Partially-shared zero-suppressed multi-terminal BDDs
Concept, algorithms and applications

Form Methods Syst Des (2010) 36: 198–222
DOI 10.1007/s10703-010-0095-8

Partially-shared zero-suppressed multi-terminal BDDs: concept, algorithms and applications

Kai Lampka · Markus Siegle · Joern Ossowski · Christel Baier

Published online: 10 June 2010

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.

K. Lampka (✉)
Computer Engineering and Communication Networks Lab., ETH Zurich, Zurich, Switzerland
e-mail: lampka@tik.ee.ethz.ch

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

J. Ossowski · C. Baier
Inst. for Theoretical Comp. Sc., Technical University Dresden, Dresden, Germany

J. Ossowski
e-mail: mail@jossowski.de

C. Baier
e-mail: baier@tcs.inf.tu-dresden.de

Keywords Binary Decision Diagrams and their algorithms · Quantitative verification of systems · Symbolic data structures for performance analysis

1 Introduction

1.1 Motivation

Finite state/transition systems (also called state graphs (SG)) are a fundamental ingredient when it comes to formal, automatized reasoning about the qualitative and quantitative behavior of systems. Contemporary tools for evaluating system behavior accept a high-level model description as input rather than the plain state/transition system. However, for analyzing a high-level system model, it has to be transformed into a state/transition-system which incorporates all possible behavior. This step, known as state space exploration, expands all concurrency as contained in the high-level model. It may therefore lead to an exponential blow-up in the number of system states, which is known as the notorious *state space explosion problem*.

Decision diagrams (DDs) are directed acyclic graphs for representing finite functions. They have shown to be very helpful when it comes to the generation, representation and analysis of extremely large state/transition-systems, easing the restriction imposed on the size and complexity of models and thus systems to be analyzed. Multi-Terminal Binary Decision Diagrams (MTBDDs) are among the most efficient techniques for the state based quantitative analysis of large and complex systems, commonly described by Markovian extensions of well known high-level model description techniques, e.g. Stochastic Process Algebra [7] or Generalized Stochastic Petri Nets [3], among many others. The success of modeling and evaluation tools such as the probabilistic symbolic model checker PRISM [20], the stochastic process algebra tool CASPA [11] (both based on MTBDDs) or the modeling and analysis tool SMART [25] (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 usually 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 by Minato in [15] has the potential to improve such situations, since, contrary to MTBDDs, it avoids allocating 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 state/transition systems 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 two (or more) zero-suppressed Binary Decision Diagrams (ZBDDs) 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 in the case of ZBDDs with a global set of variables. The Shannon expansion [23] requires that for deducing the function represented by a node of a ZBDD, the set of its variables $\mathcal{F}$ must be known since skipped variables from $\mathcal{F}$ are assumed to be *0-sup*, whereas skipped variables not in $\mathcal{F}$, which are also defined within the respective environment, are assumed to have a *don't-care* semantics. Consequently, in the presence of multi-rooted DDs [22], as provided by standard implementations such as CUDD [26] or the recently developed package JINC [9, 19], the nodes of the ZBDDs 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 ZBDDs cannot be employed directly.*

There are many applications where several functions, depending on different sets of variables, need to be manipulated at the same time. For example, symbolic model checking requires the execution of a (symbolic) reachability analysis, where a high-level model's state/transition system is given as a DD `Trans`. The symbolic representation of `Trans` is commonly defined over two sets of variables, namely the s-variables, encoding the values of the potential source states and the t-variables, encoding the values of the potential target states. Thus each path within the DD defines at least one state-to-state-transition. Contrary to this, the symbolic structure `Unexpl`, encoding the set of unexplored states, solely takes the s-variables as input. For obtaining all transitions emanating from the states of `Unexpl`, one must multiply DDs `Trans` and `Unexpl`, even though they are defined on different sets of variables. In the presence of Minato's *0-sup* reduction rule and when recursing on the DDs for applying operators such as multiplication, it is essential, that one distinguishes whether a variable is an input variable or not for the respective DD, since this determines the applicable semantics, i.e. *dnc*- or *0-sup*. Standard shared DD environments as provided by CUDD [26] solely assume global sets of variables, therefore their implementations of ZBDDs fail to apply operators to DDs with differing sets of variables.

1.2 Contributions

In [12, 13] we had introduced an extension of Minato's ZBDDs to the multi-terminal case, thereby obtaining zero-suppressed Multi-Terminal Binary Decision Diagrams (ZMTBDDs), and used this data structure for the quantitative analysis of systems. In the present article we describe this extension in full detail, show that this new type of decision diagram (DD) is a canonical representation for multi-valued functions on finite input sets and introduce algorithms for efficiently manipulating DDs of that kind. When applying Minato's 0-suppressing (*0-sup*) reduction rule, the sharing of the graphs of the DDs is not trivial any more. For deducing the function represented by a *0-sup* DD's graph correctly, the set of Boolean input variables must be known. Thus, within (fully) shared DD-environments as provided by contemporary DD-packages, nodes of a *0-sup* DD lose their uniqueness, if the DDs are meant to be defined on differing sets of variables. To solve this problem in an efficient way, this paper develops the concept of *partially shared zero-suppressed DDs* (pZDDs), 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 pZDDs 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*).

The concept of partially shared DDs as presented here is not the only solution to the problem of differing sets of variables, orthogonal procedures exist: Since the semantics 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 those variables which are not in the input set 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 ZBDDs agree with pseudo-reduced BDDs.

For evaluating the efficiency of the presented approach, the paper compares pZDDs 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, ZDDs 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

Verification of stochastic systems plays an important part when it comes to ensuring the correctness of timed (soft-real-time) systems. Over the last decade, many derivatives of decision diagrams (DDs) were developed, which have turned out to be very useful for representing stochastic transition relations. The most prominent types in this context are *multi-terminal* Binary Decision Diagrams (MTBDDs) [1, 6, 24] and (multi-terminal) *Multi-valued Decision Diagrams* (MDDs) [5, 10]. However, all these derivatives are mainly extensions of Binary Decision Diagrams (BDDs) [2, 14], for which Bryant [4] designed algorithms for efficiently manipulating them and keeping them reduced. Variants of his algorithms have been implemented in contemporary DD-packages providing *multi-rooted DDs [22] defined on a global set of variables.* In this area, Minato's idea of employing the *0-sup*-reduction rule is also well known, and ZBDDs have found their way into several contemporary DD-packages. In his work [15, 16], Minato focuses on the representation of combinatorial sets, rather than on the representation of characteristic functions of sets encoded as bit strings, which is the reason why the concept of ZBDD local sets of input variables was not considered before (cf. Sect. 2, p. 203 for more details). To the best of our knowledge, previous work on *0-sup* DDs [15, 16, 27] did not consider the multi-terminal case. Furthermore, it required, when applying n-ary operators to ZBDDs, that all operands be defined on the same set of variables for making the manipulating algorithms work properly. This work overcomes these limitation by developing the concept of *partially shared 0-sup* DDs, which makes it possible to implemented *0-sup* DDs with differing sets of input variables within a single DD environment in an intuitive way. Since this idea also applies to Minato's standard *0-sup* BDDs, this article clearly extends previous works and thus ultimately allows one to employ zero-suppressed DDs in a wide range of applications.

1.4 Organization

Section 2 presents the background material and repeats some useful concepts with respect to Boolean functions and their representation. Section 3 introduces our new type of DD and discusses its canonicity aspects. Section 4 introduces the concept of partially shared *0-sup* DD and presents generic algorithms for efficiently manipulating them. Section 5 describes our implementation of pZDDs within JINC, as well as the benchmarking experiments carried out, and Sect. 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 $\mathcal{F} := \{\mathsf{v}_1, \dots, \mathsf{v}_n\} \subseteq \mathcal{V}$ employed in a Boolean function f is referred to as set of function variables (or, synonymously, 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 $f(\mathsf{v}_1, \dots, \mathsf{v}_{i-1}, 0, \mathsf{v}_{i+1}, \dots, \mathsf{v}_n) \neq f(\mathsf{v}_1, \dots, \mathsf{v}_{i-1}, 1, \mathsf{v}_{i+1}, \dots, \mathsf{v}_n)$. Otherwise the variable v_i is not essential. A non-essential variable is also commonly known as *don't-care* (*dnc*) variable.

Canonical representations of Boolean functions 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 *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 refer to 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 cofactors of f with respect to v_i, namely (a) the one-cofactor $f|_{\mathsf{v}_i:=1} := f(\mathsf{v}_1, \dots, \mathsf{v}_{i-1}, 1, \mathsf{v}_{i+1}, \dots, \mathsf{v}_n)$ or (b) the zero-cofactor $f|_{\mathsf{v}_i:=0} := f(\mathsf{v}_1, \dots, \mathsf{v}_{i-1}, 0, \mathsf{v}_{i+1}, \dots, \mathsf{v}_n)$. 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 cofactors, which are also denominated positive and negative cofactor. For all $\mathsf{v}_i \in \mathcal{F}$ it holds:

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

This so-called *Shannon-expansion*, introduced in 1938 by Shannon in the context of switching functions [23], 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-cofactors 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 sub-function 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 (cf. 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 *0-sup* if and only if its one-cofactor is the constant zero-function ($f_1 = 0$). Applying the Shannon expansion to a function f and a *0-sup* variable v_k yields:

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

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 [2, 4, 14]. It consists of a set of inner nodes and a set of terminal nodes, where the inner nodes are labeled 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 labeling 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 [22].

A node referring to a *0-sup*-variable is a *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 [15]. A ZBDD is a weakly canonical representation of a Boolean function [22].

A note on combinatorial sets As already pointed out in the introduction, Minato originally developed ZBDDs for representing combinatorial sets [15, 16]. Instead of representing such sets by their usual characteristic function, he uses a different semantics: His symbolic representation only depends on those variables which are contained in at least one valid combination of the combinatorial set. With this kind of encoding, it turns out that ZBDDs are actually strongly canonical representations and that *dnc*-reduced BDDs are only weakly canonical representations of combinatorial sets. In the following, however, we will stick to the usual interpretation, i.e. solely non-essential variables are considered as being irrelevant for the function under consideration.

3 Data structure

We consider n-ary pseudo-Boolean functions, i.e. functions of the type $f : \mathbb{B}^n \to \mathbb{D}$. 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 $\mathsf{A} = (\mathcal{K}_{NT}, \mathcal{K}_T, \mathcal{F}, \texttt{var}, \texttt{then}, \texttt{else}, \texttt{value}, \texttt{root})$ 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, labeling 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. $\mathtt{var}: \mathcal{K}_{NT} \cup \mathcal{K}_T \to \mathcal{F} \cup \{\mathsf{t}\}$ such that $\forall k \in \mathcal{K}_{NT} \cup \mathcal{K}_T$: $\mathtt{var}(k) = \mathsf{t} \Leftrightarrow k \in \mathcal{K}_T$.
4. $\mathtt{then}: \mathcal{K}_{NT} \to \mathcal{K}_{NT} \cup \mathcal{K}_T$ such that $\forall n \in \mathcal{K}_{NT}$: $\mathtt{var}(n) <_\pi \mathtt{var}(\mathtt{then}(n))$.
5. $\mathtt{else}: \mathcal{K}_{NT} \to \mathcal{K}_{NT} \cup \mathcal{K}_T$ such that $\forall n \in \mathcal{K}_{NT}$: $\mathtt{var}(n) <_\pi \mathtt{var}(\mathtt{else}(n))$.
6. $\mathtt{value}: \mathcal{K}_T \to \mathbb{D}$, 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. (Isomorphism 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

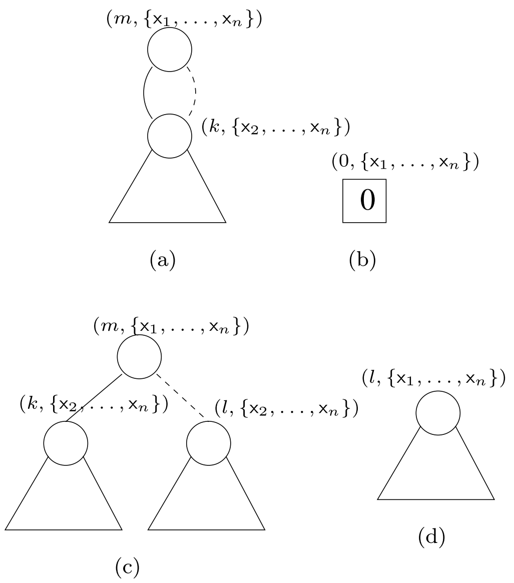
$$f(n, \mathcal{F}) := \left(\prod_{\mathsf{x}_i \in \mathcal{F}} (1 - \mathsf{x}_i) \right) * \mathtt{value}(n)$$

- 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 $\mathtt{var}(n) \in \mathcal{F}$)

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

The last defining equation of Definition 2 is a combination of the Shannon expansion for Boolean functions [23] and the application of the *0-sup*-reduction rule. According to Definition 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.



$(m, \{x_1, \ldots, x_n\})$
$(k, \{x_2, \ldots, x_n\})$
$(0, \{x_1, \ldots, x_n\})$
0
(a)
(b)
$(m, \{x_1, \ldots, x_n\})$
$(k, \{x_2, \ldots, x_n\})$
$(l, \{x_2, \ldots, x_n\})$
$(l, \{x_1, \ldots, x_n\})$
(c)
(d)

Fig. 1 Constructing ZDDs

Theorem 1 (Existence) *Let $\mathcal{F} = \{x_1, x_2, \ldots, x_n\} \subseteq \mathcal{V}$ 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.*

Proof 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 $x_1 <_\pi \cdots <_\pi x_n <_\pi t$.

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 $\{x_2, \ldots, x_n\}$. We show that then the conjecture also holds for the n-ary Boolean function f defined on $\{x_1, x_2, \ldots, x_n\}$. We expand f as

$$f(x_1, \ldots, x_n) = x_1 \cdot f_1(x_2, \ldots, x_n) + (1 - x_1) \cdot f_0(x_2, \ldots, x_n)$$

where $f_1(x_2, \ldots, x_n) = f(1, x_2, \ldots, x_n)$ and $f_0(x_2, \ldots, x_n) = f(0, x_2, \ldots, x_n)$. 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 $(k, \{x_2, \ldots, x_n\})$, where $k \in \mathcal{K}_{NT} \cup \{e\}$, $e \in \mathcal{K}_T$ and `value`$(e) = 1$ (i.e. k can be the terminal one-node). In this case f can be represented by $(m, \{x_1, \ldots, x_n\})$ as shown in Fig. 1(a).

Case 2: If $f_1 = f_0 = 0$ then f_1 and f_0 are both represented by $(k, \{x_2, \ldots, x_n\})$, where $k \in \mathcal{K}_T$ and `value`$(k) = 0$ (the terminal zero-node). In this case f is also represented by the zero-node, i.e. by $(0, \{x_1, \ldots, x_n\})$ 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 $(k, \{x_2, \ldots, x_n\})$ and f_0 is represented by $(l, \{x_2, \ldots, x_n\})$, then f can be represented by $(m, \{x_1, \ldots, x_n\})$ as shown in Fig. 1(c).

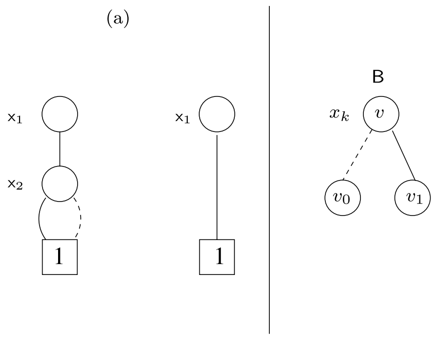


Fig. 2 (**a**) Weak canonical representation. (**b**) Illustrating the proof

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). □

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 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 $\mathsf{x}_1 <_\pi \mathsf{x}_2 <_\pi \cdots <_\pi \mathsf{x}_m <_\pi \mathsf{t}$ 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 labeled with c.

Induction step: We will now assume that the root nodes of B and C are non-terminal nodes, each labeled 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 $v_0 = \texttt{else}(v)$, $v_1 = \texttt{then}(v)$, $w_0 = \texttt{else}(w)$ and $w_1 = \texttt{then}(w)$ (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. □

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 Boolean and 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 [22]. 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-dependent 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 at the end of Sect. 1.1, 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 Isomorphism of pZDD-nodes

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

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

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

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

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-dependent 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.

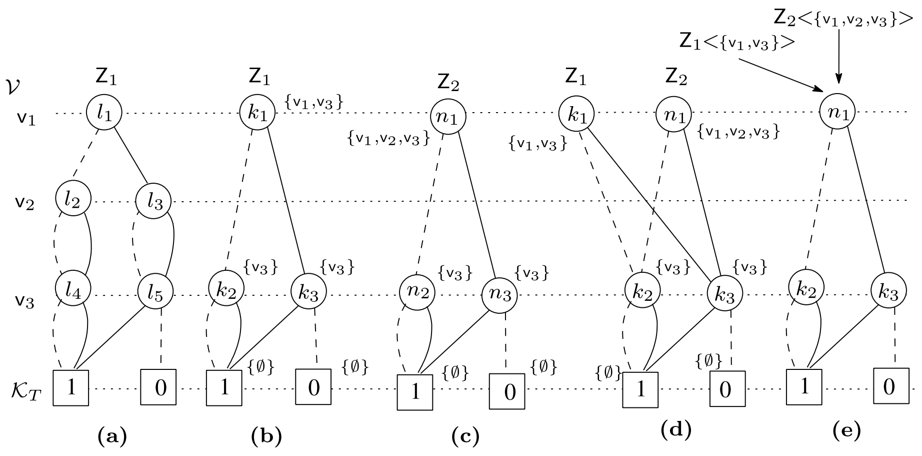


Fig. 3 Allocating multi-rooted pZDDs within standard DD-environments

An exemplification is provided by Fig. 3. Let the set of all variables defined in the shared DD-environment be $\mathcal{V}^G := \{\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3\}$. 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 nodes 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 statically 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 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 dynamically associated with a set of function variables.*

4.2 Applying binary operators to pZDDs

A symbolic representation of a function $f := g \,\mathtt{op}\, h$ is computed by executing the generic pZApply-algorithm or its variants. However, before we give details on the algorithms we comment on the generation of sets of function variables when applying binary operators to pZDDs. Furthermore, we will also briefly introduce the concept of operator functions, which keeps the pZApply-algorithm and its variants more generic, due to the separation of operator-specific terminal conditions and the pZApply-specific recursion.

4.2.1 Function variables in case of binary operators

Contrary to existing implementations, the algorithms for manipulating pZDDs must not only compute the result pZDD-graph, i.e. return its root node, but also need to generate the correct set of function variables. Let $\mathcal{F}$ be the set of function variables to be associated with the result node *res* for representing the result function f, and let $\mathcal{N}$ and $\mathcal{M}$ be the sets of function variables of the operand pZDDs, the graphs of which are rooted in node n and m respectively. When computing $res := n\ \texttt{op}\ m$ the following situations must be covered:

$\mathcal{F} = \mathcal{N} \cup \mathcal{M}$: Once the pZDD-graph for representing f has been generated nothing needs to be done, since all variables are essential for the result function f.

$\mathcal{F} \subset \mathcal{N} \cup \mathcal{M}$: When computing the pZDD-graph for representing f some variables have become non-essential. The minimal set $\mathcal{F}$ for function f can be determined on-the-fly, i.e. while generating the result pZDD-graph or by executing a depth-first traversal on the latter once the recursion has terminated. Either way this allows one to find out which variables are associated only with *dnc*-nodes, and can therefore be considered as non-essential and thus eliminated.

When removing non-essential variables from $\mathcal{F}$, it is required to execute a reduction routine on the result pZDD-graph, eliminating all respective *dnc*-nodes. However, this can only be done once the result pZDD-graph for representing f has been computed. Therefore, and for keeping the discussion on the *p*`ZApply`-algorithm(s) as simple as possible, we concentrate in the following on the pZDD-graph generating routines and simply take $\mathcal{N} \cup \mathcal{M}$ as the set of function variables of the result pZDD-object; further (implementation) details are given at the end of Sect. 5.1.

4.2.2 Operator function (`op`-function)

Operator functions are (sub-)routines which steer the recursive behavior of the *p*`ZApply`-algorithm(s), such that the latter does not have to contain all of the operator-specific termination criteria itself. This is extremely useful since the conditions for terminating a recursion depend on the binary operator. 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. As input the `op`-functions take two pZDD-nodes n and m, as well as the associated sets of function variables $\mathcal{N}$ and $\mathcal{M}$. These sets are necessary since contrary to existing work here the new rule for node equality (cf. Definition 3) has to be applied. As return value an `op`-function either gives back the reference to the pZDD-node representing the result of $n\ \texttt{op}\ m$ or the empty reference ϵ, also commonly known as NULL-pointer. In the latter case the pZDD manipulating algorithm must proceed with the recursion, since the conditions for termination are not satisfied. Since the concept of partially shared pZDDs also applies to partially shared ZBDDs, we first introduce operator functions for Boolean operators, which also turn out to be more complex than heir arithmetic counterparts.

`op`*-functions implementing binary operators*

1. $\texttt{op} = \wedge$: 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

Algorithm 4.1 pZDD op-functions for Boolean operators

```
ZAnd($n, \mathcal{N}, m, \mathcal{M}$)
(0) node  $res := \epsilon$;
(1) IF ($n = m$ && $\mathcal{N} = \mathcal{M}$)
(2)     THEN $res := n$;
(3) ELSE IF ($n$ = 0-node || $m$ = 0-node)
(4)     THEN $res$ := 0-node;
(5) ELSE IF ($n$ = 1-node && $m$ = 1-node)
(6)     THEN $res$ := 1-node;
(7) RETURN $res$;
```

(a) op-function for op = $\wedge$

```
ZOr($n, \mathcal{N}, m, \mathcal{M}$)
(0) node  $res := \epsilon$;
(1) IF ($n = m$ && $\mathcal{N} = \mathcal{M}$)
(2)     THEN $res := n$;
(3) ELSE IF ($n$ = 0-node)
(4)     THEN $res := m$;
(5) ELSE IF ($m$ = 0-node)
(6)     THEN $res := n$;
(7) RETURN $res$;
```

(b) op-function for op = $\vee$

```
ZSetMinus($n, \mathcal{N}, m, \mathcal{M}$)
(0) node  $res := \epsilon$;
(1) IF ($n = m$ && $\mathcal{N} = \mathcal{M}$)
(2)     THEN $res$ := 0-node;
(3) ELSE IF ($m$ = 0-node)
(4)     THEN $res := n$;
(5) ELSE IF ($n$ = 0-node)
(6)     THEN $res$ := 0-node;
(7) RETURN $res$;
```

(c) op-function for op = $\setminus$

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($\setminus$, *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.

The pseudo-code of the above op-functions is given in Algorithm 4.1, where n and m refer to the nodes of the current recursion, and the sets $\mathcal{N}$ and $\mathcal{M}$ refer to their sets of function variables.

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 labeled with (value(n) op value(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.

4.2.3 The generic `pZApply`*-algorithm*

The algorithm takes a binary operator `op`, i.e. a reference to the operator-function, the respective operand pZDDs, i.e. their root nodes n and m and their sets of function variables $\mathcal{N}$ and $\mathcal{M}$ as input. Its basic idea 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 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 Algorithm 4.2. 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 (cf. Algorithm 4.1). If this is not successful, one checks the `op`-function specific cache (`op`-cache), if the result is already known from a previous recursion (lines 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 pZDD-nodes themselves. In case the look-up 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 (lines 27 and 29). Finally when returning from the recursion, one either

Algorithm 4.2 The generic p`ZApply`-algorithm

```
pZApply($op, n, \mathcal{N}, m, \mathcal{M}$)
(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)     $res := \mathtt{op}(n, \mathcal{N}, m, \mathcal{M})$;
(2)     IF $res \neq \epsilon$ THEN RETURN $res$;

/* Check op-cache if result is already known */
(3)     $res = \mathtt{cacheLookup}(\mathtt{op}, n, \mathcal{N}, m, \mathcal{M})$;
(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 $\mathtt{var}(n) = \mathsf{v}_c$ && $\mathtt{var}(m) = \mathsf{v}_c$ THEN
(8)         $e := p\mathtt{ZApply}(op, \mathtt{else}(n), \mathcal{N}, \mathtt{else}(m), \mathcal{M})$;
(9)         $t := p\mathtt{ZApply}(op, \mathtt{then}(n), \mathcal{N}, \mathtt{then}(m), \mathcal{M})$;

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

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

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

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

newly allocates a new pZDD-node representing $f^n \,\texttt{op}\, f^m$ (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.

4.2.4 *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 ($0 \,\texttt{op}\, 0 = 0$), 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 [15, 16]. The obtained variant will be referred to as *fs*`ZApply`-algorithm in the following.

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 lines 5–6, 12, 14–15, 18, 20–21 and 22–29, by adapting the Boolean tests of lines 7, 10 and 16 accordingly, and modifying the function calls of lines 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.

In the context of our work this fully shared variant allows us to efficiently manipulate all pZDDs which are defined on the same set of function variables. As we recently noticed, this variant is also described in [27], which is not surprising, since this algorithm covers the standard implementation of an `Apply`-algorithm for *0-sup* DDs employed within a fully shared DD environment.

Non-shared ZDDs In case two pZDDs have no variable in common, they can be manipulated by a specialized *p*`ZApply`-algorithm which we refer to 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 lines 7–9, 12–14, and 18–20 of the *p*`ZApply`-algorithm.

The `pZAnd`-*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, 15, 16, 27], 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 $\mathtt{op} \in \{\wedge, *\}$, which we denominate 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 $m_1 = 0$. This can now be employed for computing $f := n * m$ as follows:

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

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

The abstraction from a variable v is implemented by the *p*`ZOpAbstract`-algorithm, called with a respective `op`-function. The *p*`ZOpAbstract`-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. Since h's set of function variables is trivially given by $\mathcal{H} := \mathcal{F} \setminus \{\mathsf{v}\}$ we will once again focus on the pZDD-graph generating procedure. This procedure must not only eliminate all nodes label with the respective variable v, but must also consider the case that v is *0-sup* on the current path.

The pseudo-code of the generic *p*`ZOpAbstract`-algorithm is given as Algorithm 4.3. It takes the following arguments as input parameters:

Algorithm 4.3 The *p*`ZOpAbstract`-algorithm

```
pZOpAbstract(op, $\mathcal{N}^{abs}$, $n$, $\mathcal{N}$)
(0)   node   $t, e, res$;

/* Reached terminal nodes, end of recursion */
(1)   IF ($\mathcal{N} = \emptyset \,\|\, \mathcal{N}^{abs} = \emptyset$)
(2)       THEN $res := n$;

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

(5)   var $\mathsf{v}_i := \min(\mathcal{N}^{abs}), \mathsf{v}_n :=$ var($n$);
(6)   $\mathcal{N}^{abs} := \mathcal{N}^{abs} \setminus \mathsf{v}_i$;

/* Variable to be abstracted is located below $\mathsf{v}_i$ */
(7)   WHILE $\mathsf{v}_i > \min(\mathcal{N})$ DO $\mathcal{N} := \mathcal{N} \setminus \min(\mathcal{N})$; END

/* Reached variable to be abstracted */
(8)   IF $\mathsf{v}_n \geq \mathsf{v}_i$ THEN

/* Variable to be abstracted is 0-sup */
(9)       IF $\mathsf{v}_i \neq \mathsf{v}_n$ THEN
(10)          $t :=$ 0-node;
(11)          $e :=$ pZOpAbstract(op, $\mathcal{N}^{abs}, n, \mathcal{N}$);

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

/* Merge collapsing paths */
(15)      $res :=$ pZApply(op, $t, \mathcal{N}, e, \mathcal{N}$);

/* Reached node carrying variable not to be abstracted */
(16)  ELSE
(17)      $t :=$ pZOpAbstract(op, $\mathcal{N}^{abs}$, then($n$), $\mathcal{N}$);
(18)      $e :=$ pZOpAbstract(op, $\mathcal{N}^{abs}$, else($n$), $\mathcal{N}$);
(19)      $res :=$ getUniqueZMTBDDNode($\mathsf{v}_n, t, e$);

/* Insert result into pZOpAbstract-cache and terminate recursion */
(20)  cacheInsert(pZOpAbstract, op, $\mathcal{N}^{abs}, n, \mathcal{N}, res$);
(21)  RETURN $res$;
```

1. the binary operator `op` for steering the merging of collapsing paths,
2. the set of variables to be abstracted from ($\mathcal{N}^{abs}$),
3. the root node of the pZDD to be manipulated (n), and
4. the set $\mathcal{N}$ representing the set of variables of the pZDD to be manipulated.

In lines 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 (lines 3–4). In case the cache-look-up does not deliver such a result, the recursion is entered, where three different cases must be covered (lines 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 lines 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 lines 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 (lines 2 and 15).

Depending on the operator `op` passed as an argument, the *p*`ZOpAbstract` implements the following operations:

1. In case `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 `op` $\in \{\wedge, *\}$ the *p*`ZOpAbstract`-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*`ZOpAbstract`-algorithm to the case of abstracting from sets of variables instead of a singe variable.

5 Applications

In this section we present and discuss some empirical results in order to evaluate the usefulness of the new data structure. The measured data is obtained when employing pZDDs in the context of model-based performance evaluation and quantitative verification of systems, which is also an important part of our work. As already pointed out in the introduction, this area of application enforces the manipulation of DDs which are defined on different sets of variables, which in case of *0-sup* DDs requires the use of our new algorithms. The experiments were carried out within our own DD library JINC, since this provided us with a comfortable implementation environment. However, it would also be possible to build a similar implementation based on different DD environments, such as CUDD, for example.

5.1 Implementation

In a first step we implemented pZDDs within CUDD and successfully employed the obtained package within our new symbolic performance evaluation engine which we developed for the Möbius modeling framework (cf. Sect. 5.2). However, integrating the obtained pZDD package into the stochastic, symbolic model checkers PRISM [20] and CASPA [11], which also make use of CUDD, turned out to be cumbersome. The reason for this is as follows: ZDDs, contrary to MTBDDs, need to be equipped with sets of function variables. When translating high-level model descriptions into state/transition systems one commonly generates symbolic representations of transition functions. Their function variables encode the values of the state counters before and after the resp. (high-level) activity has been executed. Thus, one needs to know which Boolean function variables are associated with which state counter of the high-level model, enabling one to extend the (module-local) transition functions, their symbolic representation respectively, with the respective set of function variables. The a posteriori extraction of such internal information would have been very work-intensive, so that we decided to employ our own (fully) symbolic probabilistic model checker PROMOC [21], which also supports the PRISM input language and its symbolic semantics. However, contrary to PRISM and CASPA, PROMOC is based on the DD-package JINC, rather than on CUDD. Thus we integrated the concept of pZDDs within JINC. Overall, this strategy allowed us not only to benchmark pZDDs with the tool PROMOC, but also with our symbolic performance analysis engine of Möbius, where in case of the latter we simply had to replace CUDD by JINC, which was straight-forward.

JINC [9, 19] is an object-oriented BDD library written in C++. Its key features can be summarized as follows:

1. Clean object-oriented 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, etc.) are implemented as templates with regard to modern programming techniques (that enables current compilers to generate well optimized code) 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 and for faster memory allocation.
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 and their algorithms 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 at the same time 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. However, in our implementation the variable set for functions computed when applying binary operators to pZDDs may not be minimal. The reason for this is the fact that we assign the *union* of the sets 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 (and the *dnc*-nodes referring to these variable could be eliminated). On the other hand, our strategy implements pZDDs and their algorithms in a straight-forward manner, since the union of two cube sets can be computed very efficiently by standard BDD operation, where the resulting BDD serves as representation of the set of function variables of the resulting pZDD. In case of cofactor and exists calculation, the set of variables of the ZDD to be generated can be obtained by applying a set-minus on the respective sets of variables.

5.2 pZDDs 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 [8], 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.

[12] 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, [12] employed pZDDs, 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 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 associated with a specific activity. 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 pZDDs. We implemented the activity-local scheme within the Möbius modeling framework [18], where our implementation allows us to use either MTBDDs or pZDDs. 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

Table 1 Data for the two benchmark models

(A) Model features and data of MTBDD based analysis

				Number of MTBDD nodes		
N	*states*	*trans*	t_g in sec.	$size(Z_T)$	$size(Z_R)$	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 pZDD based analysis and ratios

		Number of pZDD nodes			Ratios			
N	t_g in sec.	$size(Z_T)$	$size(Z_R)$	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

the PRISM tool [20]. Table 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 Table 1(A) and in the left part of Table 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} is 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 pZDDs. The right part of Table 1(B) finally gives the respective ratios for comparing MTBDD and pZDDs, where we normed everything to the figures of our new data structure. As illustrated by these figures, employing pZDDs 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 [13] we adapted the hybrid solution method [17] to pZDDs 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

Table 2 Comparison of MTBDDs and pZDDs with the UMTS model

			Nodes		Savings	Run-time in sec.		Savings	
n	k	States	MTBDD	pZDD	in %	MTBDD	pZDD	in %	E[π]
8	6	2529	18274	12437	31.94	54	35	35.19	44
8	7	12673	35303	24183	31.50	91	49	46.15	30
8	8	395	8703	5615	35.48	12	10	16.67	20
10	6	3941	31888	23823	25.29	198	109	44.95	111
10	7	19761	66085	45873	30.58	472	243	48.52	72
10	8	6931	43466	31134	28.37	185	100	45.95	50
12	6	833	20415	13638	33.20	486	277	43.00	289
12	7	28417	101605	64910	36.12	1892	984	47.99	178
12	8	2815	42024	28462	32.27	358	208	41.90	116

of pZDDs 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 pZDDs instead of MTBDDs, performance measures for the FMS model and a scaling parameter of $N = 12$ (resulting in more than 111 million states) can be computed in ≈ 4 h instead of ≈ 12 h (for further details please refer to [13]).

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 phone or Internet access). The model also identifies whether 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, which is based on the PRISM input language and the JINC package.

For probabilistic model checking, a symbolic representation of the transition matrix is derived directly 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. Since PROMOC uses a fully symbolic linear solver (which is far less efficient than the hybrid solution methods mentioned at the end of Sect. 5.2), our experiments were limited to relatively small values of n and k. Nevertheless, the benchmark results in Table 2 show that pZDDs clearly outperform regular MTBDDs in both size and run-time, where the run-time advantage stems from the fact that fewer DD-nodes induce fewer recursive calls of the DD-manipulating algorithms as well as of the symbolic numerical solution algorithm.

6 Conclusion

In this paper we extended ZBDDs [15] 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 respective algorithms for manipulating them. This not only allowed us to implement pZDDs within standard shared DD-environments, such as CUDD [26] or JINC [9], but also supports the application of non-zero-preserving operators to them. The efficiency of the introduced approach was then demonstrated by analyzing case studies from different contexts, 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. Furthermore, the size reduction of the symbolic structures also leads to run-time advantages.

References

1. Formal Methods in System Design (1997) 10(2–3). Special Issue on Multi-Terminal Binary Decision Diagrams
2. Akers SB (1978) Binary decision diagrams. IEEE Trans Comput C-27(6):509–516
3. Balbo G, Conte G, Donatelli S, Franceschinis G, Ajmone Marsan M, Ajmone Marsan M (1995) Modelling with generalized stochastic Petri nets. Wiley, New York
4. Bryant RE (1986) Graph-based algorithms for Boolean function manipulation. IEEE Trans Comput C-35(8):677–691
5. Ciardo G, Lüttgen G, Miner AS (2007) Exploiting interleaving semantics in symbolic state-space generation. Form Methods Syst Des 31(1):63–100
6. de Alfaro L, Kwiatkowska M, Norman G, Parker D, Segala R (2000) Symbolic model checking for probabilistic processes using MTBDDs and the Kronecker representation. In: Graf S, Schwartzbach M (eds) Proc. of the 6th int. conference on tools and algorithms for the construction and analysis of systems (TACAS'00). LNCS, vol 1785. Springer, Berlin, pp 395–410
7. Hermanns H, Herzog U, Mertsiotakis V (1998) Stochastic process algebras—between LOTOS and Markov chains. Comput Netw ISDN Syst 30(9–10):901–924
8. Hermanns H, Kwiatkowska M, Norman G, Parker D, Siegle M (2003) On the use of MTBDDs for performability analysis and verification of stochastic systems. J Log Algebr Program 56(1–2):23–67
9. JINC BDD package. www.jossowski.de
10. Kam T, Villa T, Brayton R, Sangiovanni-Vincentelli A (1998) Multi-valued decision diagrams: theory and applications. Mult Valued Log 4(1–2):9–62
11. Kuntz M, Siegle M, Werner E (2004) Symbolic performance and dependability evaluation with the tool CASPA. In: Proc. of EPEW. LNCS, vol 3236. Springer, Berlin, pp 293–307
12. Lampka K, Siegle M (2006) Activity-local state graph generation for high-level stochastic models. In: Meassuring, modeling, and evaluation of systems 2006, April 2006, pp 245–264
13. Lampka K, Siegle M (2006) Analysis of Markov reward models using zero-supressed multi-terminal decision diagrams. In: Proceedings of VALUETOOLS 2006 (CD-edition), October 2006
14. Lee CY (1959) Representation of switching circuits by binary-decision programs. Bell Syst Tech J 38:985–999
15. Minato S (1993) Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Proc. of the 30th ACM/IEEE design automation conference (DAC), Dallas (Texas), USA, June 1993, pp 272–277
16. Minato S (2001) Zero-suppressed BDDs and their applications. Int J Softw Tools Technol Transf 3(2):156–170
17. Miner A, Parker D (2004) Symbolic representations and analysis of large state spaces. In: Baier Ch, Haverkort B, Hermanns H, Katoen J-P, Siegle M (eds) Validation of stochastic systems, Dagstuhl (Germany), 2004. LNCS, vol 2925. Springer, Berlin, pp 296–338
18. Möbius web page. www.moebius.crhc.uiuc.edu
19. Ossowski J, Baier C (2008) A uniform framework for weighted decision diagrams and its implementation. Int J Softw Tools Technol Transf 10(5):425–441
20. PRISM. www.prismmodelchecker.org

21. PROMOC modeling tool. www.jossowski.de
22. Sasao T, Fujita M (eds) (1996) Representations of discrete functions, vol 1. Kluwer Academic, Dordrecht
23. Shannon CS (2000) Eine symbolische Analyse von Relaisschaltkreisen. In: Ein/Aus. Brinkmann und Bose, Berlin. The article originally appeared with the title: A symbolic analysis of switching circuits in Trans. AIEE 57 (1938), 713
24. Siegle M (2001) Advances in model representation. In: de Alfaro L, Gilmore S (eds) Proc. of the joint int. workshop, PAPM-PROBMIV 2001, Aachen (Germany). LNCS, vol 2165. Springer, Berlin, pp 1–22
25. SMART. www.cs.ucr.edu/~ciardo/SMART
26. Somenzi F (1998) CUDD: Colorado University decision diagram package release
27. Wegener I (2000) Branching programs and binary decision diagrams. SIAM, Philadelphia