Wegener. *Branching Programs and Binary Decision Diagrams*. SIAM, 2000.