JINC – A Multi-Threaded Library for Higher-Order Weighted Decision Diagram Manipulation

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Jörn Ossowski

aus

Düsseldorf

Bonn 2009

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

Diese Dissertation ist auf dem Hochschulschriftenserver der ULB Bonn
`http://hss.ulb.uni-bonn.de/diss_online`
elektronisch publiziert.

1. Referent: Prof. Dr. Armin B. Cremers
2. Referent: Prof. Dr. Wolfgang Küchlin
Tag der Promotion: 1.10.2010
Erscheinungsjahr: 2010

JINC – A Multi-Threaded Library for Higher-Order Weighted Decision Diagram Manipulation

Jörn Ossowski

Abstract

Ordered Binary Decision Diagrams (OBDDs) have been proven to be an efficient data structure for symbolic algorithms. The efficiency of the symbolic methods depends on the underlying OBDD library. Available OBDD libraries are based on the standard concepts and so far only differ in implementation details. This thesis introduces new techniques to increase run-time and space-efficiency of an OBDD library.

This thesis introduces the framework of Higher-Order Weighted Decision Diagrams (HOWDDs) to combine the similarities of different OBDD variants. This framework pioneers the basis for the new variant Toggling Algebraic Decision Diagrams (TADDs) which has been shown to be a space-efficient HOWDD variant for symbolic matrix representation. The concept of HOWDDs has been use to implement the OBDD library JINC. This thesis also analyzes the usage of multi-threading techniques to speed-up OBDD manipulations. A new reordering framework applies the advantages of multi-threading techniques to reordering algorithms. This approach uses an abstraction layer so that the original reordering algorithms are not touched. The challenge that arise from a straight forward algorithm is that the computed-tables and the garbage collection are not as efficient as in a single-threaded environment. We resolve this problem by developing a new multi-operand `APPLY` algorithm that eliminates the creation of temporary nodes which could occur during computation and thus reduces the need for caching or garbage collection.

The HOWDD framework leads to an efficient library design which has been shown to be more efficient than the established OBDD library CUDD. The HOWDD instance TADD reduces the needed number of nodes by factor two compared to ordinary ADDs. The new multi-threading approaches are more efficient than single-threading approaches by several factors. In the case of the new reordering framework the speed-up almost equals the theoretical optimal speed-up. The novel multi-operand `APPLY` algorithm reduces the memory usage for the n-queens problem by factor 50 which enables the calculation of bigger problem instances compared to the traditional `APPLY` approach.

The new approaches improve the performance and reduce the memory footprint. This leads to the conclusion that applications should be reviewed whether they could benefit from the new multi-threading multi-operand approaches introduced and discussed in this thesis.

Keywords: OBDD, TADD, HOWDD, multi-threading, reordering framework, multi-operand APPLY, JINC

Danksagung / Acknowledgements

This thesis has gone a long way. There are many people who helped to make this work what it is today.

I would like to thank my supervisor Prof. Dr. Cremers for the encouragement to explore the potential of multi-threading architectures, Prof. Dr. Küchlin for his helpfulness by being the second examiner on short notice, Prof. Dr. Weber for his endless support and guidance and Prof. Dr. Koepke for his aid.

I owe my special thanks to Daniel Frey for this assistance in all C++ and performance related questions and his very helpful review comments. My thanks also go to Dr. Christoph Vogelbusch, Tobias Blechmann and Dr. Rolf Bardeli for reviewing this thesis.

Lastly, and most importantly, I would like to express my heart-felt gratitude to my wife Claudia and my family.

This thesis is dedicated in loving memory to

my father Hartmut Martin Ossowski.

Contents

Chapter 1 Introduction

Since the introduction of Ordered Binary Decision Diagrams (OBDDs) [21], a still increasing number of applications [59, 54, 44, 57, 23, 22, 120, 75] are adopting symbolic algorithms (based on OBDDs) to solve problems in symbolic model checking, computer-aided design (CAD), very large scale integration (VLSI), etc.

From the theoretical point of view no data structure can avoid an exponential-sized representation for certain functions. Real world applications (combinatory logic, model checking, multi-level logic, etc.) have shown that wide classes of commonly used functions can be represented by OBDDs (or their variants) very efficiently. In the context of these applications various OBDD variants have been developed to enable space-efficient representation of the different systems, e.g., in arithmetic applications FEVBDDs [114] lead to smaller sized representations than regular OBDDs.

Besides the theoretical advantages of different OBDD variants it is essential for real-world applications to have an efficient OBDD library which provides support for different variants. This thesis discusses new theoretical concepts for the implementation of the space- and run-time-efficient OBDD library JINC. Space-efficiency is JINC'S key requirement because it directly influences the number of representable systems. The purpose of this thesis is two-fold.

At first we provide a summary of several known OBDD variants. The similarities of different variants are examined and yield the basis for Output Weighted Decision Diagrams (OWDDs) [87]. OWDDs allow transformations on the terminal values and thus cover variants like OBDDs with negated edges [102], Edge-Valued Binary Decision Diagram (EVBDD) [119], Factored Edge-Valued Binary Decision Diagrams (FEVBDDs) [114], etc. To allow augmentation of the the edges with further transformation functions like input transformations we expand the definition of OWDDs to Higher-Order Weighted Decision Diagrams (HOWDDs). The advantage of this newly developed framework is that it handles any numbers of successors (like Multi-Valued Decision Diagrams (MDDs) [60]) and allows to augment the edges with a wide range of transformation functions. HOWDDs identify the minimal set of requirements needed to define a generic OBDD library. Based on these requirements we will discuss the implementation of the OBDD library JINC. The design of JINC follows the general concept of [19] and adjusts this by a more general approach, which is influenced by the idea of HOWDDs. With the new need to support arbitrary transformation functions it is necessary to utilize modern programming techniques to provide a flexible, efficient, safe and easy to maintain implementation. The differences to already existing OBDD library implementations (CUDD [109], BuDDy [72], etc) will be outlined while discussing JINC's architecture. A more detailed examination of JINC's design will be illustrated on the implementation of the newly developed Toggling Algebraic Decision Diagrams (TADDs). The intended application for TADDs is matrix representation and manipulation. The OBDD representation of a matrix encodes the coordinates of an element by x variables (for the row) and y variables (for the column). It has been shown that an interleaved variable ordering is best suited for matrix operations. The idea behind TADDs is to combine these variable pairs to one variable to provide a compact representation. Additionally, a transformation function is used so that the transposition of a matrix can be performed in constant time.

The second and main part of this thesis presents and discusses new concepts for OBDD libraries. Besides space-efficiency, run-time-efficiency is important. This thesis introduces new approaches to increase run-time efficiency and to reduce memory usage. There have been several approaches to enhance performance. The most promising is to use more than one processor. There have been several approaches to enable parallel computation for OBDD packages [111, 123]. The drawback of these solutions is that they implement parallel computation with distinct memory areas and communicate progress through a message passing interface (MPI). [111, 78] illustrated that due to the communication costs single-processor OBDD library are more efficient than multi-processor OBDD libraries. This thesis introduces a novel approach to benefit from the increasing number of multi-processor architectures [112] to speed-up OBDD manipulations. JINC makes use of multi-threading architectures so that the benefit from accessing shared memory areas results in no communication overhead and thus results in better run-times compared to single-threaded implementations. The approach of using shared memory does not change the depth-first traversal character of BDD algorithms. This is an important advantage over multi-processor implementations as they distribute the task in a breadth-first manner. The problems occurring with concurrency within shared memory is solved by modifying the existing non-thread-safe data structures to non-locking thread-safe data structures. For efficiency reasons it is required to avoid mutex usage wherever possible.

The advantage of using distinct memory areas is that more memory can be used compared to a single computer OBDD library. New memory can be appended to increase the available memory by adding a new computer to the system. The usage of distinct memory areas has its limits as the communication costs grow quadratically. JINC's multi-threading approach focuses on increasing the performance but not on increasing the available memory. This thesis introduces a novel way to face the challenge of increasing the number of representable systems within the boundaries of physical memory. The idea is based on two observations. First, frequent operator calls increase the number of temporary nodes which are needed to compute the overall result. Second, JINC's multi-threading approach disabled automatic garbage collection and automatic reordering because of concurrency. This leads to the necessity to reduce the number of temporary nodes during computations as garbage collection calls need synchronization points in a multi-threaded environment. The new idea presented in this thesis combines any number of operator calls to one multi-operand `APPLY` call. This novel multi-operand synthesis approach is superior to [51, 27] as it does not change the depth-first synthesis of the ordinary `APPLY` algorithm, is suitable for multi-threading and is optimal in terms of memory requirements as it does not create any temporary node.

This thesis also provides a novel reordering system. This system speeds up the reordering phase because it also benefits from multi-processor architectures without increasing the memory footprint. The reordering system is developed in such a way that there is no need to adjust the already existing reordering algorithms. Existing parallel reordering systems, like e.g., [81], take advantage of empirical observations and distribute different instances of a reordering algorithm over a number of computers. This approach does not lead to a significant run-time improvement because it runs unmodified reordering algorithms on different computers. JINC's approach delays the swap operations until needed and identifies parallel swaps. The identification runs in a separate thread and distributes the swap operations to different threads. These swap operations run in parallel so that this approach increases the run-time performance.

Due to the novel approaches and the modern structure we show in many competitive benchmarks that the theoretical speed improvements prove themselves in real world examples, making JINC to a space and run-time efficient OBDD library. The result can be summarized as follows:

- The HOWDD framework leads to an efficient single-threaded OBDD library design.
- TADDs are superior to ordinary ADDs. The number of nodes is reduced by almost factor two. The speed-up is around 150%.
- The multi-threaded version of JINC increases the performance by several factors without increasing the memory usage.
- The novel multi-operand `APPLY` algorithm increases the performance while eliminating the creation of temporary nodes. This enables the computation of larger models.

Chapter 2
Ordered Binary Decision Diagrams

This chapter reminds the main concepts of Ordered Binary Decision Diagrams (OBDDs) and introduces definitions and notations which are necessary for the development of a generic framework which yields the basis for the OBDD library JINC.

Binary Decision Diagrams (BDDs) created by Lee [68] and Akers [3] provide an efficient method to represent Boolean functions. Bryant [21] introduced Ordered Binary Decision Diagrams (OBDDs) which are BDDs with a fixed variable order to assure a canonical form. With this property, the equality of functions can be checked in constant time. The restriction of a fixed variable ordering enabled Bryant to develop efficient manipulation algorithms for OBDDs.

2.1 Notations and Definitions

This section will give basic definitions and notations for OBDDs and their variants.

Definition 2.1.1 (Evaluation)**.** Let $\mathcal{Z} = \{z_1, \ldots, z_n\}$ be a finite set of Boolean variables. An *evaluation* of $\mathcal{Z}$ is a map

$$\eta : \mathcal{Z} \to \{0, 1\}$$

that assigns a value $\eta(z) \in \{0, 1\}$ to each variable $z \in \mathcal{Z}$.
$\mathit{Eval}(\mathcal{Z})$ identifies the set of all evaluations of $\mathcal{Z}$.

Let $\overline{a} = (a_1, \ldots, a_n) \in \{0, 1\}^n$ and $\overline{z} = (z_{i_1}, \ldots, z_{i_n}) \in \mathcal{Z}^n$ with pairwise different z_{i_j}, then $[\overline{z} = \overline{a}]$ represents the evaluation $\eta \in \mathit{Eval}(\mathcal{Z})$ with

$$\eta(z_{i_j}) = a_j, \ j = 1, \ldots, n.$$

□

Notation 2.1.2 (Assignment)**.** Let $\mathcal{Z}$ be defined as in Definition 2.1.1, $\overline{a} = (a_1, \ldots, a_r) \in \{0, 1\}^r$ and $\overline{z} = (z_{i_1}, \ldots, z_{i_r}) \in \mathcal{Z}^r$ with pairwise different z_{i_j}. The *assignment*

$$\eta\left[\overline{z} = \overline{a}\right] \in \mathit{Eval}(\mathcal{Z})$$

is defined by

$$\eta\left[\overline{z} = \overline{a}\right](z) = \begin{cases} a_j & \text{if } z \in \{z_{i_1}, \ldots, z_{i_r}\} \text{ with } z = z_{i_j} \\ \eta(z) & \text{otherwise.} \end{cases}$$

Definition 2.1.3 ($\mathbb{K}$-function)**.** Let $\mathbb{K}$ be a set. A $\mathbb{K}$*-function* over $\mathcal{Z}$ is a map

$$f : \mathit{Eval}(\mathcal{Z}) \to \mathbb{K}.$$

The set of all Boolean functions over $\mathcal{Z} = \{z_1, \ldots, z_n\}$ will be called $I\!\!K(\mathcal{Z})$ or $I\!\!K(z_1, \ldots, z_n)$. The special cases $I\!\!K = \{0, 1\}$ and $I\!\!K = I\!\!R$ identify the *switching functions* and real-valued functions. The set of all $I\!\!K$-functions, switching functions and real-valued functions will be called $\mathbb{B}(\mathcal{Z})$ and $I\!\!R(\mathcal{Z})$, respectively.

□

Definition 2.1.4 (Cofactor)**.** Let $\overline{z}$ and $\overline{a}$ be as in Notation 2.1.2 and $f \in I\!\!K(\mathcal{Z})$. The *cofactor* of f related to $\overline{z}$ is defined by:

$$f|_{\overline{z}=\overline{a}} \in I\!\!K(\mathcal{Z})$$

where

$$f|_{\overline{z}=\overline{a}}(\eta) = f(\eta\,[\overline{z} = \overline{a}]).$$

□

Definition 2.1.5 (Composition Operator)**.** Let $\overline{z} = (z_{i_1}, \ldots, z_{i_r}) \in \mathcal{Z}^r$ with pairwise different z_{i_j}, $\overline{g} = (g_1, \ldots g_r) \in I\!\!K(\mathcal{Z})^r$ and $f \in I\!\!K(\mathcal{Z})$. The composition operator $\{\overline{z}/\overline{g}\}$ is defined by:

$$f\{\overline{z}/\overline{g}\} \in I\!\!K(\mathcal{Z})$$

where

$$f\{\overline{z}/\overline{g}\}(\eta) = f(\eta')$$

with

$$\eta'(x) = \begin{cases} g_j(\eta(x)) & \text{if } x = z_{i_j} \\ \eta(x) & \text{otherwise.} \end{cases}$$

□

Definition 2.1.6 (Rename Operator)**.** Let $\overline{x} = (x_{i_1}, \ldots, x_{i_r}) \in \mathcal{Z}^r$ with pairwise different x_{i_j}, $\overline{z} = (z_{i_1}, \ldots, z_{i_r}) \in \mathcal{Z}^r$ with pairwise different z_{i_j} and $f \in I\!\!K(\mathcal{Z})$. The rename operator $\{\overline{x} \leftarrow \overline{z}\}$ is defined by:

$$f\{\overline{x} \leftarrow \overline{z}\} \in I\!\!K(\mathcal{Z})$$

where

$$f\{\overline{x} \leftarrow \overline{z}\} = f\{\overline{x}/\overline{\hat{z}}\}$$

with

$$\overline{\hat{z}} = (\hat{z_{i_1}}, \ldots, \hat{z_{i_r}}) \text{ and } \hat{z_{i_j}} = \text{projection function of } z_{i_j}.$$

□

Definition 2.1.7 (Variable Ordering)**.** A *variable ordering* over $\mathcal{Z}$ is an ordered tuple

$$\pi = (z_{i_1}, \ldots, z_{i_n}),$$

that contains every variable $z_{i_j} \in \mathcal{Z}$ exactly once. A variable ordering π defines an order relation over variables in a canonical way. For every two variables $z_{i_j}, z_{i_k} \in \pi$ the following holds:

$$z_{i_j} <_\pi z_{i_k} \Leftrightarrow j < k.$$

□

2.2 Shared Ordered Binary Decision Diagrams

Ordered Binary Decision Diagrams have been introduced by Bryant [21]. Bryant also introduced an efficient algorithm to reduce an OBDD. This Reduced OBDDs (ROBDDs) are of canonical form. The canonical form can be obtained in several ways. Regarding the implementation of an efficient OBDD library it is not reasonable to apply the reduction algorithm on an OBDD after every manipulation. The reduction algorithm is therefore integrated into the manipulation algorithms. The consequence for OBDDs would be to hold isomorphic tables for every OBDD instance. A further disadvantage of storing OBDDs in disjunctive memory areas is that subgraph-sharing cannot apply. For that reason Minato [102, 115] introduced Shared Ordered Binary Decision Diagrams (SOBDDs). The idea is to use one memory area for all OBDDs. With that approach it is possible to integrate the reduction algorithm into the manipulation operations in an efficient manner. This also allows to check for equality of two functions in constant time. In this thesis we will focus on the idea of SOBDDs as a basis for implementing JINC (see Chapter 4 for more details).

The disadvantage of the idea of storing all functions in one memory area is that the fixed variable ordering for all functions could lead to an exponential-sized representation whereas storing the functions in several OBDDs with different orderings would be of linear size. It has been proven that these “bad” functions rarely occur in real applications. In practice, the advantages of faster algorithms[1] and an equivalence test in constant time outweigh the disadvantages so that we will focus on the idea of Shared Ordered BDDs.

In the following sections we want to show the need for a general framework for OBDDs with output transformations by means of different OBDD variants. It will be assumed that the following variants are reduced and canonical. We will recall the basic ideas and concepts for every variant instead of discussing every detail. The formal definitions of reducedness and uniqueness are discussed and proven once for Output Weighted Decision Diagrams (OWDDs) and assigned to every variant thereafter.

2.2.1 SOBDD with Negative Edges

The idea of sharing all functions in one OBDD leads to a problem when negating a given function represented in a SOBDD. In the case of OBDDs the values of the drains have to be negated to negate the represented function (see Figure 2.1). This is not possible for SOBDDs because changing the values of the drains would negate all represented functions.

[1] Algorithms operating on OBDDs with the same variable ordering are generally of lower complexity than algorithms operating on OBDDs with different orderings. This does not mean that all algorithms on SOBDDs are “faster” than algorithms on OBDDs. In Section 2.2.1 it will be explained how to avoid the problem of negating a function in a SOBDD – compared to the negation of an OBDD in constant time.

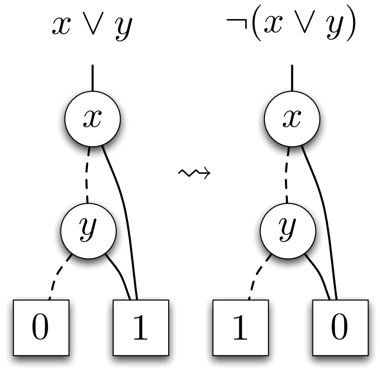


Figure 2.1: Negation on OBDDs

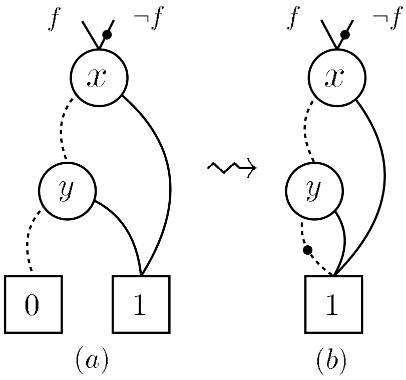


Figure 2.2: The idea of negative edges

To overcome this drawback, attributed edges [102] were introduced. This means that additional information can be stored on the edges. In the above example negative edges are being used. Figure 2.2 illustrates the idea of negative edges. The dot on an edge means that the function represented by the succeeding node must be negated. Instead of changing the values of the drains the edges can now be altered to negate the represented function. The OBDD shown in Figure 2.2 (a) is not reduced. To reduce the OBDD further, the idea of negative edges must be applied to all nodes (see (b)) which results in a smaller sized BDD.

In view to implementation it means that all data structures must be able to handle transformation functions. Chapter 4 discusses different approaches to handle different classes of transformation functions.

2.2.2 Algebraic Decision Diagrams

The above mentioned OBDD variants are used to represent switching functions. The following OBDD variants are used for algebraic computation, i.e., real-valued functions.

The former definition of SOBDDs (without negative edges) can be expanded to real-valued functions so that real values are possible for the drains. Figure 2.3 shows the

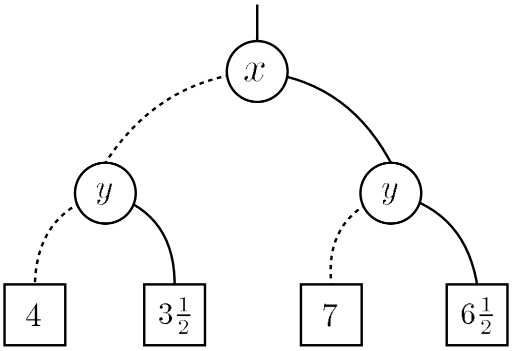


Figure 2.3: ADD of $3x - \frac{1}{2}y + 4$

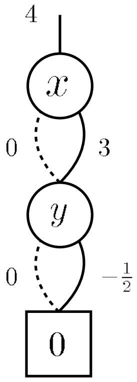


Figure 2.4: EVBDD of $3x - \frac{1}{2}y + 4$

representation of the function $3x - \frac{1}{2}y + 4$. This OBDD variant is called Algebraic Decision Diagram (ADD) or Multi-Terminal BDD (MTBDD) [7, 30].

2.2.3 Edge-Valued Binary Decision Diagrams

The disadvantage of ADDs is that the size of the OBDD representation depends on the number of different function values. Therefore a similar approach as negative edges is used. Instead of using a negative point on the edges an additive value is used. Figure 2.4 shows an example for function $3x - \frac{1}{2}y + 4$. To calculate the value of the Edge-Valued Binary Decision Diagram (EVBDD) [119], all the edge-values on the way to the drain have to be added. To ensure a canonical form the edges of an EVBDD are restricted in a way so that only the one-successor can have a non-trivial weight and the zero-successor weight is fixed to zero. With this restriction only one weight has to be stored and so this approach leads to a more compact implementation.

Another important advantage of EVBDDs over ADDs is that the addition of a

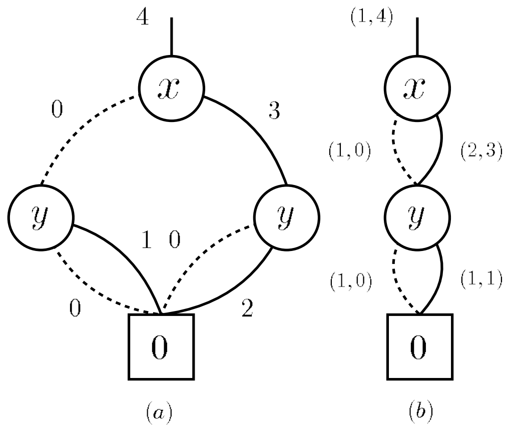


Figure 2.5: (a) EVBDD and (b) FEVBDD of $3x + xy + y + 4$

constant value to a function can be computed in constant time. In the case of ADDs this could be done by changing the values of the drain, but that would lead to the same problems as in the case of SOBDD. The edge attributes of EVBDDs cannot be handled as efficient as negated edges (see Section 4.1.1.1 for more details).

2.2.4 Factored Edge-Valued Binary Decision Diagrams

The idea of EVBDDs can be expanded further to Factored Edge-Valued Binary Decision Diagrams (FEVBDDs) [114]. Instead of just adding a value to the succeeding represented function a multiplicative and an additive weight are used. This leads to a more compact representation than EVBDDs and enables the multiplication with a constant value in constant time. Figure 2.5 shows the EVBDD and FEVBDD for the function $f(x, y) = 3x + xy + y + 4$. The first number in brackets is the multiplicative weight and the second the additive. The cofactor $f|_{[x=1,y=0]}$ is calculated with $1 \cdot (2 \cdot (1 \cdot 0 + 0) + 3) + 4 = 7$. As in the case of EVBDDs the edge weights of FEVBDDs are restricted so that only the one-successor can have a non-trivial weight and the zero-successor weight is fixed to $(1, 0)$.

2.2.4.1 Normalized Algebraic Decision Diagrams

The restrictions of the weights on FEVBDDs[2] reduces the number of parameters that have to be considered. With this restriction the calculation of minima and maxima needs a full traversing. The idea of Normalized Algebraic Decision Diagrams (NADDs) [84, 87] is that in addition to the previously mentioned advantages of FEVBDDs (multiplication and addition of a constant value in constant time) the calculation of minima and maxima can be calculated in constant time. Instead of fixing the weights, the functions that are represented by each inner node v is normalized, i.e., $f_v : \{0, 1\}^n \rightarrow [0, 1]$, $\min f_v = 0$ and $\max f_v = 1$. With this

[2] the parameters of the zero-successor are fixed

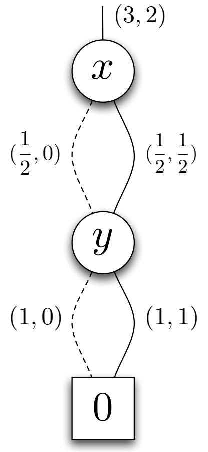
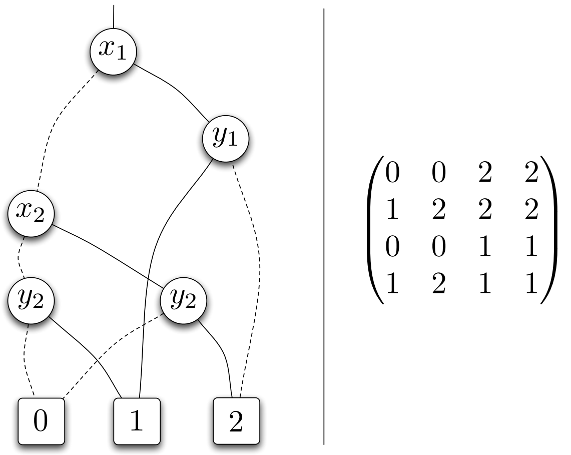


Figure 2.6: NADD representing $\frac{3}{2}x + \frac{3}{2}y + 2$



$$\begin{pmatrix} 0 & 0 & 2 & 2 \\ 1 & 2 & 2 & 2 \\ 0 & 0 & 1 & 1 \\ 1 & 2 & 1 & 1 \end{pmatrix}$$

Figure 2.7: Matrix representation with ADDs

property the minimum and maximum of a NADD function can be calculated using only the parameters pointing to that node. The disadvantage of that method is that due to more complex calculations the run-time increases for normal operations. Figure 2.6 shows an example NADD representing the function $\frac{3}{2}x + \frac{3}{2}y + 2$.

2.3 Matrix Representation and Operations

Key features of an OBDD library are the provided operations. In this section we will recall the idea of matrix representation with OBDD variants. These operations yield the basis for the benchmarks provided in Chapter 7. Chapter 4 illustrates the implementation details for matrix operations.

Matrices can be expressed with OBDD variants as described in [30, 43] (Figure 2.7 illustrates the idea). The representation of a vector can be seen as the representation of a matrix with dimension $n \times 1$.

$$Ax = \boxed{A} \cdot \boxed{x} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

$$= \begin{bmatrix} A_{00} & A_{01} \\ \multicolumn{2}{c}{0} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} + \begin{bmatrix} \multicolumn{2}{c}{0} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

$$= \begin{bmatrix} \boxed{A_{00}} \cdot \boxed{x_0} + \boxed{A_{01}} \cdot \boxed{x_1} \\ \boxed{A_{10}} \cdot \boxed{x_0} + \boxed{A_{11}} \cdot \boxed{x_1} \end{bmatrix}$$

Figure 2.8: Block-wise matrix vector multiplication

We will now discuss methods to solve linear systems of equations that are well suited for OBDDs. Methods that need arbitrary access to an element of a matrix cannot be used efficiently with OBDDs. The recursive structure of OBDDs supports very efficient block-wise access to the elements [43]. In this thesis we just focus on pure OBDD approaches as we want to see how the different variants perform. For more details on a hybrid approach combining OBDDs and the sparse matrix representation see [92].

2.3.1 Direct methods

Direct methods for solving a linear equation system like the Gauss' method [32] cannot be used efficiently with OBDDs because they need arbitrary access to the elements of a matrix. Additionally, the complexity of such methods is higher than the complexity of iterative methods. Another reason not to use these methods on OBDDs is that these methods are more vulnerable to arithmetic errors.

2.3.2 Iterative methods

Iterative methods have been established for all kinds of applications. We will only focus on stationary iterative methods [10], i.e., the same operation is used for all iterative steps. As mentioned above, block-wise access is well supported by OBDD structures. With an interleaved variable ordering [42] the calculation of the substructures (blocks) can be performed in constant time. Figure 2.8 illustrates the idea of block-building on the example of matrix-vector multiplication.

With this approach the matrix-vector multiplication can be performed efficiently with OBDDs.

The idea of an iterative method is based on the following observation.

Let $A = M + N$ with appropriate matrices M and N where M is invertible.

$$\begin{aligned} Ax = b &\Leftrightarrow (M+N)x = b && (2.1)\\ &\Leftrightarrow Mx + Nx = b && (2.2)\\ &\Leftrightarrow Mx = b - Nx && (2.3)\\ &\Leftrightarrow x = M^{-1}(b - Nx) && (2.4)\\ &\Leftrightarrow x = M^{-1}b - M^{-1}Nx && (2.5)\end{aligned}$$

This leads to the iterative instruction:

$$x^{(n+1)} = M^{-1}b - M^{-1}Nx^{(n)}$$

If M is the diagonal matrix of A, this method is called Jacobi method. The condition that M has to be invertible means that no zero element exists on the diagonal.

Chapter 3
Higher-Order Weighted Decision Diagrams

In the preceding chapter we recalled the main concepts of several OBDD variants. The basic concept of OBDDs holds for all variants with just a few small modifications. The definition of reducedness and the proof of canonicity is similar for every variant. For this reason we introduce the general framework of Higher-Order Weighted Decision Diagrams (HOWDDs). A generic framework that covers several well established OBDD variants also supports the design of an efficient OBDD library. The observations about similarities of several OBDD variants influenced JINC's design.

3.1 Output Weighted Decision Diagrams

In this section we introduce the general framework of Output Weighted Decision Diagrams (OWDDs) that covers all above mentioned variants to reason about reducedness and canonicity.

Parts of the material of this chapter has been published in [87] (where we used the notion WDD rather than OWDD).

3.1.1 Definition and Semantics

Notation 3.1.1 (The Φ sets)**.** In the sequel, let $\mathcal{Z}$ be a finite set of variables, $I\!K$ a set with at least two elements[1], and let $I\!F$ denote the set of functions $f : \mathit{Eval}(\mathcal{Z}) \to I\!K$. $\Phi_{I\!F}$ denotes a nonempty set of bijections $I\!K \to I\!K$ such that

(1) $\Phi_{I\!F}$ is closed under inversion and composition, i.e., if $\varphi, \psi \in \Phi_{I\!F}$ then $\varphi^{-1} \in \Phi_{I\!F}$ and $\varphi \circ \psi \in \Phi_{I\!F}$. (In particular, $\Phi_{I\!F}$ contains the identity id.)

(2) If $f \in I\!F$, $\varphi \in \Phi_{I\!F}$ and $f = \varphi \circ f$ then $\varphi = id$.

In our notion of output weighted decision diagrams the edges will be augmented with functions $\varphi \in \Phi_{I\!F}$ that serve as transformations. The idea is that any φ-labelled edge to (a node for) a function f stands for the function $\varphi \circ f$. Condition (2) will be important for the uniqueness of the function representation. Note that if $\varphi, \psi \in \Phi_{I\!F}$, $f \in I\!F$ and $\varphi \circ f = \psi \circ f$ then $\varphi = \psi$ (as we have $f = (\varphi^{-1} \circ \psi) \circ f$, and hence, $\varphi^{-1} \circ \psi = id$ by (2)).

In some BDD-variants with weighted edges, the constant functions (represented by the incoming edges of the terminal nodes) require special treatment as there might be several possibilities for transforming a constant $c \in I\!K$ into another constant

[1] The requirement $I\!K$ to be a semi-ring as in [8] could be added, but it is irrelevant as long as we do not discuss operations on $I\!K$.

$c' \in I\!\!K$ via the bijections $\varphi \in \Phi_{I\!\!F}$. Therefore, we split $I\!\!F$ into the sets $I\!\!F^{const}$ and $I\!\!F^{non\text{-}const}$ of constant and non-constant functions in $I\!\!F$, respectively, and assume that we are given sets of transformations $\Phi = \Phi_{I\!\!F^{non\text{-}const}}$ for the non-constant functions in $I\!\!F$ and Φ^{const} for the constant functions in $I\!\!F$, such that Φ and Φ^{const} fulfill conditions (1) and (2) in Notation 3.1.1.

Definition 3.1.2 (Output Weighted Decision Diagram (OWDD))**.** Let $\mathcal{Z}$, $I\!\!K$, Φ, Φ^{const} be as above and π a variable ordering for $\mathcal{Z}$. A π-OWDD for $(\mathcal{Z}, I\!\!K, \Phi, \Phi^{const})$ is a rooted, binary branching, acyclic graph $\mathcal{B}$ with several additional information. For the inner nodes, we have

- a function *var* that assigns a variable $var(v) \in \mathcal{Z}$ to any inner node v,
- functions $v \mapsto succ_0(v)$ and $v \mapsto succ_1(v)$ that specify the successors of v,
- functions $v \mapsto \phi_0(v)$ and $v \mapsto \phi_1(v)$ that specify the transformations associated with the outgoing edges from v.

For the terminal nodes, we have a function $v \mapsto value(v) \in I\!\!K$. If v is an inner node and $\xi \in \{0,1\}$ such that $succ_\xi(v)$ is an inner node then we require that $var(v)$ occurs in π before $var(succ_\xi(v))$ and $\phi_\xi(v) \in \Phi$. If $succ_\xi(v)$ is a terminal node then we require $\phi_\xi(v) \in \Phi^{const}$.
The root of $\mathcal{B}$ is a pair $r = \langle \phi_r, v_r \rangle$ consisting of a function $\phi_r \in \Phi \cup \Phi^{const}$ and a node v_r from which all other nodes in $\mathcal{B}$ are reachable. As for the edges we require $\phi_r \in \Phi$ if v_r is an inner node and $\phi_r \in \Phi^{const}$ if v_r is terminal[2]. □

Notation 3.1.3 (Semantics of OWDDs)**.** The semantics of a OWDD $\mathcal{B}$ is formalized by associating a function

$$f_v \in I\!\!F = I\!\!F^{const} \cup I\!\!F^{non\text{-}const}$$

to any node in $\mathcal{B}$ and a function for the root r. Intuitively, an incoming edge of node v labelled with φ stands for the function $\varphi \circ f_v$. Formally, the function $f_{\mathcal{B}}$ for OWDD $\mathcal{B}$ is induced by its root $r = \langle \phi_r, v_r \rangle$ which is given by $f_{\mathcal{B}} = \phi_r \circ f_{v_r}$ where the function f_v for the nodes is defined in a bottom-up fashion. (The level of a z-node denotes the position of variable z in π. The nodes on the bottom-level are the terminal nodes.)

The terminal nodes represent constant functions as expected, i.e., $f_v = value(v)$ for any terminal node v. If v is a z-node then $f_v : Eval(\mathcal{Z}) \to I\!\!K$ is defined as follows: Let $\eta \in Eval(\mathcal{Z})$ and $\eta(z) = \xi \in \{0,1\}$ then

$$f_v(\eta) = \phi_\xi(v) \circ f_{succ_\xi(v)}(\eta).$$

If $I\!\!K = \{0,1\}$ we may write f_v as

$$f_v \;=\; (\neg z \wedge \phi_0(v) \circ f_{succ_0(v)}) \;\vee\; (z \wedge \phi_1(v) \circ f_{succ_1(v)})$$

and if $I\!\!K = I\!\!R$

$$f_v \;=\; (1 - z) \cdot (\phi_0(v) \circ f_{succ_0(v)}) \;+\; z \cdot (\phi_1(v) \circ f_{succ_1(v)}).$$

[2] In this case, v_r is the only node in $\mathcal{B}$.

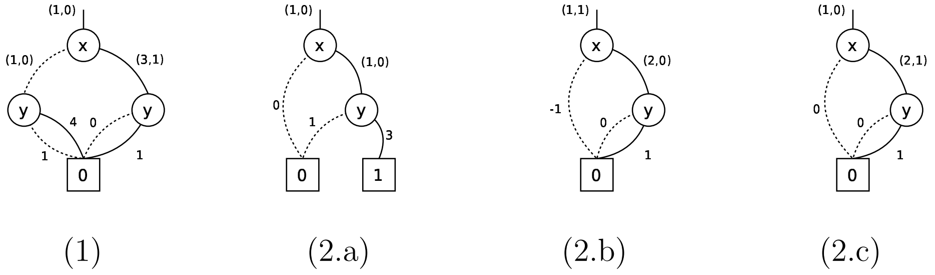


Figure 3.1: Example for OWDDs with multiplicative and additive edge-weights

It is possible to define another kind of expansion on OWDDs to cover OBDD variants like ZBDDs, BMDs, ... We will only focus on the Shannon's Expansion because it is the most intuitive approach. All following definitions may be adapted to almost any kind of expansion in a natural way. ZBDDs are an exception and need a special treatment because non-essential variables occur in a ZBDD as nodes. Furthermore, skipped levels on a path to the one drain represent an assignment with zero. Therefore the number of essential variables does not correspond to the number of occupied levels in the associated ZBDD. The following proofs and definitions are based on expansions in which the number of essential variables and the number of occupied levels are equal.

Before discussing the reducedness and canonicity for OWDDs, we must observe that the notion of OWDDs covers several types of known BDD-variants. For $\mathbb{K} = \{0, 1\}$ our notion of a OWDD specializes to an ordinary ordered BDD [21] when dealing with $\Phi = \{id\}$ and to ordered BDDs with complement bits for the edges [102] when dealing with $\Phi = \{id, \neg\}$. For $\mathbb{K} = \mathbb{R}$ (or $\mathbb{K} = \mathbb{N}$ or any other semi-ring) MTBDDs [8, 31, 43] are obtained through $\Phi = \{id\}$, while edge-valued BDDs [119] arise by taking $\Phi = \{x \mapsto x + b : b \in \mathbb{K}\}$. In these examples, the incoming edges of the terminal nodes do not play a special role and we may deal with $\Phi^{const} = \Phi$ in either case. Factored edge-valued BDDs (FEVBDDs) are obtained by taking $\Phi = \{x \mapsto ax + b : a, b \in \mathbb{K}, a \neq 0\}^3$ and Φ^{const} the set of functions $x \mapsto x + b$ where $b \in \mathbb{K}$. Fig. 3.1 shows four FEVBDDs where we simply write (a, b) to denote the function $x \mapsto ax + b$. The OWDD in (1) represents the function $3y + 1$, while the OWDDs in (2.a), (2.b) and (2.c) represent the function $f = x(1 + 2y)$. The following example shows how the function f represented by the OWDD in (2.a) can be calculated.

- $f_y = (1 - y) \cdot (1 + 0) + y \cdot (3 + 1) = 3y + 1$
- $f_x = (1 - x) \cdot (0 + 0) + x \cdot (1 \cdot f_y + 0) = x \cdot (3y + 1)$
- $f = 1 \cdot f_x + 0 = x \cdot (3y + 1)$

Note that it is not possible to differentiate between FEVBDDs and NADDs because they have equal transformation sets.

[3] $a \neq 0$ because elements of Φ must be bijections.

3.1.2 Reducedness and Canonicity

With the definition of OWDDs it is possible to cover all before mentioned OBDD variants. The remaining questions are how to formalize freedom of redundancies and how to ensure the unique representation of functions by π-OWDDs.

Intuitively, the freedom of redundancies in OWDDs means that two different nodes in a OWDD represent transformations of different functions, i.e., if f and g are the represented functions then $f \notin \{\varphi \circ g | \varphi \in \Phi\}$.

Notation 3.1.4 (The equivalence $\equiv$)**.** Let $\mathbb{F}'$ be $\mathbb{F}^{\textit{non-const}}$ or $\mathbb{F}^{\textit{const}}$ and Φ' the corresponding set of transformations, i.e., $\Phi' = \Phi$ if $\mathbb{F}' = \mathbb{F}^{\textit{non-const}}$ and $\Phi' = \Phi^{\textit{const}}$ if $\mathbb{F}' = \mathbb{F}^{\textit{const}}$. Let $\equiv_{\Phi'}$ be the following equivalence on $\mathbb{F}'$ such that:

$$\text{If } f, g \in \mathbb{F}' \text{ then } f \equiv_{\Phi'} g \text{ iff there exists } \varphi \in \Phi' \text{ with } f = \varphi \circ g.^4$$

Capital letters $F, G, \ldots$ will be used for the equivalence classes of $\mathbb{F}$ under $\equiv$. □

In the following section, we will use the notation $\equiv$ for both $\equiv_\Phi$ and $\equiv_{\Phi^{\textit{const}}}$ if the meaning can be understood by the context. Additionally, we will adapt that for all $f \in \mathbb{F}^{\textit{const}}$ and $g \in \mathbb{F}^{\textit{non-const}}$ follows $f \not\equiv g$.

Definition 3.1.5 (Reduced OWDDs)**.** A π-OWDD $\mathcal{B}$ is called reduced iff for all nodes v, w in $\mathcal{B}$ we have $v \neq w$ implies $f_v \not\equiv f_w$. □

For instance, the two π-OWDDs shown in Fig. 3.1 (2.b) and (2.c) are reduced, while the ones in Fig. 3.1 (1) and (2.a) are not. In (2.a), the two sinks represent (constant) functions that can be transformed into the other via bijections in $\Phi^{\textit{const}}$. In (1), the two y-nodes represent (non-constant) functions that are equivalent.

The term reducedness on OWDDs still leaves freedom to choose the representatives in the $\equiv$-equivalence classes. Thus, reduced π-OWDDs for the same function need not to be isomorphic. (We use the notion "isomorphism" for π-OWDDs $\mathcal{B}$ and $\mathcal{C}$ in the sense that $\mathcal{B}$ and $\mathcal{C}$ agree up to renaming of the nodes.) Instead, as we will see in Theorem 3.1.7, reduced π-OWDDs are *weakly isomorphic* meaning that they agree when abstracting away from the names of the nodes and ignoring the weights for the edges and the root. In particular, weakly isomorphic OWDDs have the same size (number of nodes).

Lemma 3.1.6. *Let $f, g \in \mathbb{F}$ and $f \equiv g$. Then, f and g have the same essential variables and for all $z \in \mathcal{Z}$ we have $f|_{z=0} \equiv g|_{z=0}$ and $f|_{z=1} \equiv g|_{z=1}$.*

Proof. Obvious as $f = \varphi \circ g$ implies $f|_{z=\xi} = \varphi \circ g|_{z=\xi} \equiv g|_{z=\xi}$. □

Lemma 3.1.6 yields that cofactors can be built for $\equiv$-equivalence classes. That is, if $F \subseteq \mathbb{F}$ and $\xi \in \{0, 1\}$ then we may write $F|_{z=\xi}$ to denote the unique $\equiv$-equivalence class that contains the functions $f|_{z=\xi}$ for all $f \in F$. A similar notation $F|_{z_1=\xi_1,\ldots,z_k=\xi_k}$ is used if we consider cofactors for several variables.

[4] Note that $\equiv_{\Phi'}$ is in fact an equivalence as we have $f = \varphi \circ g$ implies $g = \varphi^{-1} \circ f$. Moreover, $\varphi \circ f = \psi \circ g$ implies $f = (\varphi^{-1} \circ \psi) \circ g \equiv_{\Phi'} g$.

Theorem 3.1.7. [Weak canonicity of reduced OWDDs] Let π be a variable ordering and $\mathcal{B}$, $\mathcal{C}$ reduced π-OWDDs. Then: If $f_{\mathcal{B}} \equiv f_{\mathcal{C}}$ then $\mathcal{B}$ and $\mathcal{C}$ are weakly isomorphic. In particular, $|\mathcal{B}| = |\mathcal{C}|$.

Proof. Let $\pi = (z_1, \ldots, z_n)$ be the variable ordering. From $f_{\mathcal{B}} \equiv f_{\mathcal{C}}$ we derive $[f_{\mathcal{B}}]_{\equiv} = [f_{\mathcal{C}}]_{\equiv}$. That together with Lemma 3.1.6 yields that we can argument with $\mathcal{B}$ and get a result for all equivalent π-OWDDs. If $F_{\mathcal{B}} = [f_{\mathcal{B}}]_{\equiv}$ then there is a 1-1-correspondence between the cofactors $F_{\mathcal{B}}|_{z_1=\xi_1,\ldots,z_k=\xi_k}$ and the nodes in $\mathcal{B}$ and $\mathcal{C}$ respectively. (Note that some of these cofactors might agree.) To see why, we may use an inductive argument starting in the root node. If v is an inner node in $\mathcal{B}$ such that $[f_v]_{\equiv} = F_{\mathcal{B}}|_{z_1=\xi_1,\ldots,z_k=\xi_k}$ then v is labelled with a variable z_ℓ where $\ell > k$ and z_ℓ is the first essential variable for f_v (and all functions in $[f_v]_{\equiv}$). Moreover, the function f_{v_0} for the 0-successor v_0 of v is in the equivalence class

$$[f_v]_{\equiv}\big|_{z_\ell=0} = F_{\mathcal{B}}|_{z_1=\xi_1,\ldots,z_k=\xi_k,\ldots,z_\ell=0}$$

for arbitrary assignments of the variables $z_{k+1}, \ldots, z_{\ell-1}$. A similar condition holds for the 1-successor of v. Hence, if we ignore the edge-weights then all reduced π-OWDDs for the same function have the same structure. □

Note that it is still not possible to differentiate between FEVBDDs and NADDs. Theorem 3.1.7 implies that both OBDD variants have the same structure and thus the same number of nodes but different weights on the edges.

The next goal is to achieve a criteria for canonicity. We mentioned that there is still the freedom to choose a representative out of an equivalence class. We also alluded to several possibilities for ensuring canonicity for different OBDD variants. In the same manner as in FEVBDDs and NADDs, we now define a selection function $\mathcal{S} : \mathbb{F}/\equiv \to \mathbb{F}$ which "selects" a unique representative $\mathcal{S}(F) \in \mathbb{F}$ out of any equivalence class $F \in \mathbb{F}/\equiv$.

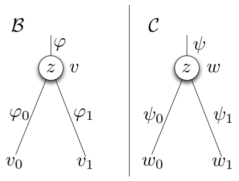
For $f \in \mathbb{F}$, we simply write $\mathcal{S}(f)$ rather than $\mathcal{S}([f]_{\equiv})$. Thus, $f \equiv \mathcal{S}(f)$ for all $f \in \mathbb{F}$.

Definition 3.1.8 ($\mathcal{S}$-reduced OWDDs)**.** A π-OWDD $\mathcal{B}$ is called $\mathcal{S}$-reduced if $\mathcal{B}$ is reduced and $f_v = \mathcal{S}(f_v)$ for all nodes v in $\mathcal{B}$. □

With this selection function it is now possible to proof the canonicity of $\mathcal{S}$-reduced OWDDs.

Theorem 3.1.9. [Canonicity of $\mathcal{S}$-reduced OWDDs] Let π be a variable ordering, $\mathcal{S}$ a selection function and let $\mathcal{B}$, $\mathcal{C}$ be $\mathcal{S}$-reduced π-OWDDs with $f_{\mathcal{B}} = f_{\mathcal{C}}$. Then, $\mathcal{B}$ and $\mathcal{C}$ are isomorphic.

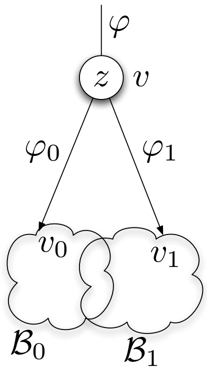
Proof. Our argument is by induction on $n = |\mathcal{Z}|$ (number of variables). If $n = 0$ then $f_{\mathcal{B}} = f_{\mathcal{C}}$ is constant. Let c be the value of $f_{\mathcal{B}} = f_{\mathcal{C}}$ and $c' = \mathcal{S}(c)$. Then, the root of $\mathcal{B}$ and $\mathcal{C}$ consists of a terminal node labelled with c' together with the unique transformation $\varphi \in \Phi^{const}$ such that $\varphi(c') = c$.



In the induction step, we assume that the root nodes of $\mathcal{B}$ and $\mathcal{C}$ are inner nodes, say z-nodes. Then, z is the first essential variable in $f_{\mathcal{B}} = f_{\mathcal{C}}$ according to the ordering π. Let $r_{\mathcal{B}} = \langle \varphi, v\rangle$ be the root of $\mathcal{B}$ and $r_{\mathcal{C}} = \langle \psi, w\rangle$ the root of $\mathcal{C}$. Then, $\varphi \circ f_v = f_{\mathcal{B}} = f_{\mathcal{C}} = \psi \circ f_w$. Thus, $f_v \equiv f_w$. As $\mathcal{B}$ and $\mathcal{C}$ are $\mathcal{S}$-reduced we have $f_v = f_w$ and $\varphi = \psi$ (by condition (2) of Φ, cf. Notation 3.1.1). For $\xi \in \{0,1\}$, let $v_\xi = succ_\xi(v)$, $w_\xi = succ_\xi(w)$, $\phi_\xi(v) = \varphi_\xi$, $\phi_\xi(w) = \psi_\xi$. Then, $\varphi_\xi \circ f_{v_\xi} = f_v|_{z=\xi} = f_w|_{z=\xi} = \psi_\xi \circ f_{w_\xi}$. Hence, $f_{v_\xi} \equiv f_{w_\xi}$. Again, as $\mathcal{B}$ and $\mathcal{C}$ are $\mathcal{S}$-reduced, we get $f_{v_\xi} = f_{w_\xi}$ and $\varphi_\xi = \psi_\xi$ (by the conditions for Φ and Φ^{const}). Applying the induction hypothesis, the sub-OWDD with root nodes v_ξ and w_ξ are isomorphic. Hence, $\mathcal{B}$ and $\mathcal{C}$ are isomorphic. □

Theorem 3.1.10. [Universality and uniqueness of $\mathcal{S}$-reduced OWDDs] Let π be a variable ordering, $\mathcal{S}$ a selection function and $f : Eval(\mathcal{Z}) \to I\!K$. Then, there exists a unique $\mathcal{S}$-reduced π-OWDDs $\mathcal{B}$ with $f = f_{\mathcal{B}}$. (Uniqueness is up to isomorphism.)

Proof. It remains to provide the proof of the existence of an $\mathcal{S}$-reduced π-OWDD for f. The construction is by induction on the number n of essential variables of f. If $n = 0$ then f is constant. Let $c = \mathcal{S}(f)$. There exists a $\varphi \in \Phi^{const}$ with $\varphi(c) = f$. Thus, we may use an OWDD consisting of the root $\langle \varphi, v\rangle$ where v is a terminal node v labelled with c. In the induction step, we assume that f is not constant.



Let z be the first essential variable of f according to π and let $g = \mathcal{S}(f)$ and $f = \varphi \circ g$ where $\varphi \in \Phi$. For $\xi \in \{0,1\}$, let $g_\xi = \mathcal{S}(g|_{z=\xi})$ and $g|_{z=\xi} = \varphi_\xi \circ g_\xi$ where $\varphi_\xi \in \Phi$ if $g|_{z=\xi}$ is not constant and $\varphi_\xi \in \Phi^{const}$ if $g|_{z=\xi}$ is constant. By induction hypothesis there exist $\mathcal{S}$-reduced π-OWDDs $\mathcal{B}_0$ and $\mathcal{B}_1$ for g_0 and g_1 respectively. We may assume w.l.o.g. that $\mathcal{B}_0$ and $\mathcal{B}_1$ share the same nodes for common cofactors. More precisely, we may assume that if w_0 is a node in $\mathcal{B}_0$ and w_1 a node in $\mathcal{B}_1$ such that $f_{w_0} = f_{w_1}$ then $w_0 = w_1$. (Otherwise the nodes in $\mathcal{B}_1$ can be renamed as there is an isomorphism between the sub-OWDDs with root nodes w_0 and w_1, cf. Theorem 3.1.9.) We then may compose $\mathcal{B}_0$, $\mathcal{B}_1$ to a $\mathcal{S}$-reduced π-OWDD for f as shown in the picture on the left. □

3.1.3 Examples

In the section above we showed how to define reducedness and canonicity on OWDDs by means of a selection function. We will now show how the selection function for different already known OBDD variants may be chosen.

OBDDs

The ordinary OBDDs have no other transformation than the identity, so that the equivalence classes only contain one function. From this point of view there is no

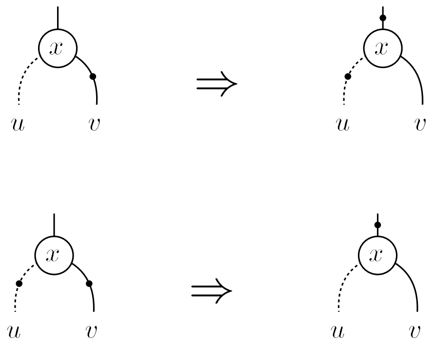


Figure 3.2: Transformation rules for negative edges

freedom to choose a selection function, i.e.,

$$\mathcal{S}(f) = f \quad \forall f \in I\!\!F.$$

OBDDs with negative edges In the case of OBDDs with negative edges every equivalence class contains two functions. The constant functions $I\!\!F^{const}$ are in the same equivalence class, i.e., we have the freedom to choose the terminal node. In much of the literature the one drain is used, so that $\mathcal{S}(f) = 1 \quad \forall f \in I\!\!F^{const}$.

The non-constant functions need a different kind of selection. We will assume that only the zero-successor can have a non-trivial weight. Figure 3.2 shows that we can transform a OBDD with negative edges without any restriction to a valid[5] OBDD. This leads to the following selection function:

$$\mathcal{S}(f)(1, \dots, 1) = 1 \quad \forall f \in I\!\!F$$

ADDs

As in the case of OBDDs (without negative edges) we have no freedom to choose a selection function and thus

$$\mathcal{S}(f) = f \quad \forall f \in I\!\!F.$$

EVBDDs

Edge-valued BDDs have $\Phi = \{x \mapsto x + b : b \in I\!\!K\}$ as transformation set. To ensure canonicity the zero-successors always have the identity as a weight. This restriction leads to a unique selection function

$$\mathcal{S}(f)(0, \dots, 0) = 0 \quad \forall f \in I\!\!F$$

and thus ensures canonicity for EVBDDs.

[5] Only zero-successors can have a non-trivial weight.

FEVBDDs

In all previous examples the transformation sets Φ and Φ^{const} were the same. This made the definition of a selection function much easier than in the case of FEVBDDs (and NADDs). FEVBDDs are the first variant with different transformation sets Φ and Φ^{const}. The constant functions have the same transformations as EVBDDs so that this case is handled in the same manner, i.e., $\mathcal{S}(c) = 0 \quad \forall c \in I\!\!F^{const}$. The non-constant functions must be treated differently. As previously mentioned we restrict FEVBDDs so that the identity function is assigned to the zero-successor. This is similar to SOBDDs with negative edges as every FEVBDD node can be transformed into a restricted one. Unfortunately, this is not enough to define a unique selection function. Figure 3.3 shows that there is still freedom to choose different representation for the same function.

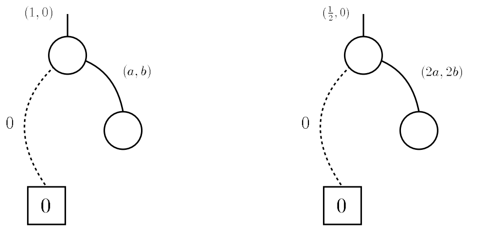


Figure 3.3: Two different zero-successor-restricted FEVBDDs representing the same function

The shown case is the only exception that is not covered by restricting the zero-successor. Let z be the first essential variable[6] of a non-constant function f with $f|_{z=0} \notin I\!\!F^{const}$. Then, this restriction can be formalized as follows:

$$\mathcal{S}(f) = g \text{ such that } g|_{z=0} = \mathcal{S}(g|_{z=0})$$

The remaining case if f is a non-constant function and $f|_{z=0} \in I\!\!F^{const}$ can be handled by fixing the multiplicative weight of the one-successor. This leads to the following selection function: $\mathcal{S}(f) = g$ such that:

- $g|_{z=0} = \mathcal{S}(g|_{z=0})$
- $g|_{z=1} = \mathcal{S}(g|_{z=1}) + b$ with capable $b \in I\!\!R$

Note that the second condition is similar to the EVBDD equivalence.

NADDs NADDs and FEVBDDs just differ in in the selection function. With Theorem 3.1.7 it follows that the NADD and FEVBDD representation of a function have the same node-structure and thus the same size. This is an important result from the OWDD framework concept.

[6] with respect to the given variable ordering

As previously noted, NADDs and FEVBDDs have different selection functions. The constant functions are the same as in the case of FEVBDDs, i.e., $\mathcal{S}(c) = 0 \quad \forall c \in \mathbb{F}^{const}$. At first it seems that normalization makes the definition of the selection function more straightforward. But unfortunately this is not the case. Before discussing this in detail we will first define conditions for the selection function.

Let f be a non-constant function. $\mathcal{S}(f) = g$ such that:

- $g \in (\{0,1\}^n \to [0,1])$
- $\min g = 0$
- $\max g = 1$

It is easy to see why $\mathcal{S}$ cannot select a unique representative out of every equivalence class. Let us assume g fulfills all conditions above, then $1 - g$ would also fulfill all conditions. The next goal is to establish additional conditions for $\mathcal{S}$, so that $\mathcal{S}$ is well-defined in the end.

Let f be a non-constant function and let z be the first essential variable with $f|_{z=0} \notin \mathbb{F}^{const}$ and $\mathcal{S}(f) = g$. Lemma 3.1.6 implies that

$$g|_{z=0} \equiv \mathcal{S}(g|_{z=0})$$

which is equivalent to

$$\lambda_0 \cdot g|_{z=0} + \tau_0 = \mathcal{S}(g|_{z=0}) \quad \lambda_0 \in \mathbb{R}^{\setminus\{0\}}, \tau_0 \in \mathbb{R}.$$

To select a unique function we demand that $\lambda_0 > 0$.

Analogously, if $f|_{z=0} \in \mathbb{F}^{const}$ we demand that $\lambda_1 > 0$ for

$$\lambda_1 \cdot g|_{z=1} + \tau_1 = \mathcal{S}(g|_{z=1}) \quad \lambda_1 \in \mathbb{R}^{\setminus\{0\}}, \tau_1 \in \mathbb{R}.$$

3.2 Higher-Order Weighted Decision Diagrams

In the last section we demonstrated how the different known OBDD variants compare in several applications. We will focus on the fact that every previously introduced OBDD variant is based on output transformations. Each variant was introduced to overcome a special problem, e.g., OBDDs with negative edges were introduced to negate a function in constant time. JINC is supposed to support a wide range of transformation sets so it is necessary to discuss if the supported transformation sets could be expanded without loosing the general concept of OWDDs.

The goal of this chapter is to develop a more general framework than OWDDs to support a wider range of transformation functions.

This framework influenced JINC's design described in Chapter 4. The necessity to support a wider range of transformation functions can be seen on the example of a variant that is no instance of OWDDs but an instance of the more general framework in Chapter 5.

3.2.1 Definition and Semantics

The definitions for this framework are similar to the definitions of OWDDs.

The main difference is that it is now possible to handle non-boolean variables (comparable to MDDs [60]).

Definition 3.2.1 (Evaluation)**.** Let $\mathcal{Z}_m = \{z_1, \ldots, z_n\}$ be a finite set of variables over $\{0, 1, \ldots, m-1\}$ with $m \geq 2$.

An *evaluation* of $\mathcal{Z}_m$ is a function

$$\eta : \mathcal{Z}_m \to \{0, 1, \ldots, m-1\}$$

that assigns a value $\eta(z) \in \{0, 1, \ldots, m-1\}$ to each variable $z \in \mathcal{Z}_m$.
$\mathit{Eval}(\mathcal{Z}_m)$ identifies the set of all evaluations of $\mathcal{Z}_m$.

Let $\overline{a} = (a_1, \ldots, a_n) \in \{0, 1, \ldots, m-1\}^n$ and $\overline{z} = (z_{i_1}, \ldots, z_{i_n}) \in \mathcal{Z}_m^n$ with pairwise different z_{i_j}, then $[\overline{z} = \overline{a}]$ represents the evaluation $\eta \in \mathit{Eval}(\mathcal{Z}_m)$ with

$$\eta(z_{i_j}) = a_j,\ j = 1, \ldots, n.$$

□

At first it is important to define which kind of transformations can be used.

Recall that $\mathbb{F}$ is the set of functions $f : \mathit{Eval}(\mathcal{Z}) \to \mathbb{K}$. The definition of $\mathcal{Z}_m$ applies analogously to $\mathbb{F}^{(m)}$.

Notation 3.2.2 (The Ψ sets)**.** Let $\mathbb{F}^{(m)\prime} \subseteq \mathbb{F}^{(m)}$ denote a nonempty set of functions. $\Psi_{\mathbb{F}^{(m)\prime}}$ denotes a nonempty set of bijections $\mathbb{F}^{(m)\prime} \to \mathbb{F}^{(m)\prime}$ such that

(1) $\Psi_{\mathbb{F}^{(m)\prime}}$ is closed under inversion and composition, i.e., if $\psi_1, \psi_2 \in \Psi_{\mathbb{F}^{(m)\prime}}$ then $\psi_1^{-1} \in \Psi_{\mathbb{F}^{(m)\prime}}$ and $\psi_1 \circ \psi_2 \in \Psi_{\mathbb{F}^{(m)\prime}}$. (In particular, $\Psi_{\mathbb{F}^{(m)\prime}}$ contains the identity *id*.)

(2) If $f \in \mathbb{F}^{(m)\prime}$, $\psi_1 \in \Psi_{\mathbb{F}^{(m)\prime}}$ and $f = \psi_1(f)$ then $\psi_1 = id$. □

As in the case of OWDDs we will assess the edges of higher-order weighted decision diagrams (HOWDDs) with transformation functions.

As seen in the example of FEVBDDs and NADDs, the constant functions in some OBDD variants with weighted edges require special treatment as there might be several possibilities to transform a constant function into another constant function via the bijections $\psi \in \Psi_{\mathbb{F}^{(m)\prime}}$. For these OBDD variants $\mathbb{F}^{(m)}$ must be partitioned into constant and non-constant functions. With this definition of transformation the restriction of only two function sets would limit possible transformations, e.g., the compose operator $f \mapsto f\{z/(1-z)\}$ would not fulfill condition (2) as $x\{z/(1-z)\} = id(x) = x$. To overcome this shortage, we partition $\mathbb{F}^{(m)}$ into several function sets with corresponding transformation sets.

Notation 3.2.3 (System)**.** In the following let $\mathbb{F}_1^{(m)}, \ldots, \mathbb{F}_k^{(m)}$ a partition of $\mathbb{F}^{(m)}$ and

$$\mathfrak{F}^{(m)} = \left\{(\mathbb{F}_1^{(m)}, \Psi_{\mathbb{F}_1^{(m)}}), \ldots, (\mathbb{F}_k^{(m)}, \Psi_{\mathbb{F}_k^{(m)}})\right\}$$

a system that fulfills the following conditions:.

(a) $\mathbb{F}_i^{(m)} \cap \mathbb{F}_j^{(m)} = \emptyset$ for $i \neq j$

(b) $\bigcup_i \mathbb{F}_i^{(m)} = \mathbb{F}^{(m)}$

(c) Every $\mathbb{F}_i^{(m)}$ has its corresponding transformation set $\Psi_{\mathbb{F}_i^{(m)}}$ satisfying conditions (1) and (2) in Notation 3.2.2.

(d) For every tuple $(\mathbb{F}_i^{(m)}, \Psi_{\mathbb{F}_i^{(m)}})$ the following condition must hold:
$\forall \psi \in \Psi_{\mathbb{F}_i^{(m)}}, f \in \mathbb{F}_i^{(m)} \Rightarrow \psi(f) \in \mathbb{F}_i^{(m)}$

□

Conditions (a) and (b) ensure that every function has exactly one corresponding function set. While condition (d) assures that all function sets $\mathbb{F}_i^{(m)}$ are closed under their transformations $\Psi_{\mathbb{F}_i^{(m)}}$. With this constrain it becomes reasonable to define equivalence classes on $\mathbb{F}^{(m)}$.

Notation 3.2.4 (The equivalence $\equiv$)**.** Let $\equiv$ be the following equivalence on $\mathbb{F}^{(m)}$ such that:

If $f \in \mathbb{F}^{(m)}$ and $g \in \mathbb{F}_i$ then $f \equiv g$ iff there exists $\psi \in \Psi_{\mathbb{F}_i^{(m)}}$ with $f = \psi(g)$.[7] □

[7] Note that $\equiv$ is in fact an equivalence as condition (c) ensures symmetry and transitivity.

This definition of equivalence implies that if $f \in \mathbb{F}_i^{(m)}$ then $[f]_{\equiv} \subseteq \mathbb{F}_i^{(m)}$ for a system $\mathfrak{F}^{(m)}$. From this it is clear that every non-empty function set $\mathbb{F}_i^{(m)}$ contains at least one equivalence class.

Note, that constant functions and non-constant functions can be contained in the same equivalence class.

Definition 3.2.5 (Higher-Order Weighted Decision Diagram (HOWDD))**.** Let $\mathcal{Z}_m$, $\mathbb{K}$, $\mathfrak{F}^{(m)}$ be as above and π a variable ordering for $\mathcal{Z}_m$. A π-HOWDD *higher-order weighted decision diagram* for $(\mathcal{Z}_m, \mathbb{K}, \mathfrak{F}^{(m)})$ is a finite rooted, m-ary branching, acyclic graph $\mathcal{B}$ with additional information. For the inner nodes, we have

- a function *var* that assigns a variable $var(v) \in \mathcal{Z}_m$ to any inner node v,
- functions $v \mapsto succ_i(v)$ with $i \in \{0, 1, \ldots, m-1\}$ that specify the successors of v,
- functions $v \mapsto \Psi_i(v)$ with $i \in \{0, 1, \ldots, m-1\}$ that specify the transformations associated with the outgoing edges from v.

For the terminal nodes, we have function $v \mapsto value(v) \in \mathbb{K}$. If v is an inner node and $\xi \in \{0, 1, \ldots, m-1\}$ such that $succ_\xi(v)$ is an inner node, then we require that $var(v)$ occurs in π before $var(succ_\xi(v))$. If $f_{succ_\xi(v)} \in \mathbb{F}_i^{(m)}$, then we require $\Psi_\xi(v) \in \Psi_{\mathbb{F}_i^{(m)}}$[8].
The root of $\mathcal{B}$ is a pair $r = \langle \Psi_r, v_r \rangle$ consisting of a function $\Psi_r \in \bigcup_i \Psi_{\mathbb{F}_i^{(m)}}$ and a node v_r from which all other nodes in $\mathcal{B}$ are reachable. As for the edges we require $\Psi_r \in \Psi_{\mathbb{F}_i^{(m)}}$ if $f_{v_r} \in \mathbb{F}_i^{(m)}$. □

If m is clear from the context we will leave this parameter out.

The semantics of a HOWDD is formalized in the same way as in OWDDs.

Notation 3.2.6 (Semantics of HOWDDs)**.** The semantics of a HOWDD $\mathcal{B}$ is formalized by associating a function

$$f_v \in \mathbb{F}^{(m)}$$

to any node in $\mathcal{B}$ and a function for the root r. Intuitively, an incoming edge of node v labelled with ψ stands for the function $\psi \circ f_v$. Formally, the function $f_{\mathcal{B}}$ for HOWDD $\mathcal{B}$ is induced by its root $r = \langle \psi_r, v_r \rangle$ which is given by $f_{\mathcal{B}} = \psi_r \circ f_{v_r}$ where the function f_v for the nodes is defined in a bottom-up fashion. (The level of a z-node denotes the position of variable z in π. The nodes on the bottom-level are the terminal nodes.)

[8] See Notation 3.2.6 for a formulation of f_v with terminal or inner node v.

The terminal nodes represent constant functions as expected, i.e., $f_v = value(v)$ for any terminal node v. If v is a z-node then $f_v : Eval(\mathcal{Z}_m) \to I\!K$ is defined as follows: Let $\eta \in Eval(\mathcal{Z}_m)$ and $\eta(z) = \xi \in \{0, 1, \ldots, m-1\}$ then

$$f_v(\eta) = \psi_\xi(v) \circ f_{succ_\xi(v)}(\eta).$$

That is, if $I\!K = I\!R$ then we may write f_v as

$$f_v \ = \sum_{i=0}^{m-1} \delta_i(z) \cdot (\Psi_i(v)(f_{succ_i(v)}))$$

with $\delta_i(z) = \begin{cases} 1 & \text{if z=i} \\ 0 & \text{otherwise} \end{cases}$ and

$$f_v \ = \ (\neg z \wedge \Psi_0(v)(f_{succ_0(v)})) \ \vee \ (z \wedge \Psi_1(v)(f_{succ_1(v)}))$$

if $I\!K = \{0, 1\}$ and $m = 2$.

However with this definition of $\mathfrak{F}^{(m)}$ it is not guaranteed that the cofactors of two equivalent functions are equivalent. Thus, unlike OWDDs (see Theorem 3.1.7), HOWDDs over the same system $\mathfrak{F}_k^{(m)}$ and for the same function f might not be structural equivalent.

Example 3.2.7. Let $m = 2$, $I\!K = \{0, 1\}$, $f = \neg x \wedge y$, $g = x \wedge y$, $\psi = \{x/y, y/\neg x\}$, $\Psi_{I\!F'} = \{\psi, \psi^{-1}, id\}$ and $I\!F' = \{f, g\}$. $f \equiv g$ because $f = \psi(g)$. It is obvious that $f|_{x=0} = y \not\equiv g|_{x=0} = 0$. If $f|_{x=0}$ is transformed via $\psi^{-1} = \{y/x, x/\neg y\}$ then $\psi^{-1}(f|_{x=0}) = x = g|_{y=1}$, although $f|_{x=0} = y \not\equiv x = g|_{y=1}$ then. ■

3.2.2 Reducedness and Canonicity

The term reducedness is defined the same way as for OWDDs:

Definition 3.2.8 (Reduced HOWDDs)**.** A π-HOWDD $\mathcal{B}$ is called reduced iff for all nodes v, w in $\mathcal{B}$ we have $v \neq w$ implies $f_v \not\equiv f_w$. □

Canonicity is not guaranteed without further restrictions. The main problem in the case of HOWDDs are transformations that alter the variables of a function. It is possible that a transformation increases the number of essential variables of a formula or renames the variables in it. Both cases make it more difficult to define a selection function akin to guaranteeing canonicity with OWDDs.

The next example illustrates the problem with essential variable increasing transformations.

Example 3.2.9. Let $m = 2$, $I\!K = \mathbb{B}, I\!F_1 = \{x \wedge y, y\}$ and $I\!F_2 = I\!F \backslash I\!F_1$. The corresponding transformation sets are:

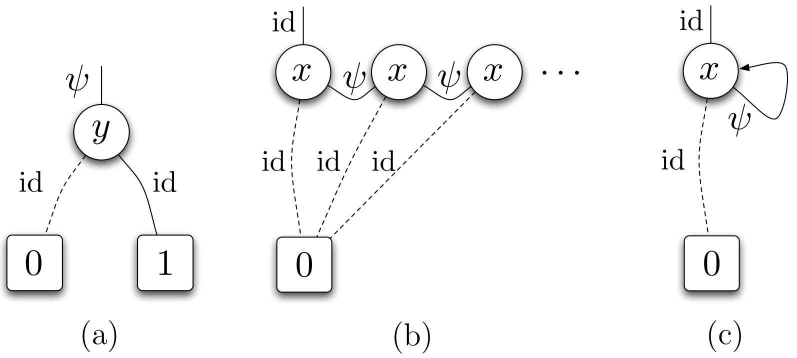


Figure 3.4: Different HOWDDs for $x \wedge y$.

- $\Psi_1 = \{id, \psi\}$ with $\psi(f) = \begin{cases} x \wedge y & \text{if } f = y \\ y & \text{if } f = x \wedge y \\ f & \text{otherwise} \end{cases}$
  Note that $\psi : \mathbb{F}_1 \to \mathbb{F}_1$ is bijective (see Notation 3.2.2) and $\psi^{-1} = \psi$.
- $\Psi_2 = \{id\}$

It is clear that $\mathfrak{F} = \{(\mathbb{F}_1, \Psi_1), (\mathbb{F}_2, \Psi_2)\}$ builds a transformation system (as in Notation 3.2.3).

We now follow the same idea as in the proof of Theorem 3.1.10 to construct HOWDDs for the function $f = x \wedge y$. We use the variable ordering $\pi = (x, y)$.

Now we have two possible choices for a representative g out of $[f]_\equiv = \mathbb{F}_1$.

(1) We assume that $g = y$, i.e., $f = \psi(g)$ and y is the first essential variable of f according to π. The cofactors $g|_{y=0}$ and $g|_{y=1}$ are functions of $\mathbb{F}_2$. All functions in $\mathbb{F}_2$ build their own equivalence class and thus the representation of f should look like Figure 3.4(a).

(2) We assume that $g = x \wedge y$, i.e., x is the first essential variable of f according to π. The cofactor $g|_{x=0} \in \mathbb{F}_2$ and $g|_{x=1} \in \mathbb{F}_1$. We assumed that $x \wedge y$ is the representative of $\mathbb{F}_1$. Therefore the function $g|_{x=1} = y$ must be represented by $\psi(y) = x \wedge y$ [9]. This leads to a repeating or cyclic behavior (see Figure 3.4(b) or (c)[10]) and thus this function cannot be represented by a HOWDD with this specific selection function. The mentioned behavior would also violate the ordering conditions of HOWDDs.

■

The next example shows the problem with permutations.

Example 3.2.10. Let $\mathbb{K} = \mathbb{B}, \mathbb{F}_1 = \{x, y\}$ and $\mathbb{F}_2 = \mathbb{F} \backslash \mathbb{F}_1$ be the function sets and $\pi = (x, y)$ the variable ordering. The corresponding transformation sets are:

[9] This is because otherwise two different nodes representing equivalent functions would exist.

[10] Both cases are not possible because HOWDDs have to be finite and acyclic.

- $\Psi_1 = \{id, \psi\}$ with $\psi(f) = f\{x \leftarrow y, y \leftarrow x\}$ Then ψ is a bijection and $\psi = \psi^{-1}$.
- $\Psi_2 = \{id\}$

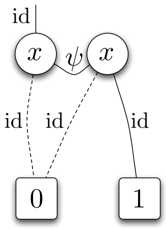
It is clear that $\mathfrak{F} = \{(\mathbb{F}_1, \Psi_1), (\mathbb{F}_2, \Psi_2)\}$ builds a transformation system (as in Notation 3.2.3) with essential variable conserving transformations, i.e., the transformations do not change the number of essential variables.

We will now define a selection function $\mathcal{S}$ for the equivalence classes of $\mathfrak{F}$.

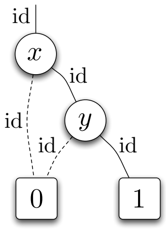
All functions in $\mathbb{F}_2$ build their own equivalence class and thus we do not have a choice for a selection ($\mathcal{S}([f]_\equiv) = f$ if $f \in \mathbb{F}_2$). The only choice remains in the equivalence class $[x]_\equiv = [y]_\equiv = \mathbb{F}_1$ ($\mathcal{S}(\mathbb{F}_1) = x$ or $\mathcal{S}(\mathbb{F}_1) = y$).

If we choose x as a representative, i.e., $\mathcal{S}(\mathbb{F}_1) = x$, there are functions that cannot be represented by a reduced HOWDD with selection function $\mathcal{S}$. Let $f = x \wedge y$ be the function we want to represent with a π-HOWDD. We will follow the same steps as in Theorem 3.1.10 to try to construct the π-HOWDD.

We start with $f = x \wedge y$. The first essential variable with respect to π is x. The negative cofactor of f is the constant zero function, i.e., $f|_{x=0} = 0$. The positive cofactor $f|_{x=1} = y$ cannot be represented directly because x and y lie in the same equivalence class and $\mathcal{S}$ selects x as representative. Again x is the first essential variable and thus this would lead to a HOWDD that violates the ordering conditions (see the picture on the right).



In the other case, if we choose y as a representative for the equivalence class $\mathbb{F}_1$, we can construct the π-HOWDD for $f = x \wedge y$. The positive cofactor $f|_{x=1} = y$ can be represented directly and thus this HOWDD does not violate the ordering conditions (see the picture on the right for a valid reduced π-HOWDD for $x \wedge y$).



■

For both examples above the order condition can be met by a suitable a selection function $\mathcal{S}$. The key for solving the problems shown in Example 3.2.9 and 3.2.10 is to regard the *index* of the equivalence classes.

Definition 3.2.11 (Index of a function)**.** Let $\pi = (z_{i_1}, \dots, z_{i_n})$ be a variable ordering over $\mathcal{Z}_m = \{z_1, \dots, z_n\}$, $\mathfrak{F}^{(m)} = \left\{(\mathbb{F}_1^{(m)}, \Psi_{\mathbb{F}_1^{(m)}}), \dots, (\mathbb{F}_k^{(m)}, \Psi_{\mathbb{F}_k^{(m)}})\right\}$ a transformation system and $f \in \mathbb{F}^{(m)}$ a function. The *index* $\alpha(f)$ of f with respect to π is defined by

$$\alpha : \mathbb{F}^{(m)} \to (\mathbb{N} \cup \{\infty\})$$

with

$$\alpha(f) = \min\{j|\ f|_{z_{i_j}=k} \neq f|_{z_{i_j}=l} \text{ with } l, k \in \{0, 1, \dots, m-1\} \text{ and } l \neq k\}$$

where

$$\min \emptyset = \infty,$$

i.e., $\alpha(f)$ is the index of the first essential variable or infinity if f is constant.

The *index* of an equivalence class $[f]_\equiv$ is defined by

$$\alpha : \mathbb{F}^{(m)}/\equiv \longrightarrow (\mathbb{N} \cup \{\infty\})$$

with

$$\alpha([f]_\equiv) = \max_{g \in [f]_\equiv} \alpha(g),$$

i.e., the maximum index of all functions in $[f]_\equiv$. □

As seen in Example 3.2.10, further restrictions for the selection function are required. The reason why the first selection function $\mathcal{S}$ failed is that it chooses the representative $g \in [f]_\equiv$ which has a lower index than the other function in the equivalence class $[f]_\equiv$, i.e., $\alpha(g) < \alpha([f]_\equiv)$. To ensure canonicity for HOWDDs we require that $\mathcal{S}$ chooses a function with the highest index out of every equivalence class.

Definition 3.2.12 (Selection Function)**.** A selection function $\mathcal{S} : \mathbb{F}^{(m)}/\equiv \longrightarrow \mathbb{F}^{(m)}$ is defined as for OWDDs (in Section 3.1.1) with the additional condition that:

$$\alpha(\mathcal{S}(F)) = \alpha(F) \text{ for all } F \in \mathbb{F}^{(m)}/\equiv .$$

□

Remind that, for $f \in \mathbb{F}$, we simply write $\mathcal{S}(f)$ rather than $\mathcal{S}([f]_\equiv)$.

In the case of the Shannon expansion on ordinary BDDs, the represented function f_v of an inner node v has a lower index than $f_{succ_\xi(v)}$ $\forall \xi \in \{0, 1, \ldots, m-1\}$. This can be generalized for HOWDDs with a selection function $\mathcal{S}$ as defined above.

Lemma 3.2.13. *Let $\pi = (z_1, \ldots, z_n)$ be the given variable ordering, $f \in \mathbb{F}^{(m)}$ a non-constant function, $z = z_{\alpha(f)}$ the first essential variable of f and $\mathcal{S}$ defined as above. Then,*

$$\alpha(\mathcal{S}([f|_{z=\xi}]_\equiv)) > \alpha(f)$$

for all $\xi \in \{0, 1, \ldots, m-1\}$.

Proof. It is obvious that the cofactor $f|_{z=\xi}$ has a higher index than f.

$$\alpha(f|_{z=\xi}) > \alpha(f) \tag{3.1}$$

The cofactor is contained in its equivalence class.

$$f|_{z=\xi} \in [f|_{z=\xi}]_\equiv \tag{3.2}$$

This, together with Definition 3.2.11, yields that the index of the equivalence class is greater or equal to the index of the function itself.

$$\alpha([f|_{z=\xi}]_\equiv) \geq \alpha(f|_{z=\xi}) \tag{3.3}$$

Definition 3.2.12 implies that the index of the equivalence class is equal to the index of the selected function.

$$\alpha([f|_{z=\xi}]_{\equiv}) = \alpha(\mathcal{S}([f|_{z=\xi}]_{\equiv})) \tag{3.4}$$

From 3.3 and 3.4 we get:

$$\alpha(\mathcal{S}([f|_{z=\xi}]_{\equiv})) \geq \underbrace{\alpha(f|_{z=\xi}) > \alpha(f)}_{3.1} \tag{3.5}$$

□

With definition of the selection function for HOWDDs and the result from Lemma 3.2.13, it is possible to implement the same concepts to ensure canonicity as for OWDDs.

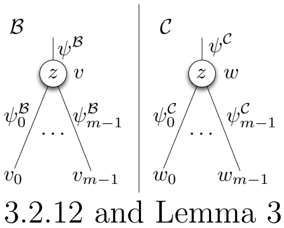
Definition 3.2.14 ($\mathcal{S}$-reduced HOWDDs)**.** A π-HOWDD $\mathcal{B}$ is called $\mathcal{S}$-reduced if $\mathcal{B}$ is reduced and $f_v = \mathcal{S}(f_v)$ for all nodes v in $\mathcal{B}$. □

In contrast to OWDDs, HOWDDs do not provide a weak canonicity for all instances (compare Theorem 3.1.7). This is due to the fact that there is in general no 1-1-correspondence between the cofactors and the nodes. If there is a direct correspondence between cofactors and nodes (e.g., all OWDD instances represented as HOWDD instances) the same property can be proven for those special HOWDD instances.

Theorem 3.2.15. [**Canonicity of $\mathcal{S}$-reduced HOWDDs**] Let π be a variable ordering, $\mathcal{S}$ a selection function and let $\mathcal{B}$, $\mathcal{C}$ be $\mathcal{S}$-reduced π-HOWDDs with $f_{\mathcal{B}} = f_{\mathcal{C}}$. Then, $\mathcal{B}$ and $\mathcal{C}$ are isomorphic.

Proof. Our argument is by induction on the height of the function. The height of function f is the number h of possible lower variables according to first essential variable index of $\mathcal{S}(f)$ and π plus one, i.e., $h = |\mathcal{Z}| - \min\{\alpha(S(f)), |Z| + 1\} + 1$. If $h = 0$ then $f_{\mathcal{B}} = f_{\mathcal{C}}$ is constant. Let c be the value of $f_{\mathcal{B}} = f_{\mathcal{C}}$ and $c' = \mathcal{S}(c)$. Then, the root of $\mathcal{B}$ and $\mathcal{C}$ consists of a terminal node labelled c' together with the unique transformation $\psi \in \Psi_i$ such that $f_{\mathcal{B}} \in I\!F_i$ and $\psi(c') = c$.

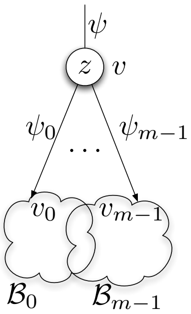
In the induction step, the root nodes of $\mathcal{B}$ and $\mathcal{C}$ are inner nodes, say z-nodes. Then, z is the first essential variable in $S(f_{\mathcal{B}}) = S(f_{\mathcal{C}})$ according to the ordering π. Let $r_{\mathcal{B}} = \langle \psi^{\mathcal{B}}, v \rangle$ be the root of $\mathcal{B}$ and $r_{\mathcal{C}} = \langle \psi^{\mathcal{C}}, w \rangle$ the root of $\mathcal{C}$.



Then, $\psi^{\mathcal{B}}(f_v) = f_{\mathcal{B}} = f_{\mathcal{C}} = \psi^{\mathcal{C}}(f_w)$. Thus, $f_v \equiv f_w$. As $\mathcal{B}$ and $\mathcal{C}$ are $\mathcal{S}$-reduced we have $f_v = f_w$ and $\psi^{\mathcal{B}} = \psi^{\mathcal{C}}$ (by condition (2) of Ψ, cf. Notation 3.2.2). For $\xi \in \{0, 1, \dots, m-1\}$, let $v_\xi = succ_\xi(v)$, $w_\xi = succ_\xi(w)$, $\Psi_\xi(v) = \psi_\xi^{\mathcal{B}}$, $\Psi_\xi(w) = \psi_\xi^{\mathcal{C}}$. Then, $\psi_\xi^{\mathcal{B}}(f_{v_\xi}) = f_v|_{z=\xi} = f_w|_{z=\xi} = \psi_\xi^{\mathcal{C}}(f_{w_\xi})$ (by Definition 3.2.12 and Lemma 3.2.13). Hence, $f_{v_\xi} \equiv f_{w_\xi}$. Again, as $\mathcal{B}$ and $\mathcal{C}$ are $\mathcal{S}$-reduced, we get $f_{v_\xi} = f_{w_\xi}$ and $\psi_\xi^{\mathcal{B}} = \psi_\xi^{\mathcal{C}}$ (by the conditions for Ψ_i). By induction hypothesis, the sub-HOWDD with root nodes v_ξ and w_ξ are isomorphic. Hence, $\mathcal{B}$ and $\mathcal{C}$ are isomorphic. □

Theorem 3.2.16. [Universality and uniqueness of $\mathcal{S}$-reduced HOWDDs] Let π be a variable ordering, $\mathcal{S}$ a selection function and $f : \mathit{Eval}(\mathcal{Z}_m) \to I\!\!K$. Then, there exists a unique $\mathcal{S}$-reduced π-HOWDDs $\mathcal{B}$ with $f = f_{\mathcal{B}}$. (Uniqueness is up to isomorphism.)

Proof. The need remains to provide proof for the existence of a $\mathcal{S}$-reduced π-HOWDD for f. The construction is by induction on the height of the function. The height of function f is the number h of possible lower variables according to first essential variable index of $\mathcal{S}(f)$ and π plus one, i.e., $h = |\mathcal{Z}| - \min\{\alpha(S(f)), |Z|+1\}+1$. If $h = 0$ then f is constant, i.e., $\alpha(\mathcal{S}(f)) = \infty$. Let $c = \mathcal{S}(f)$. There exists a $\psi \in \Psi_i$ with $f \in I\!\!F_i^{(m)}$ and $\psi(c) = f$. Thus, we may use a HOWDD consisting of the root $\langle \psi, v \rangle$ where v is a terminal node v labelled with c. In the induction step, we assume that f is not constant.



Let $g = \mathcal{S}(f)$ and $f = \psi(g)$ where $\psi \in \Psi_i$ if $g \in I\!\!F_i$. Let z be the first essential variable of g according to π. For $\xi \in \{0, 1, \dots, m-1\}$, let $g_\xi = \mathcal{S}(g|_{z=\xi})$ and $g|_{z=\xi} = \psi_\xi(g_\xi)$ where $\psi_\xi \in \Psi_i$ if $g|_{z=\xi} \in I\!\!F_i^{(m)}$. Definitions 3.2.11 and 3.2.12 ensure that $\alpha(S(g|_{z=\xi})) = \alpha([g|_{z=\xi}]_\equiv)$. This together with Lemma 3.2.13 implies that $\alpha([g|_{z=\xi}]_\equiv) = \alpha(S(g|_{z=\xi})) = \alpha(g_\xi) > \alpha(g)$ and thus by induction hypothesis there exist $\mathcal{S}$-reduced π-HOWDDs $\mathcal{B}_0, \dots, \mathcal{B}_{m-1}$ for $g_0, g_1, \dots, g_{m-1}$ which are of lower height. We may assume w.l.o.g. that $\mathcal{B}_0, \mathcal{B}_1, \dots, \mathcal{B}_{m-1}$ share the same nodes for common sub-functions. More precisely, we may assume that if v_i is a node in $\mathcal{B}_i$ and v_j a node in $\mathcal{B}_j$ such that $f_{v_i} = f_{v_j}$ then $v_i = v_j$[11]. (Otherwise the nodes in $\mathcal{B}_j$ can be renamed as there is an isomorphism between the sub-HOWDDs with root nodes $v_0, v_1, \dots, v_{m-1}$, cf. Theorem 3.2.15.) We then may compose $\mathcal{B}_0, \mathcal{B}_1, \dots, \mathcal{B}_{m-1}$ to a $\mathcal{S}$-reduced π-HOWDD for f as shown in the picture on the left. □

3.2.3 Examples

The HOWDD framework identifies the minimal set of requirements needed to guarantee canonicity and uniqueness. These observations influenced the design of JINC, which is described in Chapter 4.

It also eases the development of new OBDD variants. Chapter 5 shows how the development process of a new variant. It starts with the conceptional idea and finishes with its implementation.

This section introduces some further ideas for new variants which can be expressed as instances of HOWDDs.

Input inverters [102] cannot be expressed as an OWDD instance but as a HOWDD instance. A MDD variant with input inverters could be defined as:

- **Transformation function:** $\varphi_{\mathcal{I}} = \{\overline{x}/\overline{g}\}$ with $\overline{g} = (m - x_1 - 1, \dots, m - x_n - 1)$

[11] with $i, j \in \{0, 1, \dots, m-1\}$ and $i \neq j$

- **Partitions:** $\mathbb{F}_1 = \{f | f \in \mathbb{F} \text{ and } \varphi_{\mathcal{I}} \circ f = f\}$ and $I\!\!F_2 = I\!\!F \backslash I\!\!F_1$

It is also possible implement any kind of input permutation for MDD variants. Toggling Algebraic Decision Diagrams (TADDs) which are defined in Chapter 5 are a special instance of input permutations.

Another example is to enrich MDDs with edge weights as in the case of FEVBDDs.

Chapter 4 JINC – Basic Concept

The observations of similarities of different OBDD variants have made it possible to define a general framework. The same observations can help to implement an efficient OBDD library. We will use the principles of HOWDDs to define data structures that are common for all OBDD variants. The alternative implementation of a BDD package based on multi-operand synthesis [51] stands in no contrast with this design because in JINC each operator has its own efficient implementation so that the MORE operator could also be implemented besides the existing algorithms. The idea of implementing the HOWDD concept is more general as it yields the basis for all kinds of algorithms. The HOWDD concept is the minimal set of requirements that is needed to develop common data structures which can be used for a wide range of variants. Besides the theoretical approach, we will also discuss how a new HOWDD instance can be implemented (see Chapter 5). The data structures defined with the concept of HOWDDs in mind can also be used for non-HOWDD variants. In this case it is not guaranteed that JINC's complete functionality can be used[1]. Information about a non HOWDD variant which is also supported by JINC is illustrated in Section 4.2.2. In Section 6.7 we will discuss a multi-operand synthesis which is superior to the MORE concept.

The description of JINC's concept is two-fold. All relevant concepts of [19], like computed-tables, hash maps, etc. and their modifications to commonly used approaches are described in this chapter and afterwards expanded to the requirements of a multi-threaded environment in Chapter 6. This chapter shows the concept and design of JINC [85]. The single-threaded design is a modification of [88].

4.1 Basic Concept

As JINC is based on the HOWDD concept it is designed to be flexible in almost any way. JINC provides a clean object oriented API [116, 74] and uses state of the art programming techniques [113, 1, 4, 98, 106] to provide an efficient and easy to use OBDD library. JINC makes extensive use of template programming [98, 106] in the context of policy-based design [4] to ensure compile-time optimized data structures and algorithms besides readable, maintainable, safe, correct and compact code.

Figure 4.1 shows the concept behind JINC's function API. The function object provides a feature-rich API and encapsulates details like reference counting, i.e., the object is assignable and takes care of increasing and decreasing the embedded reference counter. The algorithms use the computed-table to increase the performance and the unique-table to find or add nodes. The unique-table uses the memory pool to request new nodes. Every part of the overall concept will be discussed in more detail in the following sections. Details about the implementation concept of algorithms

[1] or requires additional implementation effort

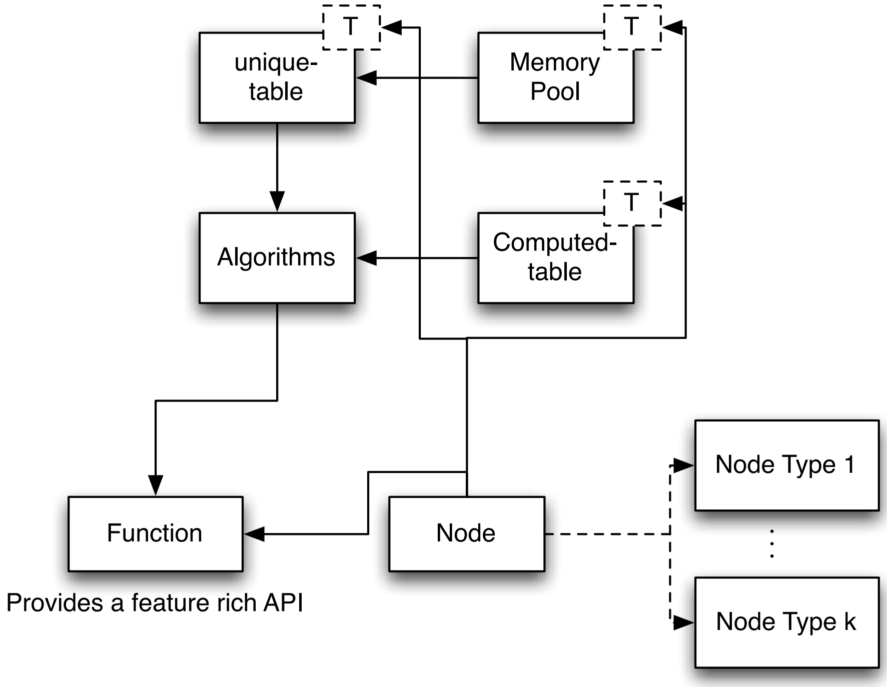


Figure 4.1: The concept of the function API

will be described in Section 5.2.

4.1.1 Data Structures

This section describes the data structures provided by JINC.

4.1.1.1 Nodes

The transformation functions of different HOWDD variants have to be stored in a generic node structure. An ADD node has to store its successors without any further information. A FEVBDD node also has to store the weights for its successors. In HOWDDs these variations are represented by the tuple $< \varphi, v >$ with transformation function φ and node v. Besides the implementation of the different node types, there has to be an implementation of this tuple. In this thesis, we will discuss the case where the transformation functions are non-trivial, i.e., there must be transformation functions besides the identity function *id*. For non-trivial transformations it is not necessary to make use of the tuple. JINC's data structures are optimized to use either the tuple implementation or the node implementation (depend of the used OBDD variant). In the case that the transformation function can be represented by a Boolean value (e.g., negated edges) the node implementation could also be used (see Section 5.2 for an example implementation).

The traditional approach to implement nodes (e.g., in CUDD [109], BuDDy [72],

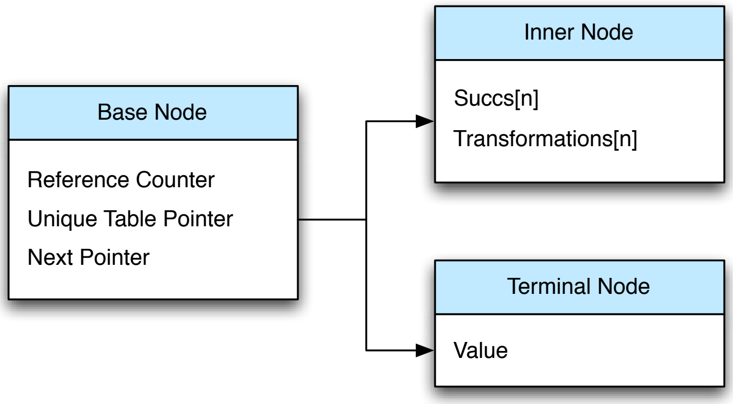


Figure 4.2: The node class hierarchy of JINC

etc.) is that there is no different implementation for different node types, e.g terminal and inner nodes. The HOWDD framework differentiates between function partitions and this is reflected by JINC. JINC follows the approach of a uniform base node implementation which is common for every variant. It contains the reference counter, a pointer to the corresponding unique-table (or zero if it is a terminal node) and a `next` pointer[2]. The different node types are then derived from this base node. These types have to be specified for every new variant. Figure 4.2 illustrates this concept for two node types (inner and terminal node). This approach does not require more memory than the other approaches[3] and eases the development process for new variants.

For variants that uses non-trivial transformation functions, JINC provides a generic tuple implementation to store node and transformation function.

Figure 4.3 shows a typical HOWDD node and its implementation in JINC. Based on the inner node implementation a tuple is created by combining a successor and its corresponding transformation function.

The specific node classes (e.g., inner and terminal node) yield the core of the implementation. The next sections will discuss the data structures provided by JINC to implement a HOWDD instance based on the node classes.

4.1.1.2 Unique-table

The unique-table is the data structure that ensures uniqueness and reducedness of the HOWDD instance. The unique-table is used to find if a node already exists or if it must be created and initialized. This operation is used very frequently so that an efficient implementation of the unique-table is essential for the overall performance. JINC uses a hash-table with unlimited collision list[4] to ensure fast access to an

[2] the different uses are explained later

[3] In fact, it could reduce the memory usage (see Section 4.1.1.2).

[4] because all elements have to be stored

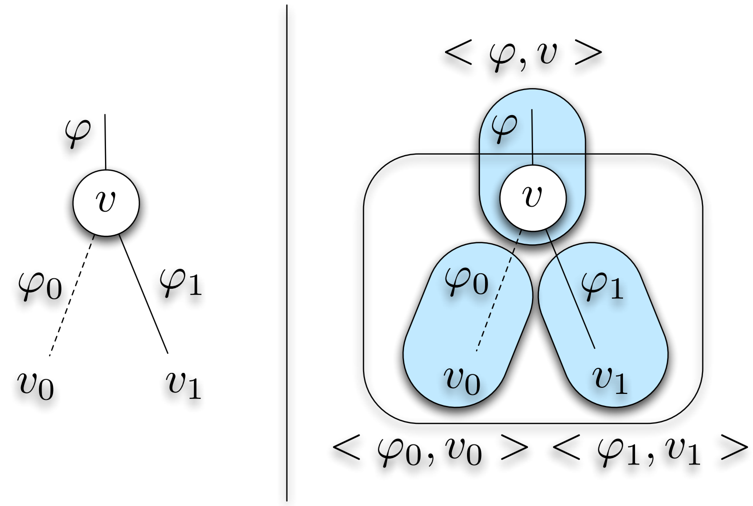


Figure 4.3: A HOWDD node and its implementation in JINC

existing node. The transformation function has to be hashable[5]. The `next` pointer of the base node implementation is used to build the collision list. Figure 4.4 shows how a look-up is performed. At first, the method to get a hash value is called for all tuples[6]. With these numbers a general hash value h is generated. If a node with the requested successor tuples exists it must be an element in the list at slot number h. To find the requested node the list must be traversed.

The presented technique can be improved if the list is sorted. This means that the search can be stopped if the current examined node is greater than the requested one[7]. A further optimization that is implemented in JINC is a flexible hash-table size, i.e., the hash-table is resized if the collision lists get too long. This is a commonly used method to implement an efficient unique-table with small memory overhead.

The concept of HOWDDs support any kind of terminal values. It is for example possible to define a HOWDD variant which operates on strings. For this reason it is important for an efficient library design to support any kind of hash function. JINC provides a unique-table implementation for every node type. This enables to use specialized hash functions for every node type, e.g., CUDD uses the same kind of hash function for inner nodes and terminal nodes, i.e., the value will be handled as two pointers and is therefore not corresponding to the value of the node but to the IEEE representation [89]. This approach uses the assumption that two pointers have the same size as one double value. If two pointers are bigger than one double value it could lead to uninitialized memory. The usage of one implementation of terminal and inner node leads to an overhead of memory per terminal node[8]. Another disadvantage is that there is no control (other than the normalized IEEE

[5] usually this has to be implemented for every new kind of transformation functions

[6] as JINC's unique-table can handle any number of successors

[7] The order of a node is completely defined by its successor tuples.

[8] the same is true for systems if two pointers are smaller than one double value

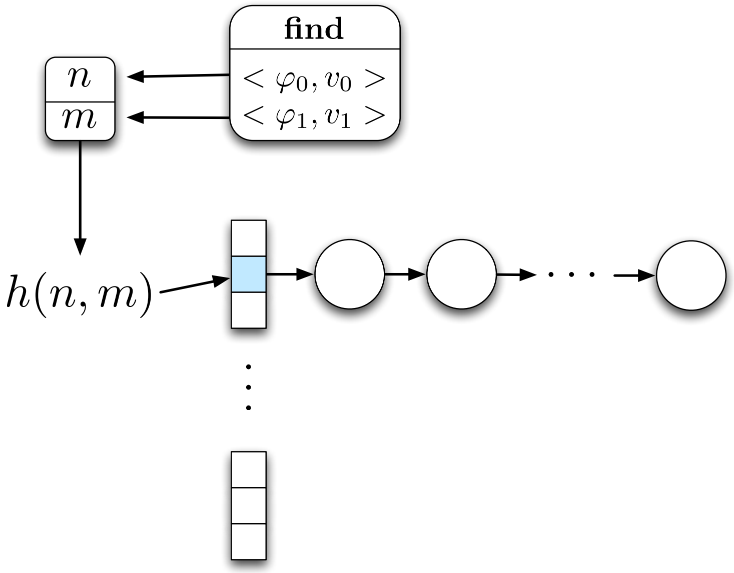


Figure 4.4: A unique-table look-up

representation) how the hash value is generated. JINC's approach can treat all kinds of node types and has full control over the hashing mechanism.

4.1.1.3 Variable Ordering

The variable ordering is implemented and optimized in JINC for a large number of variables. Variables can be inserted and removed at any point of the ordering. Common variable order implementations support the adherence of variables at the end in constant time. JINC's idea is to support adherence at the beginning and the end in constant time while reducing the needed copy operations for insertion and deletion. This technique uses an array of fixed size which will be resized if no free slots are available. The overhead of the resize operation compared to a linked list implementation is in practice negligible as the initial array size could be appropriately chosen to avoid resizing. The reason to choose an array implementation over a linked list implementation is due to the fact that the most frequent operation of a variable ordering is to get the variable level. In a linked list implementation this operation requires $O(n)$ steps compared to $O(1)$ with the array implementation.

Figure 4.5 shows how the variable ordering is implemented in JINC. The base class is the variable. It consists of the name of the variable and a pointer to its unique-table[9]. The pointer to its group will be described later. The variable ordering itself is implemented as a large array. The problem of inserting (or removing) a variable at any place of the ordering is that all other variables have to be copied and the level information (inside the unique-table) must be updated. The idea behind JINC's approach is that inserting (or removing) variables at the beginning or end

[9] Every variable gets its own unique-table to increase performance.

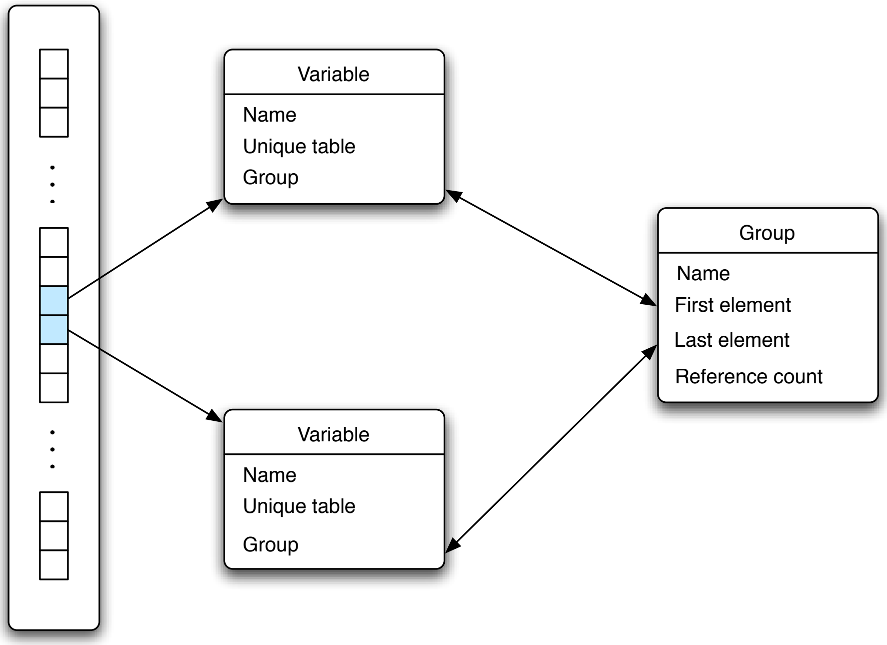


Figure 4.5: The variable ordering implemented in JINC

of the variable ordering can be performed in constant time. For that reason, a skip value[10] is also stored in the variable ordering. The approach for prepending variables in constant time is to reserve additional slots for variables at the beginning of the variable ordering. Whenever a variable is prepended, a free slot can be used. Additionally, the skip value is decreased by one. The real level of a variable is calculated by subtracting the skip value from the slot position of the variable. Another positive effect of this approach is that the removal of a variable needs at most $\frac{n}{2}$ copy operations, compared to n copies required without skip values.

Usual OBDD packages just provide the appending of variables in constant time. [117] shows one application where inserting at the beginning increases the performance. This application also uses a lot of variables, which is uncommon for OBDDs. Therefore, JINC can be configured to deal with a great number of variables.

The insertion of variables inside the variable ordering also benefits from this concept as it uses at most $\frac{n}{2}$ copy operations compared to $n - 1$ copy operations for the traditional concept. Figure 4.8 shows the necessary steps to perform an insertion.

If a variable is prepended to the variable ordering and the skip value is positive the skip value has to be decreased by one. Figure 4.6 illustrates the idea.

Appending a variable to the variable ordering is shown in Figure 4.7.

Removing variables can be performed analogously.

[10] i.e., a counter for the array positions not yet used for some variables

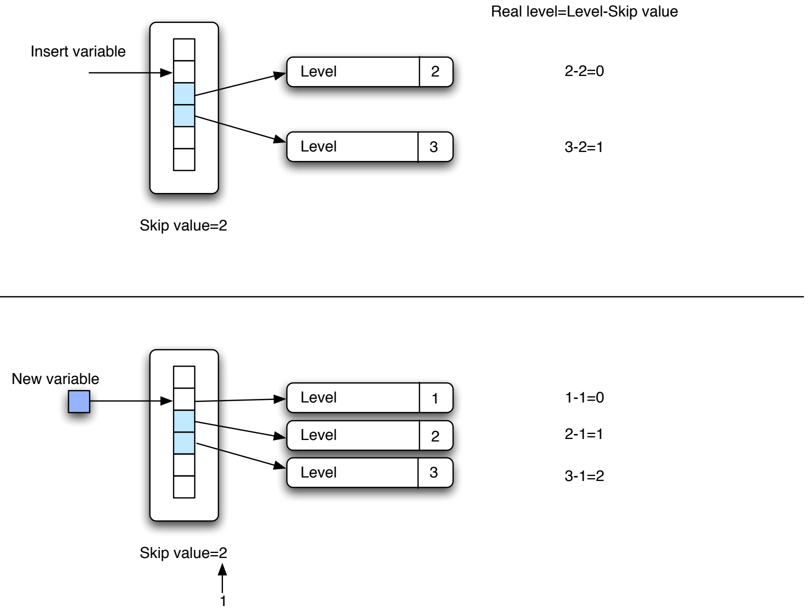


Figure 4.6: Prepending a variable to the variable ordering

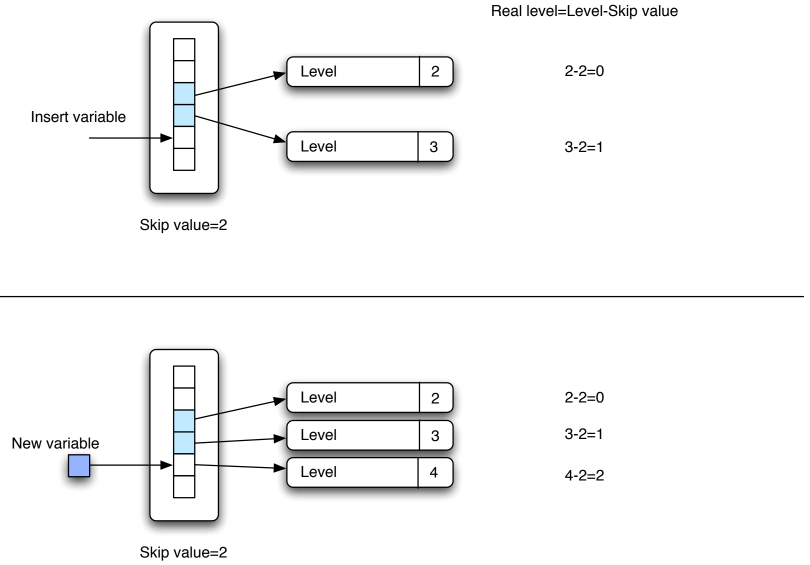


Figure 4.7: Appending a variable to the variable ordering

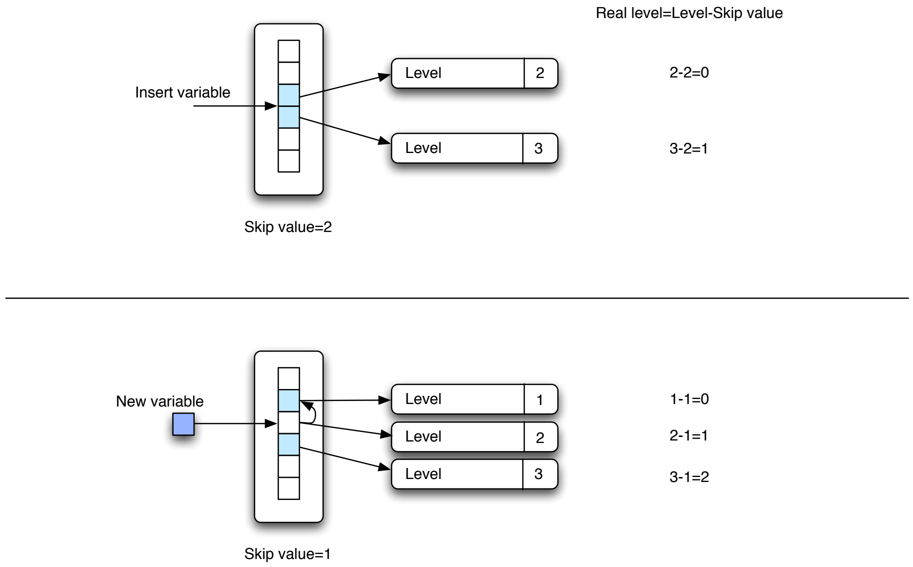


Figure 4.8: Inserting a variable

Groups Grouping variables is also supported by the variable ordering template in JINC. To support this, each variable has a pointer to its group. The group class stores the first element and the last element in the group. Access to variables inside the group is possible through the random access of the variable ordering. With these information, it is possible to manage group operations in a natural way. For instance, if a new variable is inserted, it could easily be determined if it belongs to a group or if it is independent.

4.1.2 Algorithms

JINC provides a framework to implement operations for every HOWDD instance. Besides generic `APPLY` and `ABSTRACT` algorithm implementations (and efficient compile-time specializations for every operator), JINC provides several additional algorithms for e.g., matrix manipulations. With this rich set of functions it is possible to implement symbolic applications in an efficient manner. Examples for projects using JINC are [14, 75, 62, 86].

Figure 4.9 shows the generic algorithm framework. Listing 4.1 shows pseudo code for the generic `APPLY` algorithm. The concept relies on a template implementation that provides the terminal cases implementation and the look-up ID needed for caching (see Section 4.1.2.1 for more details). Every algorithm can be categorized to be commutative or non-commutative. This is illustrated by common objects which influence the caching behavior[11] of the algorithms. The terminal cases have to be implemented for every algorithm. The `findOrAdd` function is the key function in this implementation. It checks for the equality of the tuples (*don't care* semantics of the Shannon's expansion) and calculates the unique representative of the equiv-

[11] the cache look-up for commutative algorithms is automatically optimized

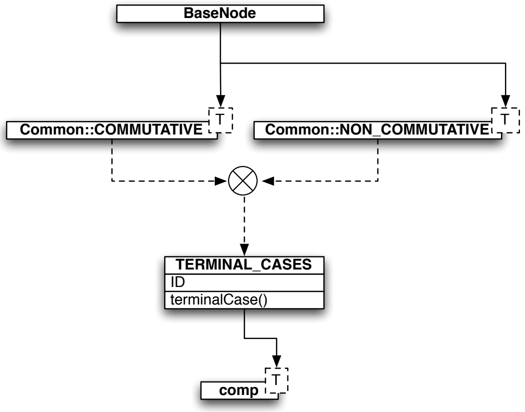
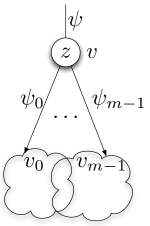


Figure 4.9: JINC's algorithm concept



alence class. It also acts like the selection function $\mathcal{S}$ in the HOWDD framework. The tuples $w_0, \dots, w_{m-1}$ (from Listing 4.1) are translated to the tuples $\langle \psi_0, v_0 \rangle \dots \langle \psi_{m-1}, v_{m-1} \rangle$ (see the picture on the left). This is the parameter normalization phase as in the cases of FEVBDDs, NADDs, SOBDDs with negative edges, etc. With these normalized parameters a unique-table look-up is performed. The returned node v stands for the unique representative function. The result is expressed by the tuple $\langle \psi, v \rangle$ (or `node` as in Listing 4.1). It follows the same principles as described in Theorem 3.2.15. The algorithm framework eases the development but cannot be presented here in all details as e.g., the selection function and cofactor calculation is variant specific. Section 5.2.2 illustrates JINC's algorithm concept in a concrete implementation context.

The idea behind this framework is that algorithms like e.g., addition, multiplication, less, greater, logical and, logical or, etc. can be implemented by providing one specialized implementation of Listing 4.1[12] and terminal case template implementations for every algorithm. Other algorithms like e.g., cofactor calculation have a different recursive structure and thus cannot be represented in this framework. As a consequence these algorithms have to be implemented as in other OBDD libraries.

JINC uses the same technique for the `ABSTRACT` algorithm, i.e., by providing a terminal case template for plus, the sum algorithm can be used as it is derived from it.

4.1.2.1 Computed-table

Computed-tables [19] have an dramatic impact on the run-time behavior of manipulating algorithms. Usually, we measure the complexity of OBDD operations in the number of nodes. This is often reasonable because the size of the resulting OBDD is limited to $|\mathcal{B}| \cdot |\mathcal{C}|$ if $\mathcal{B}$ and $\mathcal{C}$ are the corresponding OBDDs. To measure the

[12] the details for cofactor calculation and normalization of the tuples differ from variant to variant so that JINC could only support the framework

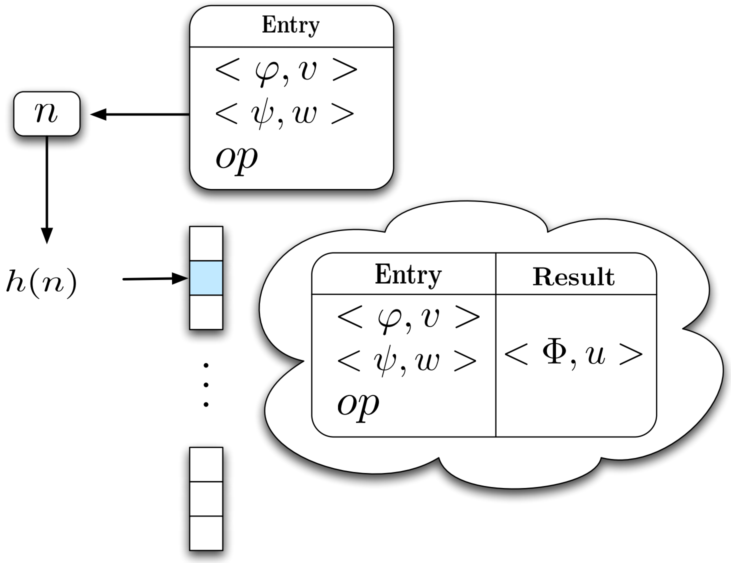


Figure 4.10: Look-up in the computed-table

run-time complexity in a non-theoretical environment, it is reasonable to count the number of possible paths in all involved OBDDs. From the theoretical point of view, both approaches are equal because every request is calculated only once. This can be accomplished by using perfect computed-tables, although it would increase the memory usage significantly and the advantage of a compact representation would be annihilated. For this reason, a dynamic hash-table (similar to the unique-table in Section 4.1.1.2) is used. The difference between this hash-table and the one used for the unique-table is that the size of the collision list is limited. In JINC's case, it is limited to one to reduce the look-up time. The concept of a hash-table look-up is illustrated in Figure 4.10. The parameters for an operation invocation are packed inside an entry class. This class must provide a method to generate a number that is used for hashing. An advantage of this abstraction layer is, that the computed-table template can be used for any kind of operand. To handle different operands, a new entry class has to be implemented. Another advantage of this casing is that the parameters can be normalized to increase the number of computed-table hits. The used techniques simplify the development of OBDD operations and provide an efficient and easy way to use the framework for computed-tables.

4.1.3 Iterators

JINC provides an iterator class which can be used similar to the Standard Template Library (STL) iterators [20]. The concept is based on a generic `TraversalHelper`.

JINC's iterator concept bases on the implementation of the `TraversalHelper`. The `TraversalHelper` is used to identify the paths of a HOWDD function. A path represents the way from the root node to a terminal node. As HOWDDs support arbitrary branch ranges a `Path` class must represent this. JINC's `Path` class is im-

```
template <typename T>
NodeWeightTuple comp(NodeWeightTuple v1, NodeWeightTuple v2){
  if(NodeWeightTuple comp=T::terminalCase(v1,v2)) return comp;

  T::cacheOpt(v1,v2);

  if(NodeWeightTuple comp=computedTable.find(Common::calcPtr(v1,T::ID),v2))
    return comp;

  if(v1.getLevel()<=v2.getRealLevel()){
    minTable=v1.getUniqueTable();
    if(v2.getUniqueTable()==minTable){
      w0=comp<T>(v1.getSucc(0),v2.getSucc(0));
      w1=comp<T>(v1.getSucc(1),v2.getSucc(1));
      ...
    } else {
      w0=comp<T>(v1.getSucc(0),v2);
      w1=comp<T>(v1.getSucc(1),v2);
      ...
    }
  } else {
    minTable=v2.getUniqueTable();
    w0=comp<T>(v1,v2.getSucc(0));
    w1=comp<T>(v1,v2.getSucc(1));
    ...
  }

  NodeWeightTuple node=findOrAdd(minTable,w0,w1,...);

  computedTable.insert(Common::calcPtr(v1,T::ID),v2,node);

  return node;
}
```

Listing 4.1: APPLY for HOWDD instances

```
template <>
struct TraversalHelper<BaseNode>{
  TraversalHelper(const BaseNode* n);
  TraversalHelper getSucc(const unsigned short succX) const;
  const BaseNode* getSuccPtr(const unsigned short succX) const;
  bool isDrain() const;
  double getValue() const;
  const BaseNode* getPtr() const;
  VariableIndexType getLevel() const;

  static const unsigned short N=0;
};
```

Listing 4.2: TraversalHelper

plemented as a template which depends on an integer template argument specifying the branch range m. `Path` represents the assignments of all variables. The assignment of each variable can be set to any number ranging from 0 to $m-1$ or specified as *don't care*.

This `Path` class can also be used to construct the representation of a path into a HOWDD instance. Each HOWDD variant implementation in JINC provides a conversion from `Path` to a HOWDD function.

JINC also provides an `Iterator` class which can be used to iterate overall paths of a HOWDD function. This functionality is provided for every HOWDD instance which implements the specialization of the `TraversalHelper` class. Listing 4.2 shows the interface which has to be implemented. The `getSucc()` method is used within the `Iterator` template class to provide access to the logical paths. The `getSuccPtr()` method is used for size calculation (as transformation functions can be ignored for size calculation). The static variable N specifies the branch range which is respected during iterator calculation. Note that this approach utilizes the idea of the Shannon's expansion as every call to `getSucc()` represents an assignment of the variable. All non-appearing variables are handled as *don't cares*. For example for zero-suppressed variants the algorithms working with the `TraversalHelper` have to be modified. This is because non-appearing variables which are zero-assigned and *don't cares*, where all successor pointers are pointing to the same node, must be handled differently.

Each HOWDD instance which implements a specialized `TraversalHelper` class provides methods to access the first path of a HOWDD function. The `Iterator` class implements `operator++()` and thus access to all paths of a HOWDD function. Similar to the STL iterators, `end()` is provided. Listing 4.3 shows the iterator usage.

```
BDDFunction::iterator it=function.begin();
while(it!=function.end()){
  ++it;
}
```

Listing 4.3: Iterator Usage

JINC also enriches this concept by an generic `Assignment_Iterator` class which iterates over all paths and replaces all *don't care* symbols with an concrete assignment. This is implemented on top of the `Iterator` class and needs no modifications. Figure 4.11 illustrates the complete iterator concept of JINC.

4.1.4 Memory Management

A crucial feature of an OBDD package is the memory management. All above mentioned techniques have minor influence on the performance. The heaviest influence on the performance has the garbage collection and efficient computed-tables. In a non-theoretical environment it is also important to have fast access to memory without any memory fragmentation.

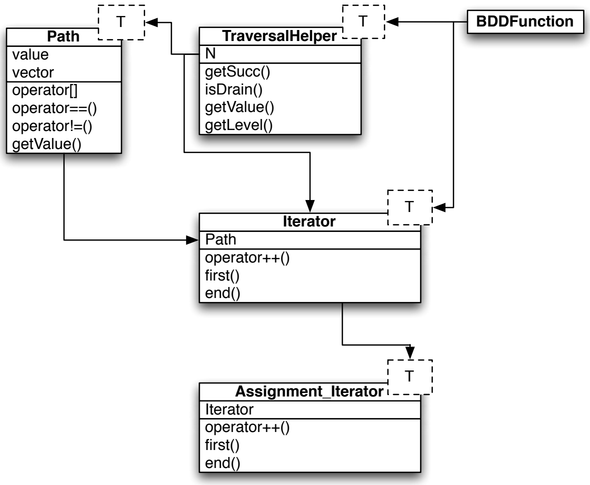


Figure 4.11: JINC's iterator concept

4.1.4.1 Memory Fragmentation

Allocating and freeing memory can cause memory fragmentation. The operating system does not store all references to freed memory. After a short time it will give the user a new block of memory instead of reusing old memory.
We solved this problem with a 'memory pool'. A memory pool allocates a lot of memory at once and enqueues the allocated objects. The memory pool uses the linked objects to reuse old memory. Whenever memory is freed it will be traced back to the memory pool. JINC uses the `next` pointer because it is not needed for a deleted node anymore. As a result, no additional memory is used to prevent memory fragmentation.

4.1.4.2 Garbage collection

Currently there is no mechanism to decide whether a node can be deleted or not. The commonly used technique in OBDD packages is the reference counter in which every node keeps track of the number of predecessors. Figure 4.12 shows an OBDD with the reference counter values for every node. Note that in this example two functions are represented. Every root node of a function increases the reference counter. JINC uses a function class to implement a garbage collection. This function class increases and decreases the reference counter of a node whenever a function is added or removed. It also covers the access to the nodes. The function class provides all operands that are needed to work with OBDDs.

Delayed garbage collection The computed-table is important for the efficiency of a OBDD package. As mentioned above, deleted nodes can be restored and used in another context. For this reason, all entries that refer to deleted nodes must be removed from the computed-tables. The costs for this operation would almost

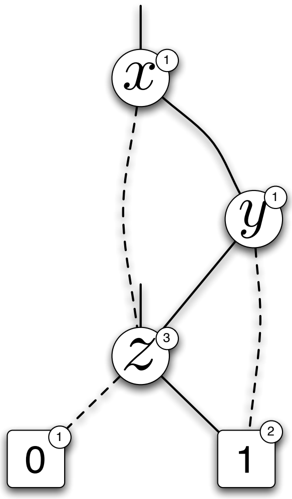


Figure 4.12: Reference counting

eliminate the advantages of computed-tables.

JINC's solution to solve this is a delayed garbage collection. Whenever a node and its corresponding subtree can be deleted, JINC marks the root node as deletable. When the number of deletable nodes reaches a configurable number, the garbage collection takes place. The approach of just marking the root nodes is different than the commonly used approach to mark all deletable nodes. The advantage of marking all nodes is that it is known how much memory will be freed during the garbage collection. The disadvantage is that every deleting operation requires a traversal of the subtree. The same is true when a as deletable masked node is used again (either through calculation or as a result from the computed-table). Figure 4.13 shows how marking all nodes works for the former example. At first, the root node is marked as deletable. After that step all successor reference counters are decreased. If a successor reference counter reaches zero, it will be marked also and the process will repeat for its subtrees.

JINC's garbage collection requires additional work when the marked nodes have to be removed. It is necessary to start at the highest variable level with marked nodes in it[13]. Every marked node can be deleted. Before that happens the subtree has to be traversed. It is quite similar to the other approach with the only difference that the node can be deleted instead of just marking it. The costs for the garbage collection are slightly higher in this approach. On the other hand, it is much faster to deleted and reuse a node with its subtrees. Practice shows that this kind of operation has more effect on the performance than the slightly increased costs for the garbage collection. Figure 4.14 illustrates the garbage collection process in JINC.

Both approaches require that the unique-table can be traversed to find the marked nodes. As discussed above, the unique-table is implemented as a hash-table. This

[13] Every variable level stores the number of marked nodes to increase the performance of the garbage collection

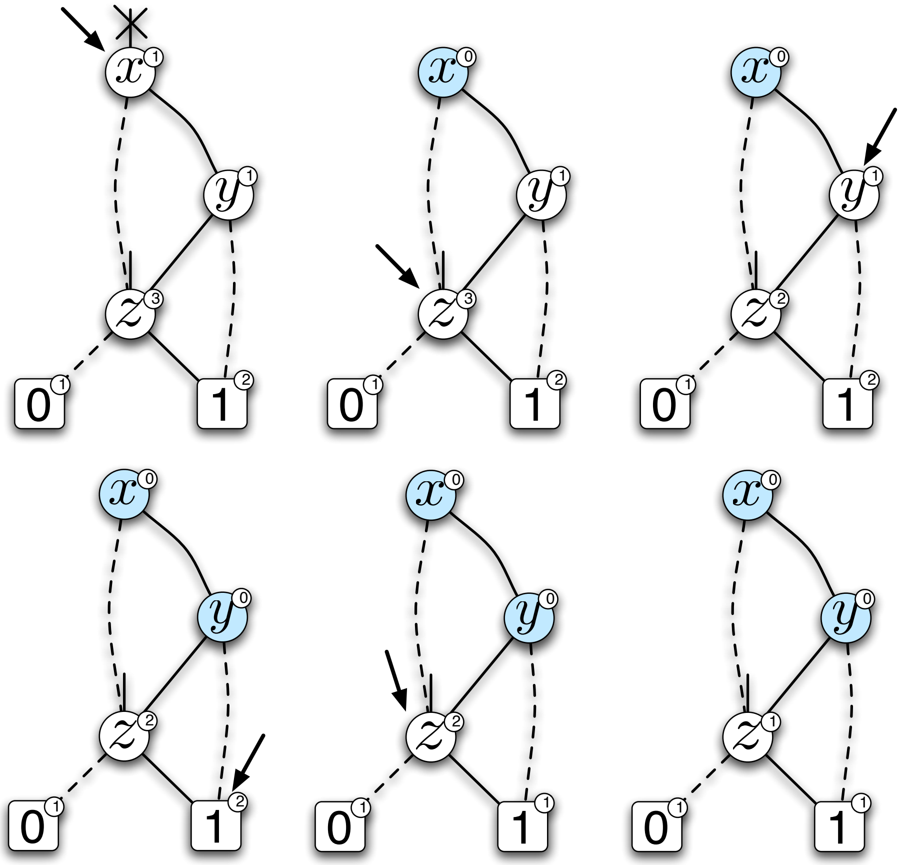


Figure 4.13: Garbage collection with marking all deletable nodes

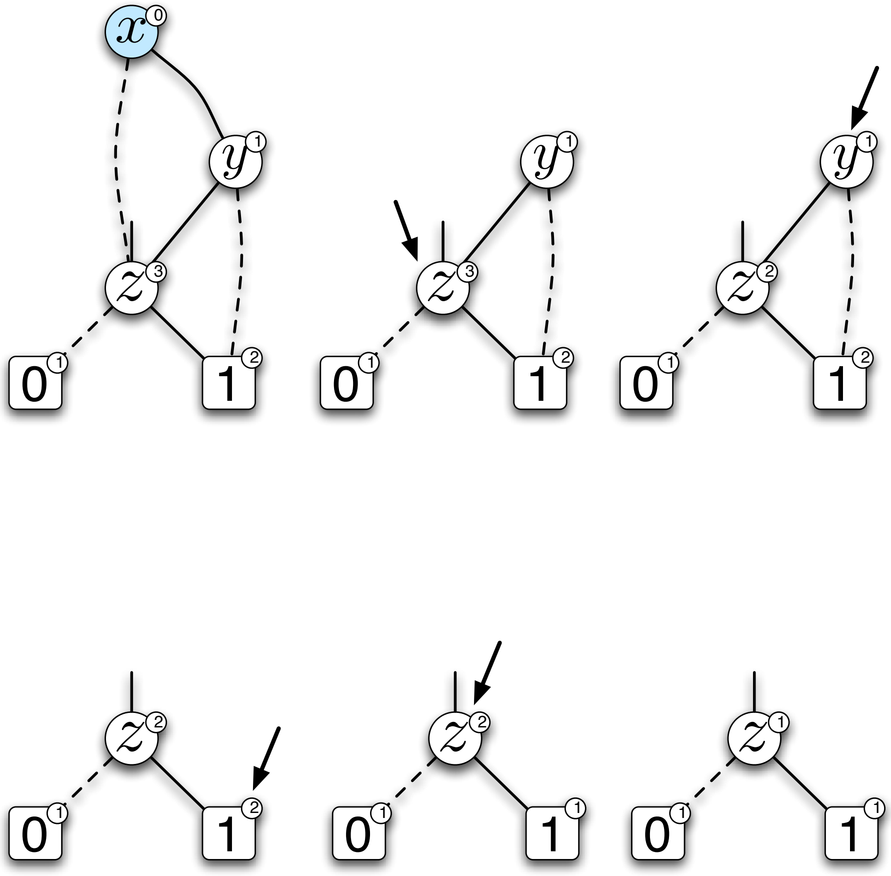


Figure 4.14: The garbage collection of JINC

means that every slot has to be investigated. If the number of empty slots is high this means that the garbage collection is very slow. For that reason, JINC uses a bitmask to easily identify non empty slots. Whenever a slot is filled, it will be masked as non empty and whenever the last node of a slot is removed, it will be masked as empty. The cost for masking the slots can be neglected.

4.1.5 Reordering

Reordering the variable ordering can have a great impact on the size of the represented OBDDs [108]. Searching for an optimal variable ordering is not appropriate for a real world applications [107, 18, 115]. Reordering heuristics like window permutation, sifting, etc. [37, 28, 55, 25, 101, 90, 100, 38, 17] have been developed to find a good variable ordering within a reasonable time. JINC provides a clean template system to implement reordering methods without caring about the used OBDD variant. The system uses the observation that all reordering methods are based on the swap function. JINC derives commonly used functions like the shift of a variable to a given position from the swap function. Every variant that uses the reordering system implements the swap function and all reordering methods can be used immediately without any further work. The variable ordering template provides a swap function for variables so that the reordering template can synchronize the change of the ordering. As mentioned above, the variable ordering template also provides grouping of variables. The reordering system has been designed to handle variables and groups (while keeping the order inside the groups). This means that an algorithm can be used for variables and groups without rewriting the algorithm. Figure 4.15 illustrates the concept behind the reordering system.

The idea behind this new concept is that a `ReorderHelper` provides the generic reordering algorithms with information about number of elements, size of the OBDD and the swap function. This `ReorderHelper` encapsulates all differences[14] so that the reordering algorithms can be used for any kind of element (variables or groups). The `ReorderHelper` itself only provides the methods which are used in the reordering algorithms. The specific behavior for groups or variables are implemented via a policy-based-design [4]. The given strategy encapsulates all implementation details for variables or groups. It is possible to implement an own `ReorderHelper` or strategy if the reordering behavior is different for a new HOWDD variant.

4.2 Further Details

In this section, we will discuss further important details that are used in JINC.

[14] e.g., between variable reordering and group reordering

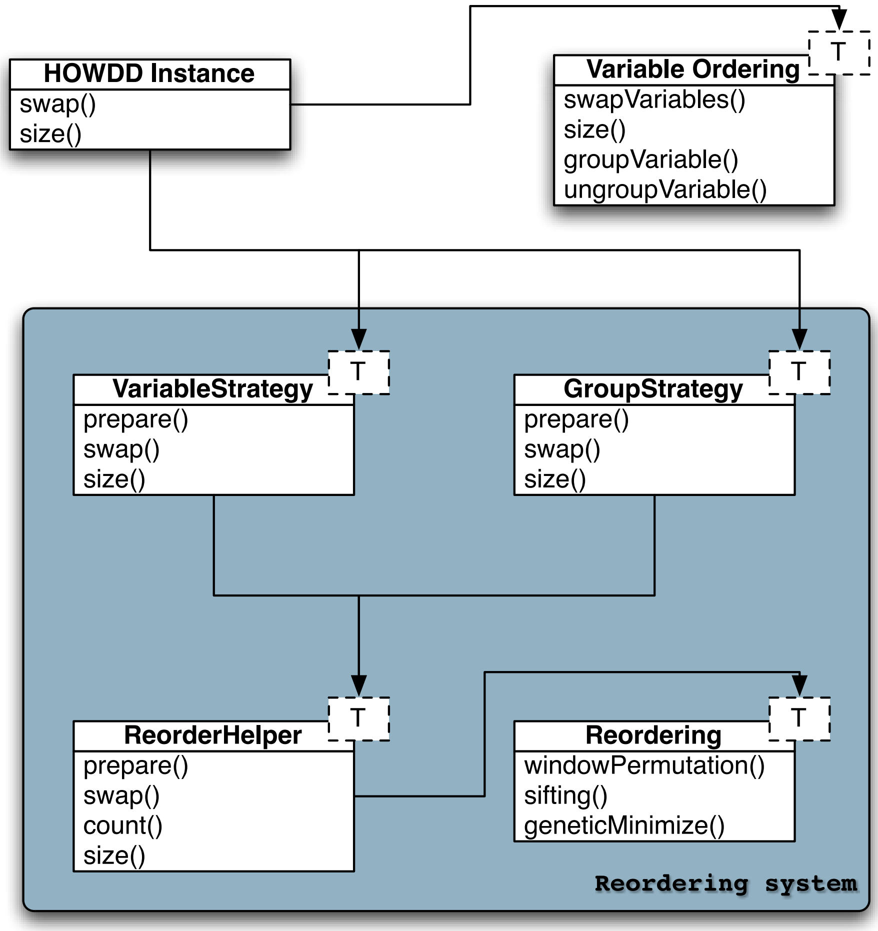


Figure 4.15: Reordering concept

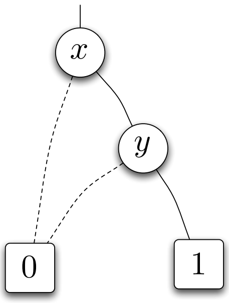


Figure 4.16: A cube set representing $\{x, y\}$

4.2.1 Cube Sets

Many operations on OBDDs (like existential quantification) require a set of variables as parameters. Usual set operations are not well suited to be used with computed-tables. The equality of two sets must be checked efficiently to use computed-tables with this kind of operations. JINC uses cube sets [102][15] to handle sets of variables and to solve the problem of the equality check. Cube sets are a conjunction of the variables that are stored in the set. The concatenation of these variables is stored as an OBDD function and thus the equality check can be performed in constant time. Figure 4.16 shows the cube set representing $\{x, y\}$. This approach also simplifies the handling of variable sets inside the OBDD algorithms, e.g., the test if the variable of the current level is inside the variable set can be done on-the-fly[16].

4.2.2 Zero-suppressed Algebraic Decision Diagrams

As mentioned above JINC also supports OBDD variants with a different expansion rule than the Shannon's expansion. Although this leads to BDD variants that do not met the conditions of HOWDDs, the internal treatment in JINC is similar to the realization of HOWDDS. We will now show how zero-suppressed OBDD variants can be used in a HOWDD based framework like JINC.

Zero-suppressed Algebraic Decision Diagrams (ZADDs) are the adaption of Minato's ZBDDs [105, 56]. All above defined data structures can be used for ZADDs as for any other variant. The iterator concept presented in Section 4.1.3 cannot be used for ZADDs as zero-assigned variables do not appear on the `Path`. A complete different `TraversalHelper` concept is needed to support `Iterators` for zero-suppressed variants. JINC's design is based on the HOWDD concept and therefore there is currently only iterator support for variants based on the Shannon's expansion. Lampka [65, 67] introduced partially shared ZADDs (p-ZADDs) which attempt to avoid the explicit representation of *don't care* variables. The idea is that, besides the nodes of a function, a set of variables is stored. Every variable inside the variable ordering

[15] like many other OBDD packages

[16] Therefore, the variable set must be traversed in parallel to the algorithm – which is equivalent to ordering consistent cofactor building.

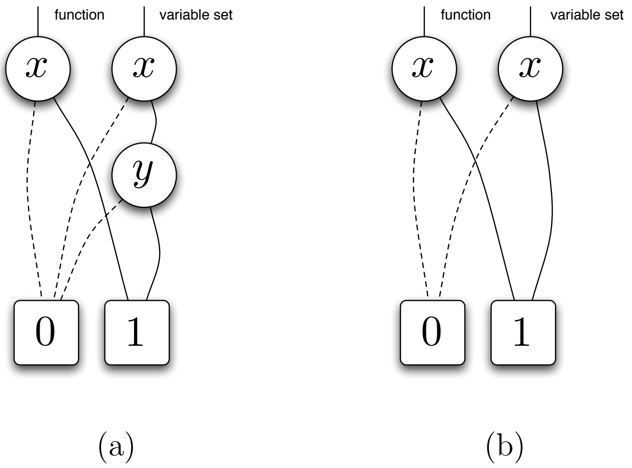


Figure 4.17: Partially shared ZADDs with different variable sets representing the function $x \cdot (1 - y)$ (a) and x (b)

that is not included in the set has to be interpreted as a *don't care* variable[17]. This approach combines the advantages of zero-suppressed variants and Shannon based variants. JINC uses the former introduced cube sets with a standard ZBDD implementation. The cube sets define how an overleaped level has to be interpreted (as a *don't care* or zero-assigned). Figure 4.17 shows one BDD structure with different variable sets. Partially shared ZADD (a) represents the function $x \cdot (1 - y)$. In this case the overleaped level is interpreted as zero-assigned. The other partially shared ZADD (b) represents the function x. In this case the overleaped level is interpreted as *don't care*. Figure 4.17 explains why it is necessary for algorithms on partially shared ZADDs to deal with different variable sets. Different cube sets require a special treatment. Algorithm 1 shows how the APPLY algorithm can be implemented. This implementation does not use the fact that operations on p-ZADDs with equal cube sets can be handled as standard ZADD operations. The main difference to ordinary OBDD algorithms is the calculation of the cofactors. The other difference relies in the usage of variable sets. It is important to understand that ZADDs do not yield a unique data structure if the variable sets are not minimal. The search for a minimal variable set cannot be performed efficiently[18]. Thus, JINC uses the variable set that is the union of both variable sets. In special cases like quantification, JINC eliminates the quantified variables from the variable set[19]. The APPLY algorithm has a completely different recursive structure than the HOWDD APPLY algorithm described in Section 4.1.2. As a consequence, a completely different algorithm framework has been implemented for ZADDs.

[17] Note, the represented function has no node on this level.

[18] the calculation needs a complete traversal of the OBDD structure

[19] because they should be interpreted as *don't care* variables

Algorithm 1 **Partially shared ZADD operations (APPLY)**

Input: Operator $\circ$, tuples $< n_1, S_1 >$ and $< n_2, S_2 >$
Output: Tuple $< n, S >$ representing $f_{<n_1,S_1>} \circ f_{<n_2,S_2>}$

```
if $S_1 = S_2 = \emptyset$ then
  //terminal case
  return $value(n_1) \circ value(n_2)$
end if
$v$ =minimal variable out of $S_1 \cup S_2$
if $v \in S_1$ then
  if $var(n_1) = v$ then
    //Shannon expansion
    $n_{1_0} = succ_0(n_1)$
    $n_{1_1} = succ_1(n_1)$
  else
    //zero-suppressed
    $n_{1_0} = zeroDrain$
    $n_{1_1} = n_1$
  end if
else
  //don't care
  $n_{1_0} = n_1$
  $n_{1_1} = n_1$
end if
if $v \in S_2$ then
  if $var(n_2) = v$ then
    //Shannon expansion
    $n_{2_0} = succ_0(n_2)$
    $n_{2_1} = succ_1(n_2)$
  else
    //zero-suppressed
    $n_{2_0} = zeroDrain$
    $n_{2_1} = n_2$
  end if
else
  //don't care
  $n_{2_0} = n_2$
  $n_{2_1} = n_2$
end if
$r_0 = APPLY(< n_{1_0}, S_1 \backslash \{v\} >, < n_{2_0}, S_2 \backslash \{v\} >)$
$r_1 = APPLY(< n_{1_1}, S_1 \backslash \{v\} >, < n_{2_1}, S_2 \backslash \{v\} >)$
if $r_1 = zeroDrain$ then
  //zero-suppressed
  return $< r_0, S_1 \cup S_2 >$
else
  return $<$ findOrAdd$(v, r_0, r_1), S_1 \cup S_2 >$
end if
```

Chapter 5

JINC – Toggling Algebraic Decision Diagrams

This section introduces a new HOWDDs instance that has been developed with the focus on matrix representation. Afterwards JINC's implementation of this variant will be discussed as an example of the flexibility of JINC's design.

Efficient matrix representation with OBDD variants involves an interleaved variable ordering [30, 43]. The idea behind Toggling Algebraic Decision Diagrams (TADDs) is that these variable pairs are store-space efficient and that renaming of those pairs can be performed in constant time. The renaming of variable pairs is equivalent to matrix transposition. In the context of symbolic reachability computation after each step a renaming operation is performed.

5.1 Definition

The key idea of this new variant is that the variables x_i and y_i are always grouped together. This is a very important requirement for the graph and matrix representation[1]. The approach is a very good example of a HOWDD which cannot be represented with OWDDs. The disadvantage of this approach is that all algorithms have to deal with two level cofactors. We will define a HOWDD with $m = 4$ to combine x_i and y_i[2]. Figure 5.1 illustrates the idea how the two levels are combined to one level.

In the sequel we assume that $m = 4$ and $\pi = (xy_1, \dots, xy_n)$ over $\mathcal{Z}_4 = \{xy_1, \dots, xy_n\}$.

Similar to the former definition we define the transformation function.

[1] used e.g., in the application of model checking

[2] this follows the idea of calculating $2 \cdot x_i + y_i$ to identify the successor

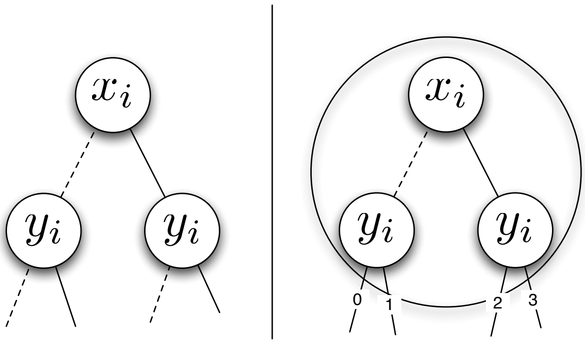


Figure 5.1: Idea of combining two levels into one level

z	$\varphi_{\mathcal{S}}$
0	0
1	2
2	1
3	3

Table 5.1: The multi-valued swap function

Definition 5.1.1 (Swapping Function)**.** Let $\overline{xy} = (xy_1, \ldots, xy_n)$ the variable tuple,

$$g_i = xy_i(2 - xy_i)(3 - xy_i) + \frac{xy_i(xy_i - 1)(3 - xy_i)}{2} + \frac{xy_i(xy_i - 1)(xy_i - 2)}{2}$$

the toggling function and $\overline{g} = (g_1, \ldots, g_n)$ the function tuple.

The swapping function $\varphi_{\mathcal{S}}$ is defined by:

$$\varphi_{\mathcal{S}} = \{\overline{xy}/\overline{g}\}.$$

□

The swapping function $\varphi_{\mathcal{S}}$ is now treated like an input inverter[3] [102]. Table 5.1 illustrates the concept of the swapping function.

The first partition of $\mathbb{F}^{(4)}$ is defined by:

$$\mathbb{F}_1^{(4)} = \{f | f \in \mathbb{F} \text{ and } \varphi_{\mathcal{S}} \circ f = f\}.$$

The second partition holds the rest of $\mathbb{F}$, i.e.,

$$\mathbb{F}_2^{(4)} = \mathbb{F}^{(4)} \backslash \mathbb{F}_1^{(4)}.$$

The transformation sets are defined as follows:

$$\Phi_{\mathbb{F}_1^{(4)}} = \{id\}$$

and

$$\Phi_{\mathbb{F}_2^{(4)}} = \{id, \varphi_{\mathcal{S}}\}.$$

$\varphi_{\mathcal{S}}$ is self inverse and for every function $f \in \mathbb{F}_2^{(4)}$ it holds that $\varphi_{\mathcal{S}} \circ f \in \mathbb{F}_2^{(4)}$. With this observation it is clear that

$$\mathfrak{F}_k^{(4)} = \left\{(\mathbb{F}_1^{(4)}, \Phi_{\mathbb{F}_1^{(4)}}), (\mathbb{F}_2^{(4)}, \Phi_{\mathbb{F}_2^{(4)}})\right\}$$

builds a system (see Notation 3.2.3).

The only freedom for the definition of a selection function exists for the situation illustrated in Figure 5.2. This is solved by defining an order on the functions.

[3] another example of a HOWDD variant that cannot be represented by a OWDDs

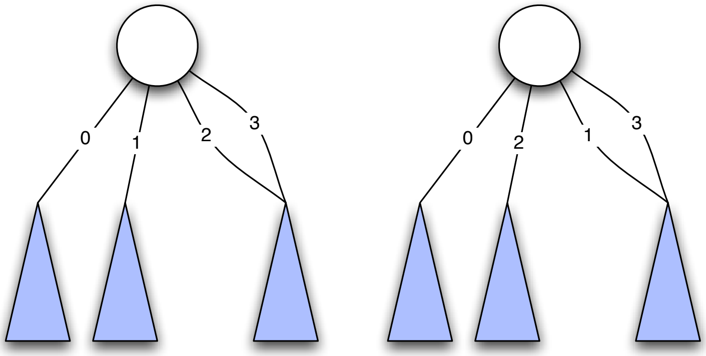


Figure 5.2: Two possible functions to be picked by a selecting function

Definition 5.1.2 (Selection Function)**.** Let $f \in I\!\!K(\mathcal{Z}_4)$ and z the first essential variable[4] of f. The selection function $\mathcal{S}_{\text{TADD}}$ is defined by:

$$\mathcal{S}_{\text{TADD}}(f) = \begin{cases} f & \text{if } f \in I\!\!F_1^{(4)} \\ f & \text{if } f \in I\!\!F_2^{(4)} \text{ and } f|_{z=1} < f|_{z=2}) \\ \varphi_{\mathcal{S}} \circ f & \text{if } f \in I\!\!F_2^{(4)} \text{ and } f|_{z=1} > f|_{z=2} \\ f & \text{if } f \in I\!\!F_2^{(4)},\ f|_{z=1} = f|_{z=2} \text{ and } f|_{z=0} < f|_{z=3} \\ \varphi_{\mathcal{S}} \circ f & \text{otherwise} \end{cases}$$

□

After defining the selection function we now have a new HOWDD instance that performs the swap of neighboring variables in constant time. At the same time, Toggling Algebraic Decision Diagrams (TADDs) provide a unique and canonical form.

5.2 Implementation

We will now focus on some facts that are important for an efficient implementation. All necessary algorithms will be presented in this section. The main difference between this variant and all others mentioned before is that it uses four successor pointers. For every successor pointer a flag is used to identify if the swap function is active or not. Only one Boolean flag is needed because the function is self inverse[5]. An active swap flag changes the way successors are calculated[6]. As mentioned in Section 4.1.1.1 we can use the pure node rather than the tuple implementation to increase the system's performance. The idea relies on the fact that objects are aligned to word sized chunks [24]. That means that the last bit

[4] Remember, that this variable represents all possible combinations of x_i and y_i.

[5] comparable to OBDDs with negative edges

[6] see Table 5.1 for more information

of the memory addresses used in the operating system is always zero. For a compact representation of a node we use the successor pointers to store the Boolean flag[7]. JINC utilizes the `TraversalHelper` approach described in Section 4.1.3 to eliminate most of the occurrences of programming errors accompanied with pointer arithmetic. The `Helper` module defines all methods needed to access the appropriate pointer[8] and the Boolean flag. The `TraversalHelper` can easily be provided with the use of JINC's helper classes. Listing 5.1 shows the complete implementation of the `TraversalHelper` for TADDs.

```
 1 template <>
 2 struct TraversalHelper<TADDBaseNode> {
 3   TraversalHelper(const TADDBaseNode* n) : node(n) {}
 4   TraversalHelper getSucc(const unsigned short succX) const {
 5     if(Common::bitSet(node.ptr)){
 6       TADDBaseNode* succ;
 7       if((succX==1) || (succX==2)){
 8         succ=node->getSucc(3-succX);
 9       } else {
10         succ=node->getSucc(succX);
11       }
12
13       if(Common::SmartAccess<TADDBaseNode>(succ)->isSymmetric()){
14         return TraversalHelper(succ);
15       } else {
16         return TraversalHelper(Common::flipBit(succ));
17       }
18     } else {
19       return TraversalHelper(node->getSucc(succX));
20     }
21   }
22   const TADDBaseNode* getSuccPtr(const unsigned short succX) const {
23     return getSucc(succX).getPtr();
24   }
25   bool isDrain() const {return node->isDrain();}
26   double getValue() const {return node->isDrain()?node->getValue():0.0;}
27   const TADDBaseNode* getPtr() const {return node.getPtr();}
28   VariableIndexType getLevel() const {return node->getLevel();}
29
30   static const unsigned short N=4;
31   Common::SmartAccess<TADDBaseNode> node;
32 };
```

Listing 5.1: TraversalHelper⟨TADD⟩

One of the most important features a HOWDD instance has to provide is that the partition of a function can be easily identified (otherwise the identification phase would be a serious performance issue). Line 13 uses the `isSymmetric()` method which identifies the equivalence class membership of the function represented by this node[9]. The partitions of TADDs can be recognized with the observations that every path of the representation of a $\mathbb{F}_2^{(4)}$ function leads to a $\mathbb{F}_1^{(4)}$ function[10] and that no node representing a $\mathbb{F}_2^{(4)}$ function can be reached by a node representing a $\mathbb{F}_1^{(4)}$ function.

These observations lead to the following criteria to identify whether a non-terminal node v (with first essential variable z) is representing a $\mathbb{F}_1^{(4)}$ function.

(a) $f_v|_{z=1} = f_v|_{z=2}$

(b) For all $\eta \in \{0, 1, 2, 3\}$ holds $f_v|_{zi=\eta} \in \mathbb{F}_1^{(4)}$.

[7] the same technique is used in CUDD for the representation of negated edged

[8] with the template helper object Common::SmartAccess⟨T⟩

[9] The TADD implementation only differentiates between inner and terminal nodes. This differentiation is reasonable because it simplifies the implementation of algorithms. The partitions are not taken as an distinctive feature because it does not simplify the implementation.

[10] constant functions are always $\mathbb{F}_1^{(4)}$ functions

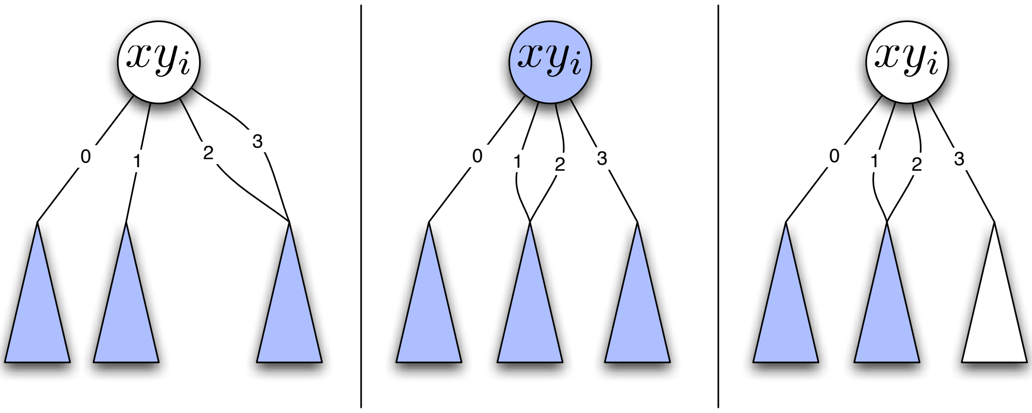


Figure 5.3: How to classify if a node is a $\mathbb{F}_1$ or $\mathbb{F}_2$ node

The first condition means that the node must be $\varphi_{\mathcal{S}}$-invariant. The second condition is a result from the observation that all cofactors have to be $\mathbb{F}_1$ functions. These criteria can be used to implement an efficient `findOrAdd` algorithm[11]. Each node also uses one Boolean flag to identify if it is representing a $\mathbb{F}_1$ function or a $\mathbb{F}_2$ function[12].

Figure 5.3 shows different situations to illustrate the identification process. The marked nodes[13] are representing $\mathbb{F}_1$ functions. The first example shows that this node cannot represent a $\mathbb{F}_1$ function because $succ_1(node) \neq succ_2(node)$. The second example shows the only situation where the node represents a $\mathbb{F}_1$ function ($succ_1(node) = succ_2(node)$ and all sub-trees represent $\mathbb{F}_1$ functions). The third example shows the situation where not all sub-trees represent $\mathbb{F}_1$ functions and therefore the node must represent a $\mathbb{F}_2$ function.

5.2.1 FindOrAdd

The selection function defined in the section above uses a function order to ensure uniqueness. The implementation uses a similar idea. We will define an order on the nodes, e.g., the topological sorting. The basis algorithm to provide a unique data structure is the `findOrAdd` algorithm. The `findOrAdd` algorithm for TADDs is slightly more complex than for ordinary BDD variants. It has to deal with four nodes and toggling information on the edges. The handling of four parameters instead of two increases the number of possible situations to deal with. The possible situations and the results are presented in Figure 5.4. In this example, all cofactors are $\mathbb{F}_2$ functions[14]. The mentioned situations are the cases where the algorithm has to alter the input parameters.

[11] as the identification rules just look at the already created sub-nodes

[12] This is needed to identify the node in constant time. Otherwise a whole traversal would be needed.

[13] or sub-trees

[14] the situation with $\mathbb{F}_1$ functions is handled analogously

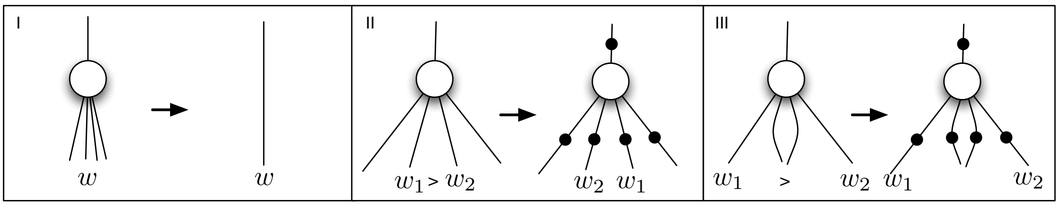


Figure 5.4: Possible input parameters for the `findOrAdd` algorithm and the resulting structure

The `findOrAdd` algorithm is shown in Algorithm 2. The algorithm first checks if the second condition is violated. This information is essential to mark a node as a $\mathbb{F}_1$ or $\mathbb{F}_2$ node. The `toggle` function is used to calculate the right transformation function. If the successor is a $\mathbb{F}_1$ node the transformation function will always be id. In the other case, it calculates $\varphi_S \circ \phi_i$ to adjust the already existing transformation function. The function `setSymm` marks a node to be a $\mathbb{F}_1$ node. By default a node is assumed to be a $\mathbb{F}_2$ node. The different cases are labeled with the corresponding number from Figure 5.4. Case IV[15] handles the cases where the transformation function is always the identity function. There is also a check if the newly created node is a $\mathbb{F}_1$ node.

5.2.2 Algorithms and Operators

The algorithm concept of JINC is efficient and flexible[16]. We will discuss the implementation of a generic `APPLY` algorithm for TADDs, other algorithms (e.g., a generic `ABSTRACT` algorithm) follow the same pattern.

Listing 5.2 shows the implementation of a generic `APPLY` algorithm on TADDs. The template argument of the `APPLY` algorithm provides the handling of terminal cases and the operator ID for computed-table look-ups. It also provides if a cache optimization should be performed. In the case of a commutative operator `T::cacheOpt()` will provide a normalization of the input parameters for a better computed-table hit ratio. In the case of a non-commutative operator the template argument provides an empty implementation which will be eliminated by the compiler and leads to no run-time overhead. Listing 5.3 shows the implementation of the plus operator as an example. The `Helper` module provides base-classes that implement the cache optimization. The plus operator is commutative and thus derived from `Common::COMMUTATIVE<TADDBaseNode>`. Line 10 checks if both arguments are equal. In that particular case JINC provides an optimized one argument plus operator. Figure 5.5 illustrates the algorithm concept on the example of the plus operator. The `APPLY` algorithm in Listing 5.2 does not take pure node pointer as an argument. It uses the `TraversalHelper` to provide convenient methods for implementation (e.g., for successor access). Every pointer will automatically converted to

[15] not shown in Figure 5.4

[16] through policy-based-design [4] which can be seen as a compile-time variant of the strategy pattern

Algorithm 2 findOrAdd

Input: Variable index i and cofactor tuples $< w_0, \phi_0 >$, $< w_1, \phi_1 >$, $< w_2, \phi_2 >$ and $< w_3, \phi_3 >$

Output: Unique tuple $< n, \phi >$

```
symm=areSymmetricCofactors($w_0, w_1, w_2, w_3$)
if $< w_0, \phi_0 >=< w_1, \phi_1 >=< w_2, \phi_2 >=< w_3, \phi_3 >$ then
   $< n, \phi >=< w_0, \phi_0 >$                                    Case I
else if $w_1 > w_2$ then
   $n$ =find or add node on level $i$ with successor tuples
             $< w_0, \text{toggle}(w_0, \phi_0) >,\ < w_2, \text{toggle}(w_2, \phi_2) >$
             $< w_1, \text{toggle}(w_1, \phi_1) >,\ < w_3, \text{toggle}(w_3, \phi_3) >$
   $< n, \phi >=< n, \varphi_{\mathcal{S}} >$                                   Case II
else if $w_1 = w_2$ and $w_0 > w_3$ then
   $n$ =find or add node on level $i$ with successor tuples
             $< w_0, \text{toggle}(w_0, \phi_0) >,\ < w_1, \text{toggle}(w_2, \phi_2) >$
             $< w_1, \text{toggle}(w_1, \phi_1) >,\ < w_3, \text{toggle}(w_3, \phi_3) >$
   if symm=true then
      setSymm($n$)
   end if
   $< n, \phi >=< n, \varphi_{\mathcal{S}} >$                                   Case III
else
   $n$ =find or add node on level $i$ with successor tuples
             $< w_0, \phi_0 >,\ < w_1, \phi_1 >,\ < w_2, \phi_2 >$ and $< w_3, \phi_3 >$
   if $w_1 = w_2$ and symm=true then
      setSymm(node)
   end if
   $< n, \phi >=< n, id >$                                          Case IV
end if
return $< n, \phi >$
```

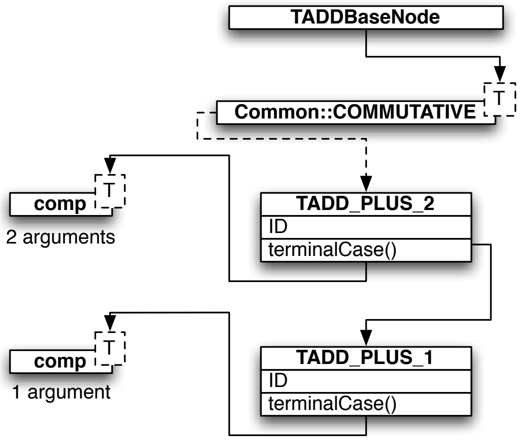


Figure 5.5: JINC's algorithm concept on the example of the plus operator

`TraversalHelper` so that there is no possibility to use a malformed pointer.

5.2.3 Iterators

TADDs uses nodes with four successors. Therefore the `TraversalHelper` enables the access to logical paths with a branch range of four. Most of the HOWDD instances supported by JINC have a branch range of two. For that reason the TADD implementation also defines a `TraversalHelper` with a branch range of two. This approach provides a compatible interface so that TADDs can be used like all other variants.

5.2.3.1 Swap

The approach of TADDs is based on edge weights. Therefore ordinary reordering methods cannot be applied directly. The swap algorithm has to be extended to handle the edge weights. The transformation of x_i and y_i variables to one z_i variable speeds up the swap operation for applications with interleaved variable orderings[42]. It is clear that both variables should not be separated and handled as a group. A swap of two neighboring groups with two variables requires four swap operations. In the case of TADDs it is performed with one swap operation. Figure 5.6 shows the idea of the TADD swap operation. (a) is the initial point. The highlighted nodes in (b) show the newly created nodes. The x node is then moved to the y level and linked to the newly created nodes (c). The second y node (from (a)) is then no longer referenced and can be deleted. Afterwards the levels are swapped (d).

```
template <typename T>
TADDBaseNode* comp(TraversalHelper<TADDBaseNode> v1, TraversalHelper<TADDBaseNode> v2,
                   PtrHashMap2<TADDBaseNode>& computedTable, TADDMemPools& pools){

  if(TADDBaseNode* comp=T::terminalCase(v1.node.ptr,v2.node.ptr,computedTable,pools))
    return comp;

  T::cacheOpt(v1.node.ptr,v2.node.ptr);

  if(TADDBaseNode* comp=computedTable.find(Common::calcPtr(v1.node.ptr,T::ID),v2.node.ptr))
    return comp;

  TADDBaseNode* w0;
  TADDBaseNode* w1;
  TADDBaseNode* w2;
  TADDBaseNode* w3;
  TADDUniqueTable* minTable;

  if(v1.getPtr()->getRealLevel()<=v2.getPtr()->getRealLevel()){
    minTable=v1.getPtr()->getUniqueTable();
    if(v2.getPtr()->getUniqueTable()==minTable){
      w0=comp<T>(v1.getSucc(0),v2.getSucc(0),computedTable,pools);
      w1=comp<T>(v1.getSucc(1),v2.getSucc(1),computedTable,pools);
      w2=comp<T>(v1.getSucc(2),v2.getSucc(2),computedTable,pools);
      w3=comp<T>(v1.getSucc(3),v2.getSucc(3),computedTable,pools);
    } else {
      w0=comp<T>(v1.getSucc(0),v2,computedTable,pools);
      w1=comp<T>(v1.getSucc(1),v2,computedTable,pools);
      w2=comp<T>(v1.getSucc(2),v2,computedTable,pools);
      w3=comp<T>(v1.getSucc(3),v2,computedTable,pools);
    }
  } else {
    minTable=v2.getPtr()->getUniqueTable();
    w0=comp<T>(v1,v2.getSucc(0),computedTable,pools);
    w1=comp<T>(v1,v2.getSucc(1),computedTable,pools);
    w2=comp<T>(v1,v2.getSucc(2),computedTable,pools);
    w3=comp<T>(v1,v2.getSucc(3),computedTable,pools);
  }

  TADDBaseNode* node=findOrAdd(pools.innerNodeMemPool,minTable,w0,w1,w2,w3);

  computedTable.insert(Common::calcPtr(v1.node.ptr,T::ID),v2.node.ptr,node);

  return node;
}
```

Listing 5.2: APPLY for TADDs

```
struct TADD_PLUS_2 : Common::COMMUTATIVE<TADDBaseNode>{
  static const unsigned short ID=ArithmeticConstants::ADD_OP;
  static inline TADDBaseNode* terminalCase(TraversalHelper<TADDBaseNode> v1,
                                           TraversalHelper<TADDBaseNode> v2,
                                           PtrHashMap2<TADDBaseNode>& computedTable,
                                           TADDMemPools& pools){

    if(v1.node.ptr==TADD::zeroDrain) return v2.node.ptr;
    if(v2.node.ptr==TADD::zeroDrain) return v1.node.ptr;
    if(v1.node.ptr==v2.node.ptr)
      return ::comp<TADD_PLUS_1>(v1.node.ptr,computedTable,pools);
    if(v1.isDrain() && v2.isDrain())
      return findOrAddDrain(pools.terminalNodeMemPool,v1.getValue()+v2.getValue());
    return 0;
  }
};

TADDFunction TADDFunction::operator+(const TADDFunction& function){
  TADDData& data=getTADDData();
  return TADDFunction(
           comp<TADD_PLUS_2>(
             rootNode,function.rootNode,
             *data.computedTables.arithmeticHashMap,data.memPools
           )
         );
}
```

Listing 5.3: Operator + for TADDs

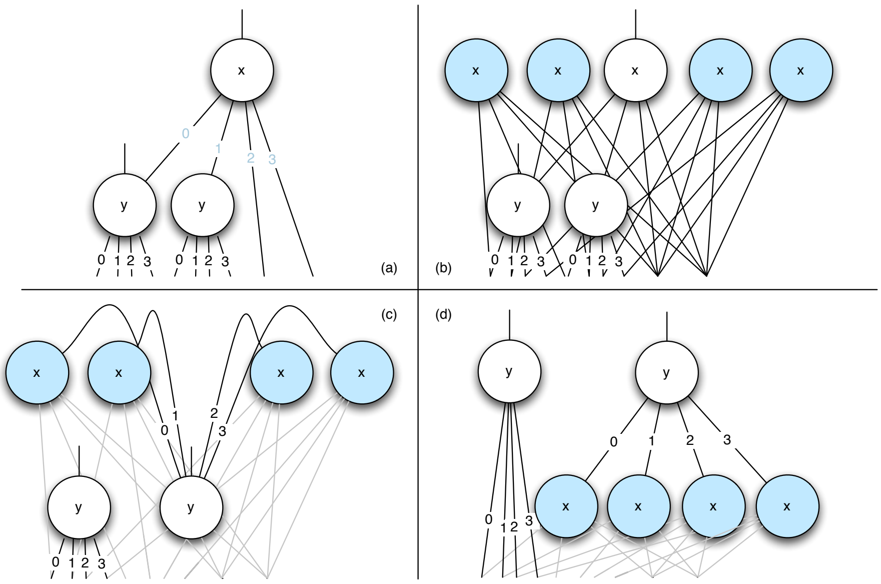


Figure 5.6: The idea of swapping the two levels.

5.3 Benchmarks

This section illustrates how this new HOWDD variant perform compared to the ordinary OBDDs.

The performance of Boolean operators is measured with the well known n-queens problem [11, 80, 47, 41]. The n-queens problem is the problem of putting n queens on a $n \times n$ chessboard such that none of them is able to attack the others. The movement and capture rules of a queen are governed by standard chess rules.

Table 5.2 shows the results for the n-queens problem. The times include the building

		OBDD		TADD	
n	Solutions	Nodes	Time(s)	Nodes	Time(s)
4	2	31	0,01	17	0,01
5	10	169	0,02	83	0,01
6	4	131	0,02	66	0,02
7	40	1.101	0,04	548	0,08
8	92	2.453	0,10	1.216	0,14
9	352	9.559	0,32	4.776	0,28
10	724	25.947	1,32	12.922	0,98
11	2.680	94.824	6,23	47.409	4,43
12	14.200	435.172	35,32	217.022	24,55

Table 5.2: Comparison between OBDDs and TADDs (n-queens problem)

of all solutions plus testing if all found solutions are valid[17]. With increasing runtime and BDD size TADD perform better than ordinary OBDDs because of the more compact representation and the more efficient cache hit-ratio. TADDs have less collisions and thus have more cache hits. The other advantage is that with the two-step approach less recursive calls are made. The overhead for the pointer arithmetic is noticeable for small instances of the n-queens problem.

[17] this involves the iterator concept described in Section 4.1.3

Chapter 6
JINC – Multi-Threading

The concepts described in the previous chapters lead to an efficient library design. The performance boost of computer hardware is going to end. [112] states that a dramatic change in software design is needed to gain more performance on modern system [96, 39, 91]. The idea to speed-up the OBDD manipulation is based on multi-threading. The discussed approach follows the principle to use shared memory to maintain the efficient recursive structure of OBDD algorithms. This method is different from the existing parallel computing approaches. Parallel computing has the advantage of more available physical memory as the memory of several computers can be combined. The communication effort reduces the performance so that the run-time performance is less than the traditional non-parallel approaches [78, 111]. JINC's design allows to apply multi-threading techniques to OBDD algorithms. This increases the performance without increasing the memory usage. A novel idea to reduce the memory footprint that can be used in a multi-threaded environment will be discussed in Section 6.7. Before discussing the details of JINC's multi-threading approaches we will discuss the possible speed-up of parallel execution and different design concepts.

6.1 Parallel Computing Concepts

Amadahl's law [5] states that if a fraction p of computation can be run in parallel while the rest of the computation must run serially the speed-up is upper-bounded by $\frac{1}{1-p}$. The goal of JINC's multi-threading approach is to speed-up all parts of calculation[1]. The alone application of Amadahl's law cannot help to understand the possible speed-up [52]. At first the fraction p must be evaluated which requires a different model.

Another model to understand parallel behavior is the direct acyclic graph (DAG) model for multi-threading [15]. The instructions of a program are represented by vertices, the edges indicate the dependencies between instructions. Instruction x precedes instruction y ($x < y$) if x must complete before y can start. If neither $x < y$ nor $y < x$ then x and y can be executed in parallel ($x|y$). Figure 6.1 shows a multi-threading DAG.

An important measure is the *work*, which is the sum of all instructions. For Figure 6.1 it is 16 (number of vertices). This model only uses uniform costs and does not included caching, overhead for communication, etc. For a more detailed theoretical view we refer to [71].

Let T_p be the fastest calculation time of the application with p processors. The *work* is equal to T_1. These notations help to understand the bound for the fastest possible

[1] no serial execution parts can be identified

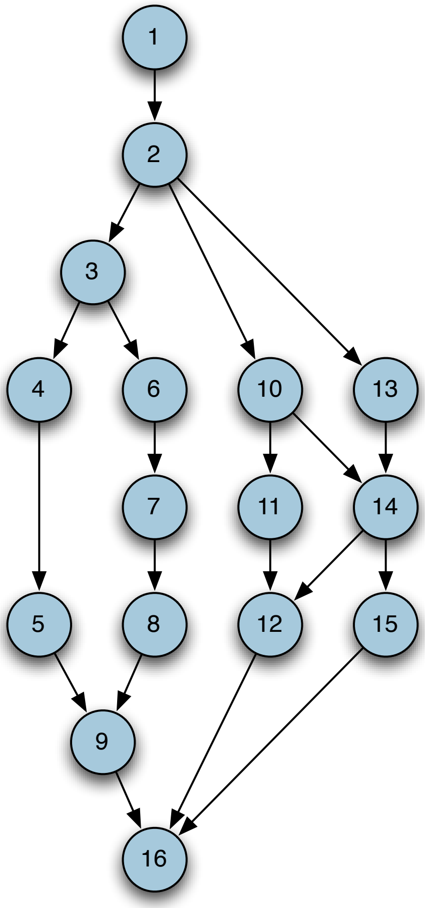


Figure 6.1: Multi-threading direct acyclic graph

execution time for p processors.

$$T_p \geq \frac{T_1}{p}$$

In this model each processor can execute one instruction per time and thus p processors can execute at most p instructions.

Another important measure is *span*, which is defined as the longest path of dependencies in the DAG. In Figure 6.1 it is 8 (e.g.,$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 16$). The *span* represents the fastest possible execution time with an infinite number of processors, i.e.,T_∞. This leads to the bound:

$$T_P \geq T_\infty.$$

Work and *span* are important measures to understand parallelism, as it is defined as $\frac{T_1}{T_\infty}$ (which means ration of *work* and *span*). These definition can be understood as the average amount of work along each step on the critical path. It is also the maximal speed-up which can be obtained by any number of processors.

In our example DAG (Figure 6.1) the maximal possible speed-up is $2 = \frac{T_1}{T_\infty} = \frac{16}{8}$.

After discussing the theoretical framework of parallel execution we will now discuss what is the *work* in an OBDD library which can be calculated in parallel. We identified three fields of activities in typical OBDD libraries. The first and most important is the calculation of OBDD structures. The second is the garbage collection phase. It is not reasonable in a parallel environment to execute the garbage collection among the regular calculation as there are strong side-effects, e.g., cache inconsistency. Therefore we will remove all automatic garbage collection and reordering methods from the calculation phase[2]. There is still the possibility for garbage collection and reordering to reduce memory usage after all calculation is finished[3]. This design separates these two phases, so that we can discuss them separately. The last part are the reordering algorithms. For efficiency reasons, the garbage collection is called directly before the reordering phase, so that only a minimal number of nodes have to be reordered.

Calculation The scope of this thesis covers parallelism only on the high-level operator view. We will not focus on the fact that the recursions can be executed in parallel. An approach like Cilk [95] can easily be integrated into the recursive structure of OBDD algorithms. The problem occurs with the work stealing policy [16]. Cilk's approach is based on independent calculations with few or no shared memory. For OBDD algorithms this is not the case. For this reason this approach is not within the scope of this thesis[4]. The unique-tables and computed-tables are the major obstacles to implement an efficient parallel OBDD library. Parallel computing

[2] See Section 6.7 for a new approach to reduce the necessity of automatic garbage collection and reordering.

[3] This can only be guaranteed in the main thread. See Section 6.2.1 for more details.

[4] Future work should investigate the possibilities if parallelism can also be implemented on an algorithmic level. A combination of JINC's high-level approach with low-level parallelism is imaginable.

with Message Passing Interface (MPI) [110] have been proven to be inefficient for OBDD libraries because of the communication overhead of unique-tables [78, 111]. Therefore JINC implements parallelism on an operator level and distributes the work to multiple threads managed in a thread-pool (see Section 6.3). The thread-pool has a fixed number of threads available and each thread has its own computed-tables and memory pools (see Section 6.2.3). This allows the use of shared memory which eliminates the communication overhead. The advantage of using thread-local data structures is that no concurrent situation occurs which increases the overall performance. The drawback of this approach is that multiple computed-tables lead to more computed-table misses. A computed result is only inserted in one computed-table. It is not unlikely that the next calculation is performed on a different thread where the result has not been inserted. To address this issue, Section 6.7 introduces a new approach to reduce the need of computed-tables.

Garbage Collection The garbage collection phase has different steps to perform. First to identify dead nodes and second to remove those nodes. Every level has its own unique-table so that the identification phase can be executed in parallel. The removal phase can also be executed in parallel. More details can be found in Section 6.5.

Reordering Reordering with JINC's reordering template approach (see Section 4.1.5) is straight forward. Each swap operation is equal to a vertex in the DAG. A more detailed description of JINC's novel approach and the modifications can be found in Section 6.6. This phase profits the most of parallel execution as the benchmarks in Section 7.2.2 shows.

6.1.1 Multi-Threading Challenges

JINC uses multi-threading with shared memory to speed-up the computation, reordering and garbage collection phase. There are several approaches for parallel OBDD packages like [111, 123] but all work on distributed BDDs (on several processes or computers) or uses breadth-first construction. JINC follows a different approach. It still remains focused on depth-first traversal to avoid the memory overhead of breadth-first traversal. The approaches differ in the underlying memory models.

The traditional approaches (parallel processes) are using different memory areas. This has the disadvantage that the main task has to communicate the data to the other processes [2]. The main advantage is that all processes run independently at full speed. Figure 6.2 illustrates the concept of parallel processes.

We will focus on the fact that computers with multi core architecture are widely available [76]. The advantages of shared memory for all parallel threads is that there is no communication overhead[5] [2]. The disadvantage of this method is that

[5] this idea is based on a lock-free algorithm [35, 46]

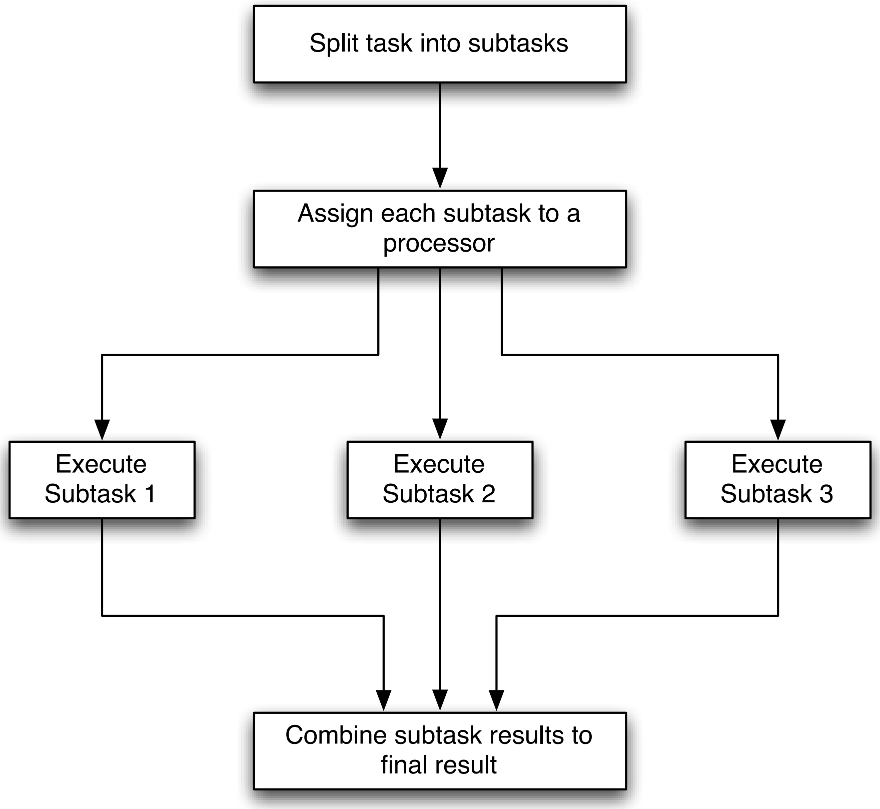


Figure 6.2: Concept of parallel computing

Init: x = 0

	Thread 1	Thread 2
A	x+1=1	x-1=-1
	x=1	
		x=-1
B	x+1=1	x-1=-1
		x=-1
	x=1	
C	x+1=1	
	x=1	
		x-1=0
		x=0
D		x-1=-1
		x=-1
	x+1=0	
	x=0	

Figure 6.3: The problem with multi-threading

(without precaution) concurrent threads can lead to unexpected behavior. Figure 6.3 shows the main problem with multi-threading. Situations *A* and *B* produce unpredicted results. To avoid concurrent reads and writes on sensitive data mutual exclusions (mutexes) [34] around critical sections must be used[6]. The main focus of JINC's implementation was to have only minimal or non-existing overhead for locking (see the following sections for more details).

6.1.2 JINC's Concept

JINC is the first multi-threaded library for OBDD manipulations. It is not only a transformation from the distributed approaches to a multi-threaded environment (like [93]), it is based on new ideas to benefit from today's available hardware architectures.

JINC's multi-threading approach does not break the basic concept described before. Figure 6.4 illustrates the new concept (modified parts are marked). The function object is nearly unchanged. The differences rely on the pooling for the computed-tables and the memory pools. A Thread Local Storage (TLS) [121] variable is used to identify the local data structures of a thread. Each thread owns a separate set of memory pools and computed-tables. The unique-table has been modified so that concurrent access is possible. A new function object has been developed that enables multi-threaded calculation.

The maximal number of parallel threads in JINC is limited (see Section 6.3 for more

[6] Situations *C* and *D* would be the resulting outcome.

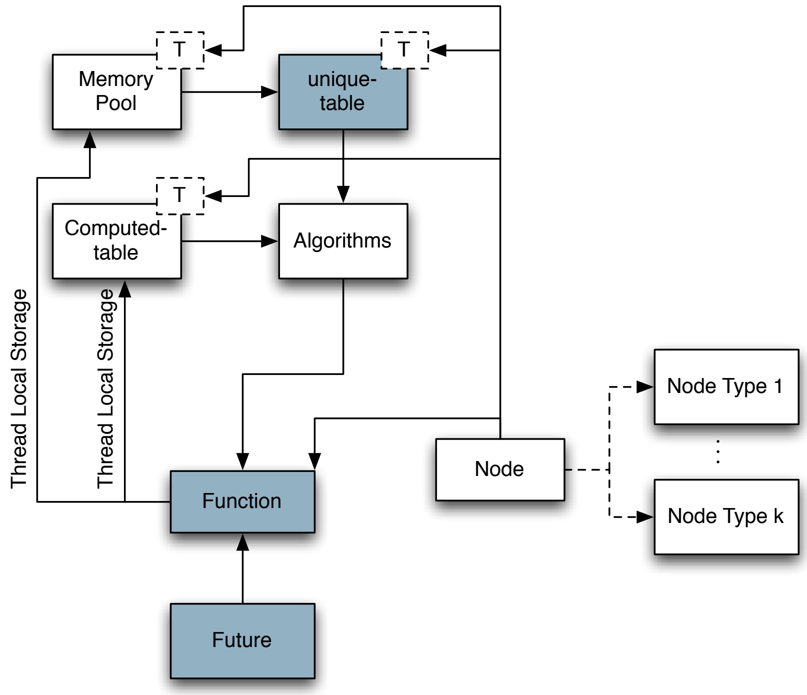


Figure 6.4: Multi-Threaded API concept

details). The number of threads can be configured at start-time. With this approach it is possible to change the multi-threading behavior without recompilation. The recommended number of parallel threads is at least the number of cores. To many threads reduce the performance as there are too many context switches and it reduces the efficiency of hardware caches.

Further details on the changes for a multi-threaded environment are discussed in the following sections.

6.2 Data Structures

The following section describes the changes in the data structures to work efficient in a multi-threaded environment.

6.2.1 Nodes

The requirements on the node types are different in a multi-threading environment. Successor pointers and transformation functions are just altered in garbage collection phases. So under the assumption that no garbage collection takes place during multi-threaded calculation no special treatment for these fields is needed. Therefore, JINC removed the automatic garbage collection process. Instead there is a management function that checks if garbage collection is needed. If the memory usage is too high the garbage collection is called. The management function must only be called in the main thread as it acts as a synchronization point. The reference counter is thereby the most critical part because it is changed multiple times during calculations. Due to concurrency in a multi-threading environment it is possible that a wrong reference counter implementation could result in incorrect values. Figure 6.3 illustrates this problem. This, in combination with the garbage collection will lead to unexpected behavior, e.g., missed incrementations of the reference counter could lead to the deletion of still used objects. JINC uses atomic incrementation and decrementation to insure correct reference counter values [46, 77] in a concurrent environment. Atomic operations are provided by most modern processors and are more efficient as protecting a critical section with a lock.

The other fields (like successors, transformations, unique-table pointer and terminal value) of the different nodes types do not need to be thread-safe. The values are only altered during the garbage collection and reordering phase. These algorithms are designed to work on distinct node sets and thereby need no special treatment for multi-threading. During OBDD manipulation these values are initialized once and not altered afterwards.

The `next` pointer is only altered in the unique-table algorithms. The unique-table is thread-safe (see Section 6.2.2) and thereby this value.

6.2.2 Unique-table

Unique-tables in a multi-threaded environment are the bottle-neck for parallel execution. The usage of one mutex for every unique-table would result in blocking behavior if several threads try to access the same unique-table. To use one mutex of every slot in a unique-table has several drawbacks. The size of a mutex increases the memory usage dramatically and the overhead for locking is greater than the performed operation in the critical section[7]. JINC uses spin locks [6] for every unique-table slot to implement an efficient multi-threaded unique-table with low memory overhead.

The `findOrAdd` algorithm in a multi-threaded environment is more complex than the single-threaded version. It is important to recognize that searching for a node in the unique-table can be implemented without any locking. This is due to the fact that JINC removed the automatic garbage collection process. In the case of a successful search the result can be returned. In the case of an unsuccessful search the slot lock must be requested. Before inserting a new node a new search (with active lock) must be performed. This is important because another thread could have inserted the searched node just between the failed search and the acquisition of the lock. If this search also fails a new node must be inserted. Note, that the second search could be started just before the first search failed[8]. Figure 6.5 shows

[7] The space efficiency is not the key factor as the space requirements does grow with the number of slots and variables but not with the number of nodes in the BDD. The mutex algorithm presented in [33] provides a space-efficient solution (if the number of threads is fixed) with fairness guarantees. The low collision probability does not justify a complex run-time locking algorithm.

[8] The linked list is sorted. See Section 4.1.1.2 for more details.

two threads that access the same unique-table slot. One thread has occupied the lock and inserts a new node. The other threads performs an unsuccessful search. This example shows that even the newly inserted node will be considered during the second search. That means that concurrent successful search operations can be performed lock-free. In the single-threaded case the acquisition of the lock is the only overhead for the insertion of a new node. Another adjustment to the single-threaded unique-table is that the dynamic resizing cannot take place during computation. The dynamic resize process has been moved to the garbage collection phase.

6.2.3 Memory Pools

The memory pool has been used in a single-threaded environment to avoid memory fragmentation. In a multi-threaded environment there are several unique-tables accessed in parallel. The access to the memory pool has to be thread-safe. Securing the memory pool with a mutex would negate all advantages of multi-core architectures. This is due to the fact that allocating and freeing nodes are highly frequent tasks of an OBDD library. JINC follows the approach that each active thread uses its own memory pool. This can be accomplished because of the limited number of parallel active threads (see Section 6.3). With this idea each thread can allocate and free memory independently.

6.2.4 Computed-table

The computed-table is important to increase the performance of an OBDD library. The read and write operations (as described in Section 4.1.2.1) are optimized to be very efficient. The computed-table performance drops dramatically when using mutexes [48]. JINC follows the same idea as for the memory pool. Every thread uses its own computed-table. The overall memory usage of the multiple computed-tables is comparable to the one computed-table in the single-threaded environment.

6.2.5 Variable Ordering

The variable ordering is almost unmodified compared to the single-threaded approach. This is due to the fact that the variable ordering definition phase is usually at the beginning of the calculation. Whenever the variable ordering has to be altered there should be only one active thread. For the case of parallel reordering (see Section 6.6) the swap of two neighbored variables has been implemented thread-safe.

6.3 Thread-pool

All former described changes are based on the idea that several threads use data structures concurrently. This section explains the details behind JINC's thread-pool

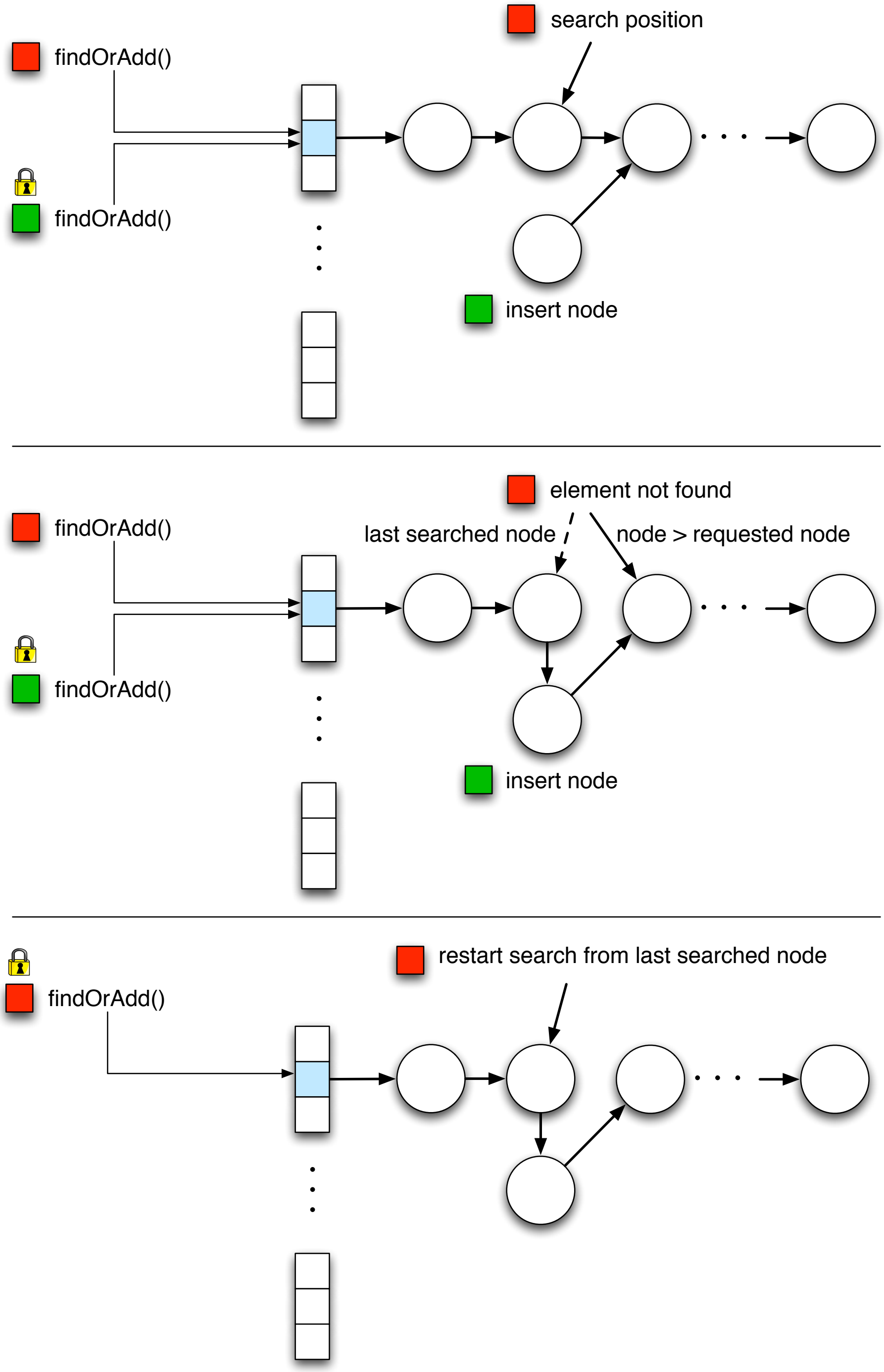


Figure 6.5: Multi-threaded unique-table look-up

[79, 73, 40].

The idea behind a thread-pool is that a number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are more tasks than threads. As soon as a thread completes its task, it will request the next task from the queue until all tasks have been completed. The thread sleeps until there are new tasks available. JINC uses a thread-pool to limit the number of parallel threads. An unlimited number of threads could lead to reduced performance because of too many context switches [70]. The fixation lead to optimized unique-table (see Section 6.2.2) and computed-table (see Section 6.2.4) handling. Another advantage of using a thread-pool over creating a new thread for every task is that the costs for thread creation and deletion are negated. This results in better performance and system stability.

The used thread-pool (limited to T threads) fulfills the following conditions:

- each task is performed the same order as in the working queue
- a maximum of T tasks are processed in parallel
- each scheduled task will be executed only once.

JINC uses one global thread-pool. In the following creating or starting a thread is equivalent with adding a new task to the thread-pool.

6.4 Futures

The idea behind JINC's multi-threaded approach is that every thread contains some local storage and makes use of the shared data structures. The function object itself has to be adjusted to use the memory pools and computed-tables that are bound to the particular thread. JINC uses a TLS pointer to bind data structures to a thread. The pointer is initially zero. Before the execution of a HOWDD algorithm, the function object addresses the TLS pointer and checks it against zero. If it is zero a container (containing memory pools and computed) is allocated and the pointer is set to the containers memory address.

To benefit from the multi-threaded data structures JINC uses the idea of delayed function evaluation [13, 99].

The single-threaded object oriented API has been modified to work in a multi-threaded environment. In a usual program flow a function call is completely evaluated before continuing with the next one. Listing 6.1 shows an example program that could benefit from the multi-threaded approach. Figure 6.6 shows the DAG of this program[9] and identifies possible paths for parallel execution. Even in this simple program some parallel calculations can be performed.

JINC uses a generic approach to solve this problem. A template object called `Future` acts like a multi-threading management layer [9]. A value or function can be assigned

[9] Line number l correspondent to the l-labelled vertex.

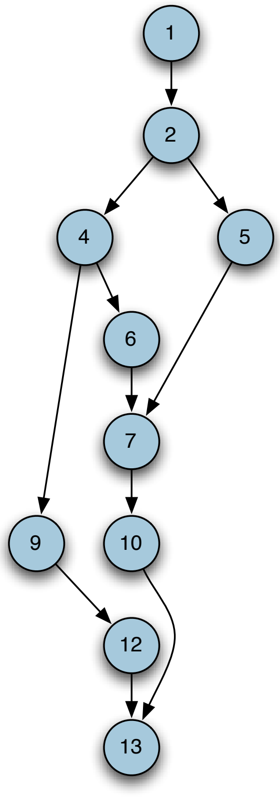


Figure 6.6: DAG of Listing 6.1

```
 1 BDDFunction someBigFunction=...
 2 BDDFunction anotherBigFunction=...
 3
 4 BDDFunction tempResult=someBigFunction*someBigFunction;
 5 BDDFunction temp1=someBigFunction*anotherBigFunction;
 6 BDDFunction temp2=tempResult*anotherBigFunction;
 7 BDDFunction result=temp1+temp2;
 8
 9 unsigned long sTempResult=tempResult.size();
10 unsigned long sResult=result.size();
11
12 std::cout << sTempResult << std::endl;
13 std::cout << sResult << std::endl;
```

Listing 6.1: The need of lazy evaluation

to a `Future` at construction time, but a `Future` cannot be reassigned. This ensures that a object holds the same state over its lifetime. Each object contains a value field and a Boolean flag to identify the validity of the value. The Boolean flag is used to identify if the calculation of the value is finished and thereby a valid value available. If an already calculated value[10] is assigned to a `Future` the value is stored and the valid flag is set to true (no thread will be created in this case). If `Future` *f* is assigned to a `Future` (e.g.,Listing 6.2 line 6) it creates a new thread and sets the valid flag to false. The thread then waits until the valid flag of *f* becomes true. In that case the value is copied and the valid flag is set to true. If a function object is assigned to a `Future` the valid flag is set to false and a thread which executes the function object is created. This function object waits until all parameters have valid values and starts the evaluation. After the execution of this function object the value is copied and the valid flag set to true. The waiting process does not consume any performance because `Future` uses elaborate to inform all waiting threads through a condition variable.

Listing 6.2 shows the same situation as in Listing 6.1 but uses JINC's lazy evaluation approach. The template parameter represents the type of the value stored in the `Future` object. Lines 9 and 10 show that it is possible to combine different calculations types. This is necessary as the size calculation is bind to the `BDDFunction` but returns a numerical type[11]. The lines 4-10 are not executed in the main thread. Each line creates a function object which is scheduled in the thread-pool. With the precaution that those function objects cannot be reassigned and that the thread-pool executes the jobs in the given order it is not possible to create a dead-lock situation. Lines 12 and 13 are executed in the main thread and thus assure the right order of the output (see Figure 6.6).

```
 1 BDDFunction someBigFunction=...
 2 BDDFunction anotherBigFunction=...
 3
 4 Future<BDDFunction> tempResult(multiplication,someBigFunction,someBigFunction);
 5 Future<BDDFunction> temp1(multiplication,someBigFunction,anotherBigFunction);
 6 Future<BDDFunction> temp2(multiplication,tempResult,anotherBigFunction);
 7 Future<BDDFunction> result(plus,temp1,temp2);
 8
 9 Future<unsigned long> sTempResult=Future::createSizeFunction<BDDFunction>(tempResult);
10 Future<unsigned long> sResult=Future::createSizeFunction<BDDFunction>(result);
11
12 std::cout << sTempResult.getValue() << std::endl;
13 std::cout << sResult.getValue() << std::endl;
```

Listing 6.2: Lazy evaluation in JINC

Another usage for lazy evaluation is the implementation of iterative methods. During the check if a further iteration step is needed the next step could be processed in parallel. Listing 6.3 shows the idea behind using lazy evaluation in iterative methods.

[10] That means a function object of the single-threaded API.

[11] in this case an unsigned long

```
{
  Future<bool> check(checkCondition,function);
  Future<BDDFunction> next(computeIterationStep,function);
  function=next.getValue();
} while(check.getValue());
```

Listing 6.3: Lazy evaluation for iterative methods

6.5 Garbage Collection

Garbage collection in a single-threaded environment halts the computation phase, collects and removes dead nodes, cleans the computed-tables and starts the computation phase afterwards. In a multi-threaded environment it is not reasonable to perform a garbage collection on one thread and perform calculations on other threads. The computation phase is also halted as in the single-threaded case. The main difference rely in the possibility to use several threads at once to collect and remove dead nodes.

JINC collects all dead nodes on every level and removes them from the unique-table. During removal the dead nodes are linked via the `next` pointer of the provided base node. This operation is common for all variants and is thus provided by the generic unique-table implementation. Each variable in the context of the variable ordering has its own unique-table so that this process can be executed in parallel (without mutexes). Figure 6.7 illustrates the parallel garbage collection process from the perspective of one unique-table. After the collection phase there are n linked lists of dead nodes[12]. For every node in the linked lists a traversal must be performed to identify the next dead nodes. JINC uses a thread-safe reference counting mechanism so that this process can be performed in parallel. The newly identified nodes are then linked to the existing lists. After this step the n linked lists have to be freed.

6.6 Parallel Reordering

Reordering algorithms are used to reduce the number of nodes. This thesis introduces a run-time efficient method to use multi-threading architectures to speed-up reordering algorithms. There have been several approaches to parallelize the reordering process [45, 97, 82, 78, 103, 12, 83]. These approaches work on several computers. The disadvantage of distributing the reordering process to several computers is the communication overhead. The advantage of the distributed solution is that the global available memory is higher. JINC follows an orthogonal idea. Using shared memory for all parallel threads is used to speed-up the calculation. This approach can be combined with the traditional ideas. Each process (of the traditional methods) could implement this new approach to speed-up each process independently.

JINC's reordering approach has been adapted to work with multiple threads in

[12] where n is the number of variables which is equal to the number of unique-tables

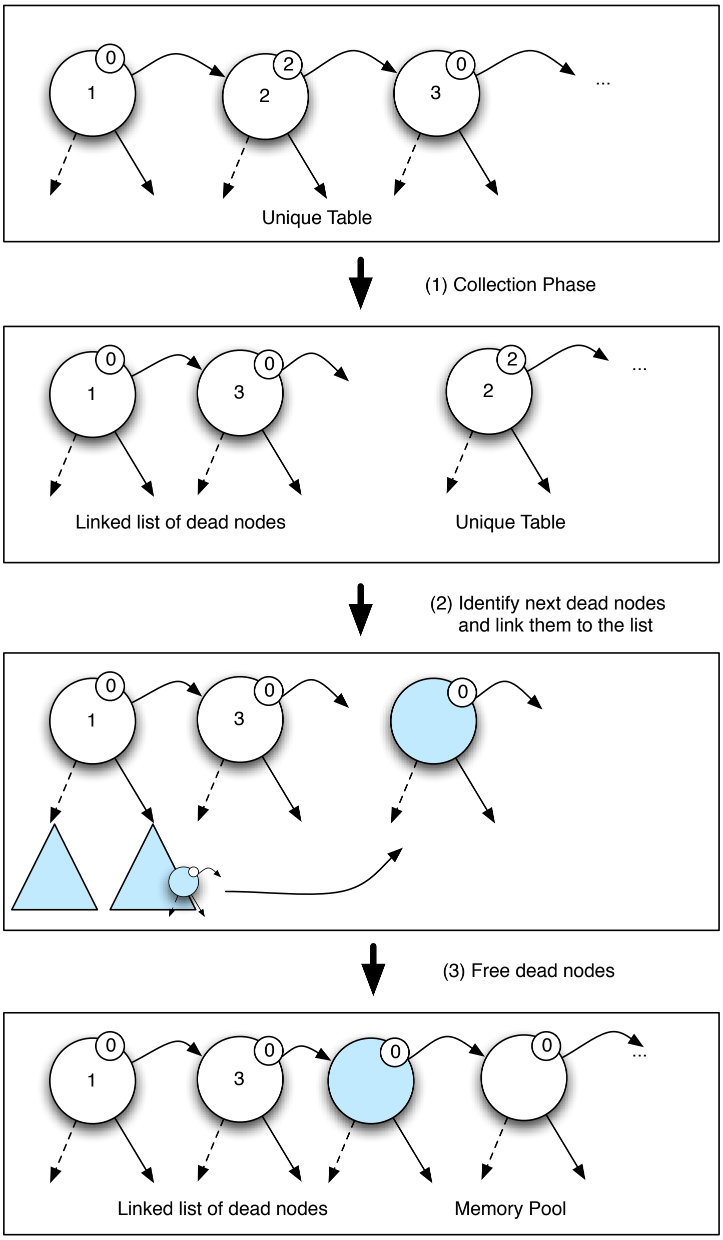


Figure 6.7: Multi-threaded garbage collection approach (from the perspective of one unique-table)

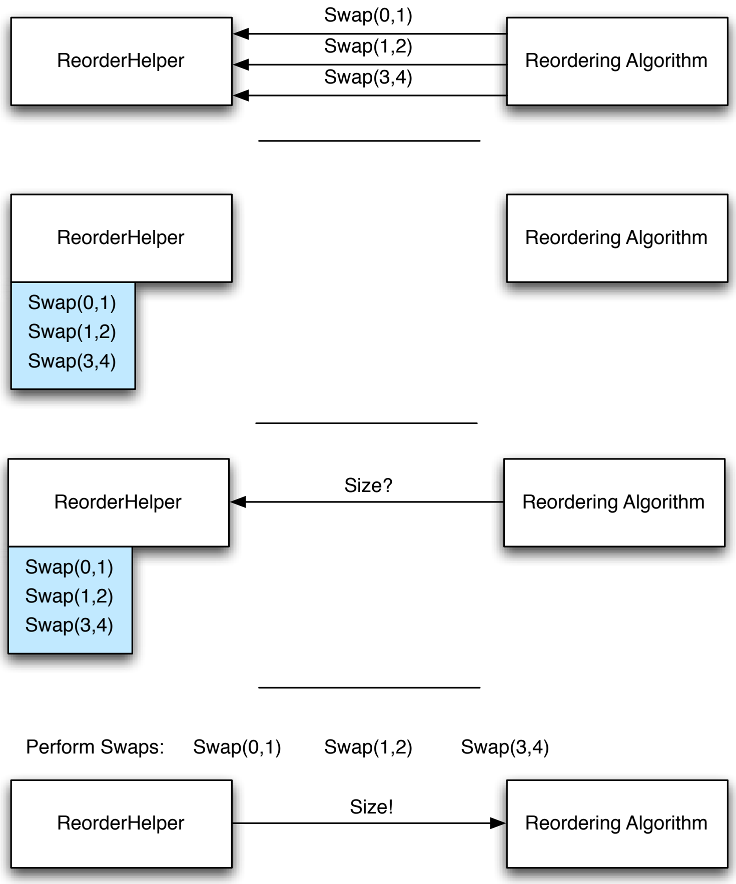


Figure 6.8: The idea of queued (delayed) swap operations

parallel. This is based on the idea that it is not necessary to perform every swap operation directly. JINC delays the swap operation till the reordering template calls the size function or the reordering is finished. The swap operations are stored in a queue. The collection of swap operations allows for checking whether it is possible to create multiple parallel threads. The parallel execution of swap operations is possible because the swap operation has only local influence (on the involved levels). The only influence on other levels is the incrementation and decrementation of the reference counter. This is implemented as an atomic operation [46, 77] (as described in Section 6.2.1) and thereby is a lock-free mechanism.

The memory overhead of JINC's approach is limited to the size of the waiting list of swap operations and is not dependent on the size of the BDD.

The used approach uses a bit field to identify possible swaps. All bits are initially set to zero. For every swap operation it is checked if the affected positions are blocked[13]. A swap operation cannot be performed if at least one position is blocked. In both cases the corresponding positions are set. With this simple check it is possible to identify independent swap operations. Figure 6.9 illustrates the identification process. Whenever a swap operation is ready it reports to the `ReorderHelper`. The

[13] i.e., swap(i,$i+1$) will check if one of the bits on position i or $i+1$ is set

`ReorderHelper` removes the finished operation from the parallel swap list and starts the identification process again[14]. Algorithm 3 shows in detail how the scheduling process is performed. The last step is repeated until all swap operations are performed. The swap operations are scheduled in the thread-pool (see Section 6.3).

The order of the swap operations is observed in the identification algorithm. The current swap operations P and the pending swap operations S are stored in a list. This ensures that the dependencies of swap operations are maintained.

Algorithm 3 Parallel-Swap-Scheduling

Input: List P of parallel swap operations, list S of pending swaps, number of variables n and optional the list F of finished swap operations till last schedule
Output: List of parallel swap operations

```
remove all entries of $F$ from $P$
create bit array $B$ of size $n$ and set all elements to zero
for all $p$ in $P$ do
  $B[\![index(p)]\!] = 1$
  $B[\![index(p) + 1]\!] = 1$
end for
for all $s$ in $S$ do
  if $B[\![index(s)]\!] = 0$ and $B[\![index(s) + 1]\!] = 0$ then
    add $s$ to $P$
    add $s$ to thread-pool
  end if
  $B[\![index(s)]\!] = 1$
  $B[\![index(s) + 1]\!] = 1$
end for
return $P$
```

The biggest advantage of this approach is that there is no need to change the reordering algorithms to work in a multi-threaded environment. Different reordering methods [101, 53] benefit differently from this new approach. The sifting algorithm usually performs one swap operation and measures the size of the OBDD. In this case no parallel execution could be accomplished. Genetic reordering methods [36, 69, 26] are best suited to benefit from this approach. Genetic algorithms create several *individuals* which have to be measured. Each individual has to be created[15] and can then be measured. The swap operation that are needed to go from one individual to another are highly independent. The individuals are measured in the order so that the global number of swap operations is minimal.

This parallel approach is also applicable for group reordering algorithms as they implement one group swap with several variable swap operations. Figure 6.10 illustrates the parallel operations and shows that the number of parallel swap operations

[14] The current active operations (parallel swap list) are set as blocking before starting the identification phase.

[15] the variable ordering has to be applied to the OBDD

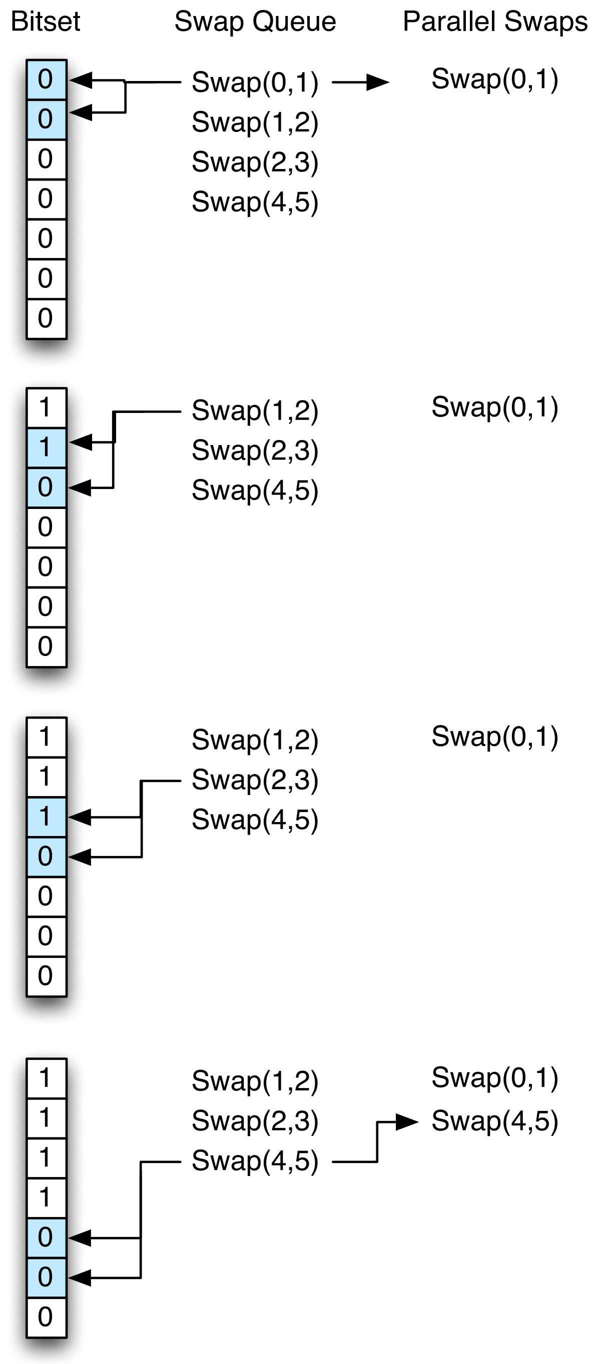


Figure 6.9: Identification of possible parallel swap operations

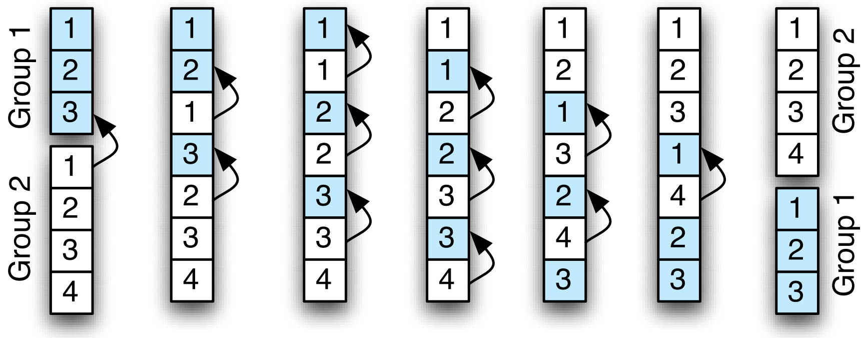


Figure 6.10: Parallel swap operations for group reordering algorithms

is limited by the group size. If the first group consists of g_1 variables and the second group consists of g_2 variables (with $g_1 \leq g_2$) the maximal speed-up factor can be calculated by:

$$\frac{work}{span} = \frac{2 \cdot \sum_{i=1}^{g_1-1} i + (g_2 - g_1 + 1) \cdot g_1}{g_1 + g_2 - 1} = \frac{g_1 \cdot g_2}{g_1 + g_2 - 1}$$

This is the theoretical possible speed-up as not every swap operation consumes the same time (see Section 7.2.2.2).

An idea to benefit from parallel swap operations with the sifting algorithms is to partition the variable ordering into groups. The ordering of each group can than be reordered in parallel. JINC's architecture supports this in a natural way.

The only changes that are necessary to support this multi-threaded reordering improvement has to be made for the `ReorderHelper`. The reordering system described in Section 4.1.5 then can be used for either variables or groups of variables.

Chapter 7 investigates the performance speed-up of this novel approach of a multi-threaded reordering system.

6.7 Generic Multi-Operand `APPLY`

The basis for multi-threaded calculation has been introduced with the `Future` approach. The object oriented API provides overloaded operators which are evaluated at once, e.g., the term $(f + g) \cdot h$ first evaluates $f + g$ and afterwards the multiplies h to the result. To benefit from the `Future` approach the complete term has to be capsulated into a function object. This does not follow the philosophy of JINC as it complicates the development. Therefore JINC introduces a novel lazy evaluation approach for OBDD operators that is well suited for being separated to other threads. The idea is based on the expression template approach [118]. This concept utilizes the compiler's parse tree to generate an object which represents a complete term.

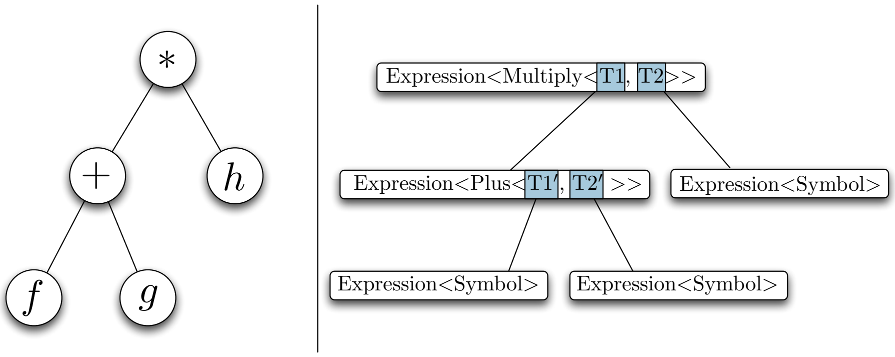


Figure 6.11: Parse tree and expression template structure for the term $(f + g) \cdot h$

The template structure is created at compile-time and does not influence run-time. Based on this object a generic `APPLY` algorithm is implemented which evaluates the term at once. This approach has two advantages. First, operator evaluation can be postponed. This in combination with the `Future` approach eases the development of algorithms using multi-threaded. Second, and more importantly, the immediate evaluation of a term with arbitrary arguments and structure creates no temporary nodes during computation and thus speeds up OBDD manipulations. The elimination of temporary nodes allows for the representation of larger systems. The reduced need of computed-tables is important as JINC uses thread-local computed tables and disabled the automatic garbage collection due to concurrency.

This approach is superior to the MORE approach [51, 27] as there is no structure modification, no node overhead and no need for applying a reduction algorithm afterwards.

6.7.1 Expression Templates

This section illustrates the expression template approach [118]. It will show how an object structure represents a term which can be evaluated later on.

Figure 6.11 shows the parse tree for $(f + g) \cdot h$ and JINC's expression template structure. Each function is represented by the `Symbol` class. Each operator is represented by its own class. The class `Expression` acts as a separator to eliminate any side-effects with already existing definitions. `Expression` combines `Symbols` and operator classes and forwards every call to its template type. For this reason we will not discuss any further details of `Expression`. The `Symbol` class stores a pointer to the root node of a function. The operator classes store their left-hand side and right-hand side argument.

6.7.2 Generic Multi-Operand `APPLY` algorithm

After illustrating the conceptual design of JINC's expression templates we will now discuss how recursive HOWDD algorithms can operate on this structure.

At first the node pointers must be extracted out of the structure. The idea is to create an array in which each `Symbol` element stores its pointer. To evaluate the needed array space each object is enriched with *size* information. Listing 6.4 illustrates the concept. `Symbol` just stores one argument and thus has size one. An operator does not store a value itself but its template types. Thus, its size is the sum of the template type sizes. Note, the `OperatorBase` class implements the common behavior of each operator.

We now can create an array with appropriate size. It remains to fill the array with the corresponding values. This is implemented with a slicing mechanism. Listing 6.4 shows the implementation. The size value is now used to slice the information. Figure 6.12 illustrates the concept on the example of term $(f+g)\cdot h$. An array of size 3 is created. The pointer to the first element of the array is passed to the object (i). The operator object first fills the left-hand side argument and thus just passes the pointer to it (ii). The next operator does the same (iii). The `Symbol` element stores its content in the corresponding element and returns (iv). The operator object now fills the right-hand side argument. Therefore, the pointer to element with index 1 (initial index plus size of left-hand side argument$= 0 + 1 = 1$) is given to the right-hand argument. The `Symbol` element stores its content in the element corresponding to the given array pointer (v). Thus, the element with index 1 is filled through a slicing mechanism. The operator object is finished with filling and returns (vi). The index to the array is calculated in the same way as before (index plus size of left-hand size argument$= 0 + 2 = 2$) (vii). The `Symbol` element stores its content in the appropriate element and returns (viii). (ix) shows the final situation.

The remaining parts of a HOWDD algorithm are the terminal case check and the recursive calculation steps. The terminal case check must be implemented for every structure. As in the case of initial value filling, a pointer to the element array must be given as parameter. `Symbol` just returns the first element of the array. The operator classes must implement their corresponding checks. The only difference to regular algorithm handling is that left-hand and right-hand side parameters can for their part be operators. This enables early termination, e.g.,$(f + g) \cdot 0$ will return 0 and not calculate $f + g$. As result no temporary nodes will be created during calculation. It is important to note that all information about terminal case handling depends on the template type and the input values. Therefore, no object instance is needed for algorithm handling. The generic `APPLY` algorithm is illustrated in Listing 6.5. The algorithm consists of two parts. First, the structure is created and the initial array is filled (line 32). Second, the recursive algorithm is called with the initial values (line 33). The recursive algorithm first checks for terminal cases (line 3). If it is a terminal case the algorithm returns the result[16]. In the other case it calculates the successors (lines 12 and 17) and calls the algorithm recursively for each successor array (lines 13 and 18). The final step is to combine the resulting nodes (line 20). Listing 6.5 includes caching only in the case that the left-hand and right-hand side parameters are `Symbols` (lines 5-7 and 22-24). This behavior is equivalent to the traditional `APPLY` algorithm with two arguments. Future work

[16] Terminal case handling is done with recursive traversal of the template structure which has been created at compile-time.

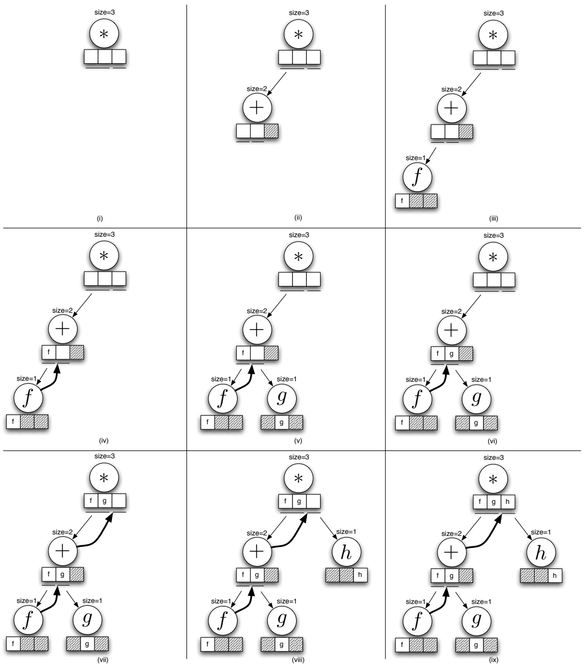


Figure 6.12: Array filling concept illustrated on the term $(f + g) \cdot h$

```
template <typename T>
struct Expression {
  enum {size=T::size};

  ...

  void fill(BaseNode** nodes) const {
    value.fill(nodes);
  }
};

struct Symbol {
  enum {size=1};

  ...

  void fill(BaseNode** nodes) const {
    (*nodes)=node;
  }
};

template <typename T1, typename T2>
struct OperatorBase {
  enum {size=T1::size+T2::size};

  ...

  void fill(BaseNode** nodes) const {
    lhs.fill(nodes);
    rhs.fill(nodes+T1::size);
  }
};

template <typename T1, typename T2>
struct Multiply : OperatorBase<T1,T2> {

  ...

};
```

Listing 6.4: Size concept and array filling with slicing

```
template <typename T>
BaseNode* apply_impl(BaseNode** nodes){
  if(BaseNode* comp=T::terminalCase(nodes)) return comp;

  if(T::size==2){
    if(BaseNode* comp=T::find(nodes)) return comp;
  }

  UniqueTable* table=T::getMinUniqueTable(nodes);

  BaseNode* succs[T::size];
  T::fillSuccs(nodes,succs,table,0);
  BaseNode* w1=apply_impl<T>(succs);

  ...

  T::fillSuccs(nodes,succs,table,m-1);
  ADDBaseNode* wm=apply_impl<T>(succs);

  BaseNode* result=findOrAdd(table,w1,w2,...,wm);

  if(T::size==2){
    T::insert(nodes,result);
  }

  return result;
}

template <typename T>
BDDFunction apply(const Expression<T>& s){
  BaseNode* nodes[T::size];
  s.fill(nodes);
  return BDDFunction(apply_impl<T>(nodes));
}
```

Listing 6.5: Generic multi-operand APPLY

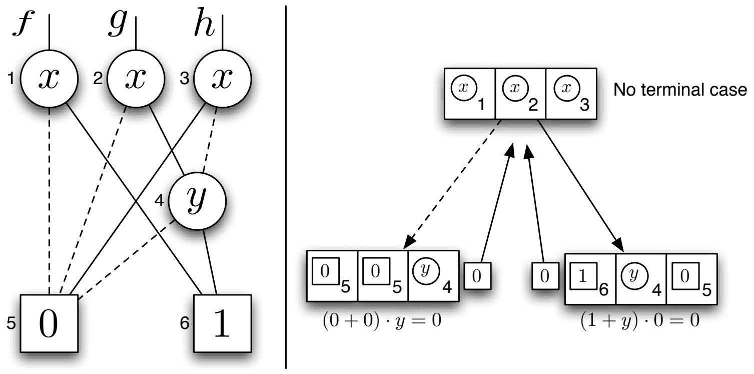


Figure 6.13: Generic multi-operand `APPLY` example

should investigate the possibility to cache operator classes. The concept enables the addition of new types without changing the algorithm. It is also possible to introduce non-operator calculations like existential quantification, etc. The correctness of this novel algorithm can be easily seen as we just use a combination of correct two-operand algorithms. Note that the described algorithm creates terminal nodes in `T::terminalCase` for simplicity reasons. In practise, terminal nodes should only be created in `apply_impl` to avoid the creation of temporary terminal values.

Figure 6.13 illustrates the algorithmic behavior on the example $(f + g) \cdot h$ with $f = x$, $g = x \cdot y$ and $h = (1 - x) \cdot y$. This figure shows the parse tree and the HOWDD structure of the functions. Within this example the functions are represented in an ADD. The initial array does not match any terminal case. The 0-successor node array is then created and the algorithm called recursively. In this step the terminal case of the plus operator matches and returns the 0 drain as result. The multiplication operator returns the 0 drain because the result from the plus operator matches a terminal case. The 1-successor node array is then created and the algorithm called recursively. In this case the multiplication operator can immediately return the 0 drain. The total result of this expression is 0 (elimination rule). The regular `APPLY` approach would first create $f + g = x \cdot (y + 1)$ and thus create a not needed temporary function.

The concept of this new generic multi-operand `APPLY` is illustrated on pointer based variants. The concept is also valid for variants which use the `TraversalHelper` approach.

Chapter 7 Case Studies

This chapter benchmarks the performance of JINC in its single-threaded and multi-threaded version.

At first JINC's performance, without using its multi-threading abilities, is measured in different settings. These benchmarks also include comparisons to the well established OBDD library CUDD [109].

The multi-threading benchmarks are divided into garbage collection, calculation and reordering phase to show the pure performance of each approach. The last benchmark shows the performance of the novel multi-operand `APPLY` algorithm.

7.1 Single-Threaded JINC

JINC's single-threaded design illustrated in Chapter 4 is not used in the following benchmarks. JINC's multi-threaded version is used. The thread-pool size is limited to zero, so that no scheduling or thread creation takes place. The data structures are designed so that they have minimal overhead in a single-threaded environment.

7.1.1 Model Checking

We will use the application model checking to measure the single-threaded performance of the HOWDD library JINC. Model checking has been chosen as a benchmark because model construction and validation of properties uses a broad range of OBDD algorithms. The used model checking environment PROMOC is based on JINC [86]. The used formalism to model the reactive modules is also used by PRISM [92]. PRISM uses the BDD library CUDD [109]. The same model checking language makes it possible to compare the performance of JINC and CUDD. Therefore, this benchmark also includes measurements with the established model checking tool PRISM [63]. All used models can be found on the benchmark page of PRISM [64]. The methods how to build the system are similar for PRISM and PROMOC. The only differences rely on the generated variable ordering[1] and internal resolution of probabilistic and non-deterministic choices. For our benchmark we will use an example where both programs generate the same variable ordering. This is an important characteristic as this benchmark measures the performance of JINC and CUDD and not the influence of different variable orderings on model checking.

Besides a pure symbolic calculation engine, PRISM provides a sparse matrix engine and a hybrid engine which combines the advantages of OBDDs and the sparse matrix representation. We will just measure against the pure symbolic calculation engine as we are interested in the performance of single-threaded JINC. The hybrid approach

[1] both libraries use an interleaved variable ordering [42, 49]

involves less OBDD operations and we therefore would obtain unreliable information about the OBDD library performance.

7.1.1.1 Leader Election Protocol

The leader election protocol is used to elect a leader out of N processors. In this case all processors are arranged in a synchronous ring. For this case study we will use the leader election protocol from [58].

This protocol uses the idea that every processor randomly chooses a number from 1 to K. All chosen numbers are passed around the ring. The processor with the highest chosen number will be the leader. If there are more than one processors with the same number, the election starts again.

In this case study, we will vary the number of processors N and the range K. Listing 7.1 shows the leader election model. In this example, the number of processes N is three and the range of the domain K is two. We will compare the building time for the system for different variants. We will also measure the time to calculate the probabilities to elect a leader within L rounds and the expected number of rounds to elect a leader.

```
probabilistic

const N=3; // number of processes
const K=2; // range of probabilistic choice

module counter
  c : [1..N-1];

  [read] c<N-1 -> (c'=c+1);
  [read] c=N-1 -> (c'=c);
  [done] u1 | u2 | u3 -> (c'=c);
  [retry] !(u1 | u2 | u3) -> (c'=1);
  [loop] s1=3 -> (c'=c);
endmodule

module process1
  // s1=0 make random choice
  // s1=1 reading
  // s1=2 deciding
  // s1=3 finished
  s1 : [0..3];
  // has a unique id
  u1 : bool;
  // value to be sent to next process in the ring
  v1 : [0..K-1];
  //random choice
  p1 : [0..K-1];

  //choose number
  [pick] s1=0 ->   1/K : (s1'=1) & (p1'=0) & (v1'=0) & (u1'=true)
                 + 1/K : (s1'=1) & (p1'=1) & (v1'=1) & (u1'=true);
  //read
  [read] s1=1 & u1 & !p1=v2 & c<N-1 -> (u1'=true) & (v1'=v2);
  [read] s1=1 & u1 & p1=v2 & c<N-1 -> (u1'=false) & (v1'=v2) & (p1'=0);
  [read] s1=1 & !u1 & c<N-1 -> (u1'=false) & (v1'=v2) & (p1'=0);
  [read] s1=1 & u1 & !p1=v2 & c=N-1 -> (s1'=2) & (u1'=true) & (v1'=0) & (p1'=0);
  [read] s1=1 & u1 & p1=v2 & c=N-1 -> (s1'=2) & (u1'=false) & (v1'=0) & (p1'=0);
  [read] s1=1 & !u1 & c=N-1 -> (s1'=2) & (u1'=false) & (v1'=0);
  //done
  [done] s1=2 -> (s1'=3) & (u1'=false) & (v1'=0) & (p1'=0);
  //starting a new round
  [retry] s1=2 -> (s1'=0) & (u1'=false) & (v1'=0) & (p1'=0);
  //self loop
  [loop] s1=3 -> (s1'=3);
endmodule

// construct remaining processes through renaming
module process2=process1[s1=s2,p1=p2,v1=v2,u1=u2,v2=v3] endmodule
module process3=process1[s1=s3,p1=p3,v1=v3,u1=u3,v2=v1] endmodule

system
counter || process1 || process2 || process3
endsystem

rewards
(s1=0 | s2=0 | s3=0): 1;
endrewards
```

Listing 7.1: Leader election model

Table 7.1 shows the time needed to build each system. The building times for PROMOC and PRISM are comparable for small sized instances. PROMOC outperforms PRISM for bigger sized instances. The difference in building times arise from the underlying OBDD libraries CUDD and JINC.

Table 7.2 shows the times needed to verify that the election of a leader has probability one. TADDs have been included in this measurement as verification involves the renaming of neighbored variables.

As expected, TADDs benefit from their compact size and thereby gain an run-time advantage over ADDs. The indirect comparison between JINC and CUDD provide similar results as for to the building times.

Table 7.3 shows the times for calculating the expected number of rounds to elect a leader. This calculation has not been done with PRISM because it does not provide this feature. This calculation uses the iterative methods discussed in Section 2.3 to solve the linear equation system. In this benchmark, TADDs are again faster

N	K	Model			PROMOC	PRISM
		States	Transitions	Nodes	Time (s)	Time (s)
3	2	22	29	367	0	0
3	4	135	198	1.781	0	0
3	6	439	654	5.632	0	0
3	8	1.031	1.542	10.595	0	1
3	10	2.007	3.006	23.382	1	1
3	12	3.466	5.193	29.617	1	1
3	14	5.495	8.238	51.000	1	1
3	16	8.199	12.294	69.553	1	1
4	2	55	70	908	0	0
4	4	782	1.037	10.801	1	0
4	6	3.902	5.197	58.324	7	6
4	8	12.302	16.397	165.625	8	8
4	10	30.014	40.013	473.188	19	60
4	12	62.222	82.957	929.667	18	312
5	2	136	167	1.731	2	1
5	4	4.124	5.147	41.528	5	7
5	6	31.133	38.908	337.108	16	27
5	8	131.101	163.868	1.274.313	42	413
6	2	329	392	3.163	2	1
6	4	20.524	24.619	140.735	9	12
6	6	233.340	279.995	1.732.096	115	1.207

Table 7.1: Building times for the leader election protocol.

		PROMOC		PRISM
N	K	ADD	TADD	ADD
4	2	0	0	0
4	4	1	0	0
4	6	2	1	2
4	8	2	1	6
4	10	7	2	19
4	12	7	4	42
5	2	0	1	0
5	4	3	1	2
5	6	19	12	11
5	8	48	35	315
6	2	1	1	0
6	4	11	8	12
6	6	175	133	433

Table 7.2: Calculation times to verify $P^{\geq 1}[true \mathsf{U} \bigwedge_{1 \leq i \leq N} s_i = 3]$.

N	K	ADD	TADD
4	2	0	0
4	4	1	1
4	6	1	1
4	8	1	1
4	10	2	2
4	12	3	4
5	2	1	1
5	4	1	1
5	6	1	1
5	8	4	6
6	2	3	3
6	4	10	8
6	6	88	72

Table 7.3: Times to calculate the expected number of rounds.

than ADDs. TADDs result in faster computation times because they profit from the compact representation and the better cache hit/miss ratio.

7.1.1.2 Kanban

This benchmark uses the model checking tool from [66]. This tool can use JINC as well as CUDD. As such, this benchmark can be seen as a direct comparison between JINC and CUDD. Both BDD packages do not perform any garbage collection operation. The computed-table sizes have been chosen so that no resizing took place. Dynamic reordering has been switched off.

This case study is based on the Kanban system of [29]. The system is modeled with Continuous Time Markov Chains (CTMSs). Kanban is a manufacturing system related to 'Just In Time' (JIT) production. The system signals the need for an item with the so called Kanban card. The number of Kanban cards tokens is synonymic for production stations. The Kanban system is a Pull system and was successfully devised by Toyota.

Table 7.4 shows the times needed to build the Kanban system with a various number of Kanban cards. The benchmark indicates[2] that JINC is faster than CUDD. The time difference is mainly based on the faster garbage reuse algorithm (no traversal needed), the better node handling (derivation hierarchy) and the optimized run-time behavior through extensive use of template meta programming (a result from the HOWDD framework).

7.2 Multi-Threaded JINC

This section benchmarks JINC's novel multi-threaded approaches. The benchmarks are divided into three parts. At first, parallel node allocation and garbage collection

[2] like in the comparison between PRISM and PROMOC

Kanban N	JINC Time (s)	CUDD Time (s)
4	1	1
5	1	3
6	3	6
7	4	14
8	11	33
9	19	61

Table 7.4: Times to build the Kanban system.

is measured. Second, the parallel reordering approach will be investigated in terms of variable reordering and group reordering. The last part benchmarks JINC's parallel computation abilities.

7.2.1 Parallel Node Allocation and Garbage Collection

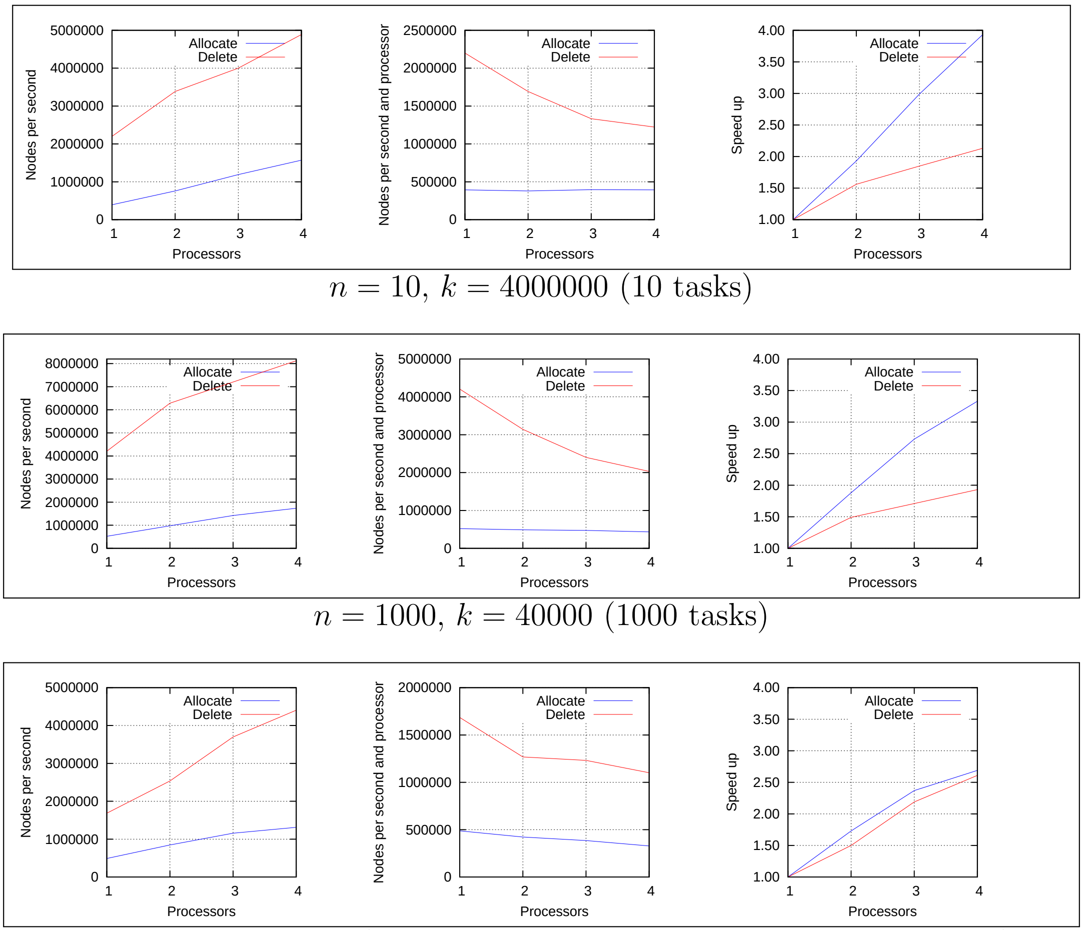
The purpose of this benchmark is to measure the maximal speed-up for node allocation and the expected speed-up for the garbage collection algorithm.

The benchmark setup uses two parameters, n for the number of variables and k for the number of nodes that are created per level. There is no dependency between levels. For level i (with variable x_i) we create the functions

$$x_i, 2 \cdot x_i, 3 \cdot x_i, \ldots, n \cdot x_i.$$

The used hardware for this benchmark consists of 4 processors with 2.5GHz and 8GB RAM.

At first we use 10 variable levels and create 4000000 nodes per level. This means that we allocate and delete 44000001 nodes (4000001 terminal nodes + $10 \cdot 4000000$ inner nodes) for the first measurement. Each level is created in a separate task, so that there is no concurrent access to the unique-tables. As a second benchmark we use 1000 variable levels and create 40000 nodes per level. This results in 40040001 total nodes. The third setup also uses 1000 variable levels and creates 40000 nodes per level, but changes the creation order. Task i creates the functions $i \cdot x_1, \ldots, i \cdot x_k$ ($i = 1, \ldots, n$). That means that there are many concurrent accesses to unique-tables. Figure 7.1 illustrates the results for all three benchmarks. In the first two benchmarks the number of nodes that are allocated per second and processor is almost constant. The garbage collection algorithm does not benefit in the equal amount from parallel threads as in the case of node allocation. This is due to the fact that the garbage collection has several phases which are separated via wait points. This reduces the effect of multiple parallel threads. In the third benchmark the performance of the allocation decreases, i.e., the concurrency increases with an increasing number of processors. The performance of the garbage collection decreased (compared to the second benchmark) dramatically. This is due to the



$n = 10$, $k = 4000000$ (10 tasks)

$n = 1000$, $k = 40000$ (1000 tasks)

$n = 1000$, $k = 40000$ (40000 tasks because of changed creation order)

Figure 7.1: Results for different allocation and garbage collection benchmarks.

fact that the nodes of a level are allocated from different memory pools. Because of this memory fragmentation the processor data cache cannot work very efficiently. Instead of fetching the data from the fast cache memory the data must be fetched from the much slower main memory. The increased execution time of the garbage collection makes the influence of the synchronization points less relevant.

7.2.2 Parallel Reordering

This benchmark is used to show how well the newly developed parallel reordering concept performs under real conditions [50].

7.2.2.1 Variable Reodering

The used benchmark function is the n-queens problem. This function is defined over many variables[3] and thereby a good candidate for parallel swap operations.

[3] the n-queens problem uses n^2 variables to represent the chess board

The used reordering algorithm is the genetic reordering algorithm presented in [69]. As described in Section 6.6 genetic algorithms are well suited for this new approach because of the high number of independent swap operations. Reordering algorithms like e.g., sifting use variable shifting which cannot profit from the new parallel reordering approach. There is no time penalty in using the parallel reordering approach for reordering methods like sifting, window permutation, etc as the swap scheduling is performed in parallel.

Figure 7.2 shows the parallel reordering approach applied to different instances of the n-queens problem. The x-axis represents the time needed to run a complete genetic reordering algorithm[4]. The y-axis represents the number of CPUs that are fully used[5]. The used hardware consists of 16 processors with 3.5GHz and 4GB RAM. The thread-pool has been limited to 16 threads. The benchmark shows that the new reordering approach scales with increasing number of variables. This can be explained with the increasing number of possible parallel operations and that the rescheduling is fast compared to a single swap operation. For the smaller examples the rescheduling process is more time consuming compared to a single swap operation. The larger examples take advantage of this new approach. The reordering of the 10-queens problem is nearly 3 times faster compared to the single threaded approach. This new approach speeds up the 11-queens problem reordering by factor 7 (700% processor load in average) and the 12-queens problem reordering by factor 10.

Figure 7.3 shows the number of possible parallel threads for the different instances of the n-queens problem. It illustrates that even the smaller benchmarks have a large number of parallel swap operations. As stated above the rescheduling and thread-pool management slows the overall performance. Another reason is that the number of nodes on each level is low and thereby the swap operations are not very time consuming. For the larger examples the number of possible parallel threads is higher than the number of CPUs used. This benchmark shows that even a small number of variables results in a high number of parallel threads. The average number of parallel threads for this benchmark is approximately $\frac{n^2}{10}$.

7.2.2.2 Group Reordering

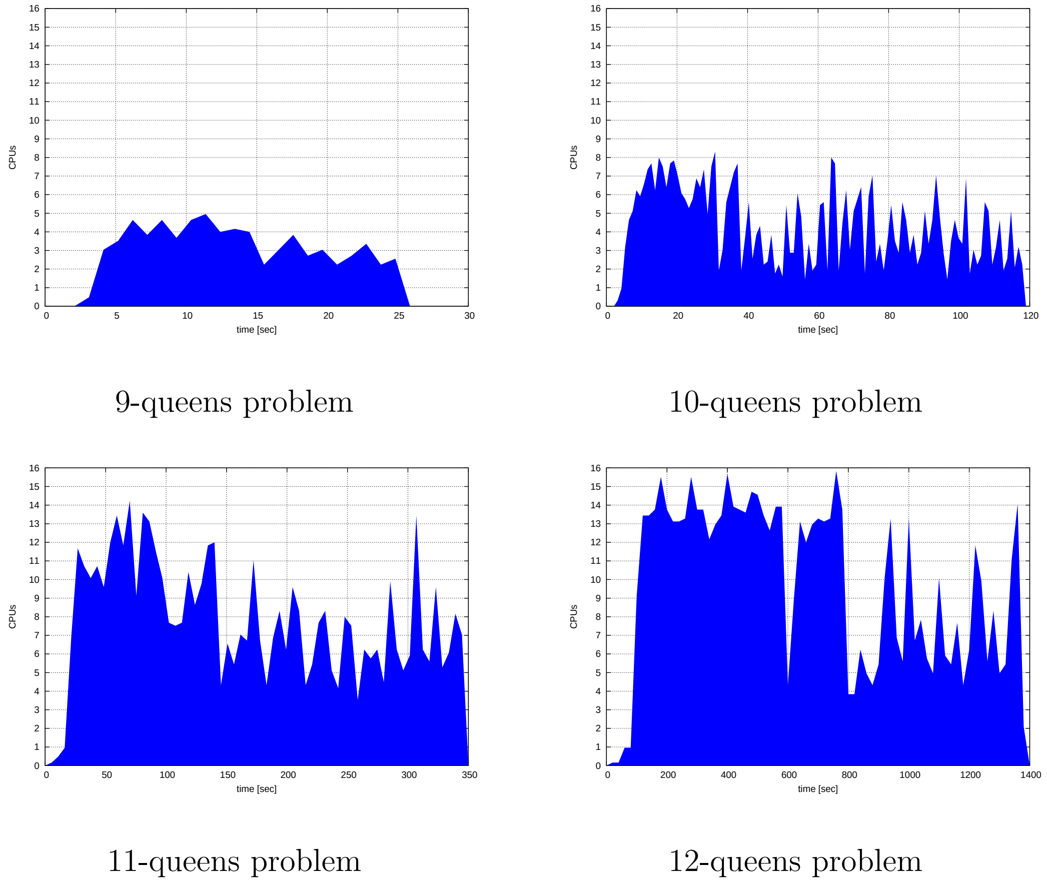
Group reordering increases the number of independent swap operations (see Section 6.6 for more details). In this case all reordering algorithms can profit from the new parallel reordering approach.

For this benchmark we use the n-queens problem and group the variables of a row into one group, i.e., there are n groups and each group consists of n variables. The sifting algorithm is the used reordering method.

Figure 7.4 shows the group sifting results for different instances of the n-queens problem. The advantage of parallel computation depends on the group size, e.g.,

[4] The algorithm generates 20 individuals per generation and is repeated until there are no further improvements for the last 10 generations.

[5] e.g., 10 threads that are using 50% single CPU time count as 5 fully used CPUs.



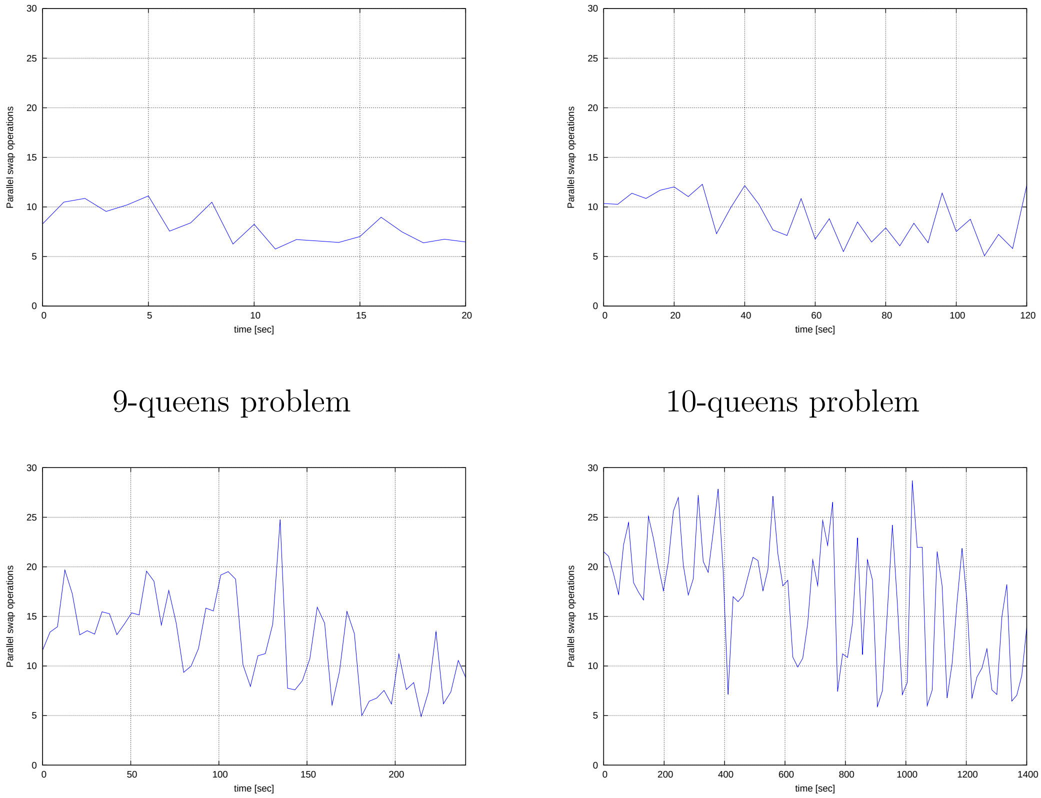
9-queens problem

10-queens problem

11-queens problem

12-queens problem

Figure 7.2: Parallel reordering approach applied to different n-queens problem instances



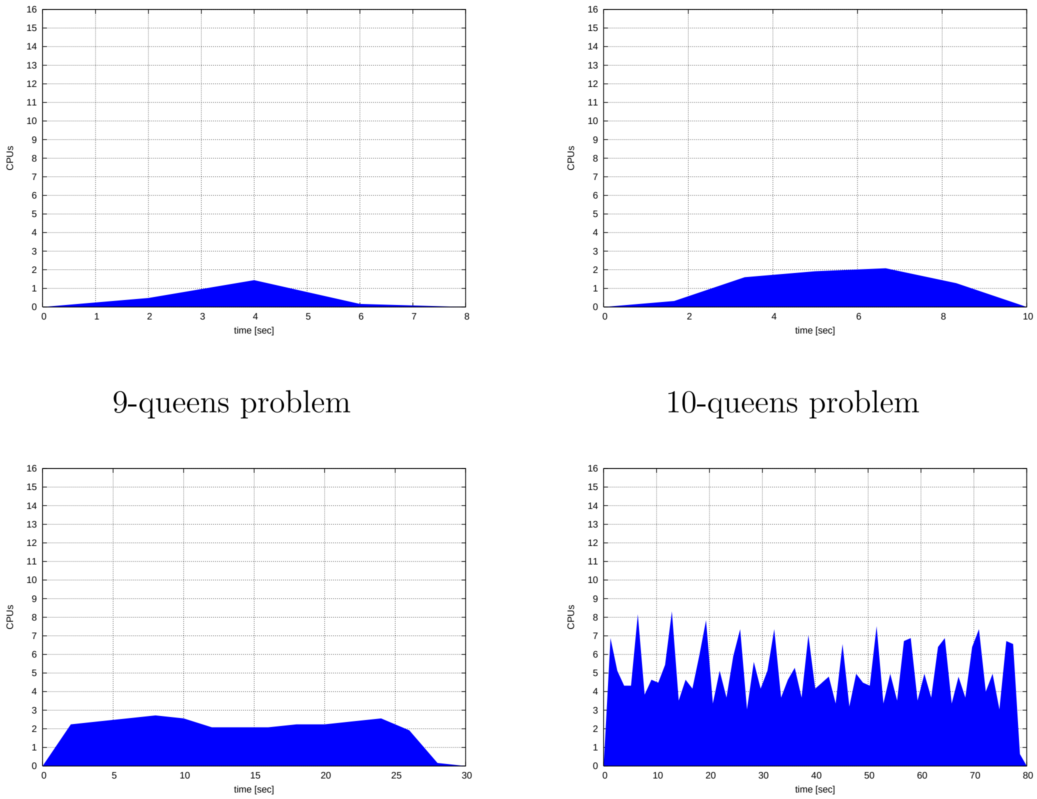
9-queens problem

10-queens problem

11-queens problem

12-queens problem

Figure 7.3: Number of parallel swap operations for different n-queens problem instances



9-queens problem

10-queens problem

11-queens problem

12-queens problem

Figure 7.4: Parallel group reordering approach applied to different n-queens problem instances

the 12-queens problem group reordering is sped up by factor 4.5, where a maximal speed-up of 6.26 is possible (see Section 6.6).

7.2.3 Parallel Computation

This benchmark measures JINC's parallel computation abilities in the application of matrix power computation. The basic idea of this benchmark is to calculate the matrix powers $A^2, \ldots, A^n$ with given square matrix A and natural number $n > 1$. Each computation step is a calculation intensive process and therefore well suited to be calculated with parallel processes. The calculations are segmented into several steps. Table 7.5 illustrates the step concept. After each step the garbage collection algorithm is invoked to clean up memory used for temporary calculations.

Step	Matrix Powers			
1	A^1			
2	$A^2 = A^1 \cdot A^1$			
3	$A^4 = A^2 \cdot A^2$	$A^3 = A^2 \cdot A^1$		
4	$A^8 = A^4 \cdot A^4$	$A^7 = A^4 \cdot A^3$	$A^6 = A^4 \cdot A^2$	$A^5 = A^4 \cdot A^1$
$\vdots$	$\dots$			

Table 7.5: Step concept of matrix power calculation.

Step	1 Processor Time (s)	16 Processors Time (s)	Speed-up
1	$0,27$	$0,27$	0 %
2	$0,33$	$0,31$	6 %
3	$0,50$	$0,45$	11 %
4	$0,62$	$0,50$	24 %
5	$0,74$	$0,67$	10 %
6	$5,07$	$1,97$	157 %
7	$1.119,34$	$214,41$	422 %

Table 7.6: Matrix power calculation with 1 and 16 processors.

The benchmark matrix $A \in \mathbb{R}^{256\times256}$ has the following form:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & & \\ 1 & \ddots & \ddots & \ddots & \ddots & \\ 2 & \ddots & \ddots & \ddots & \ddots & 3 \\ 3 & \ddots & \ddots & \ddots & \ddots & 2 \\ & \ddots & \ddots & \ddots & \ddots & 1 \\ & & 3 & 2 & 1 & 0 \end{pmatrix}$$

We will run 7 steps, i.e., the matrix powers $A^2, \dots A^{64}$ will be calculated. The last step invokes 32 threads in parallel. Figure 7.5 shows the processor load over time. In this benchmark after finishing 16 tasks in step 7 there are no more waiting tasks in the thread-pool, so that with every finished task the processor load decreases. Table 7.6 shows the calculation times for every step for the single-threaded and multi-threaded setup. The speed-up for the calculation of step 7 is around factor 5.22 with 16 processors compared to the computation with one processor (equivalent to single-threaded JINC).

7.2.4 Generic Multi-Operand `APPLY`

This section discusses the advantages of the expression template approach in the context of recursive HOWDD manipulation. The used benchmark is the well known

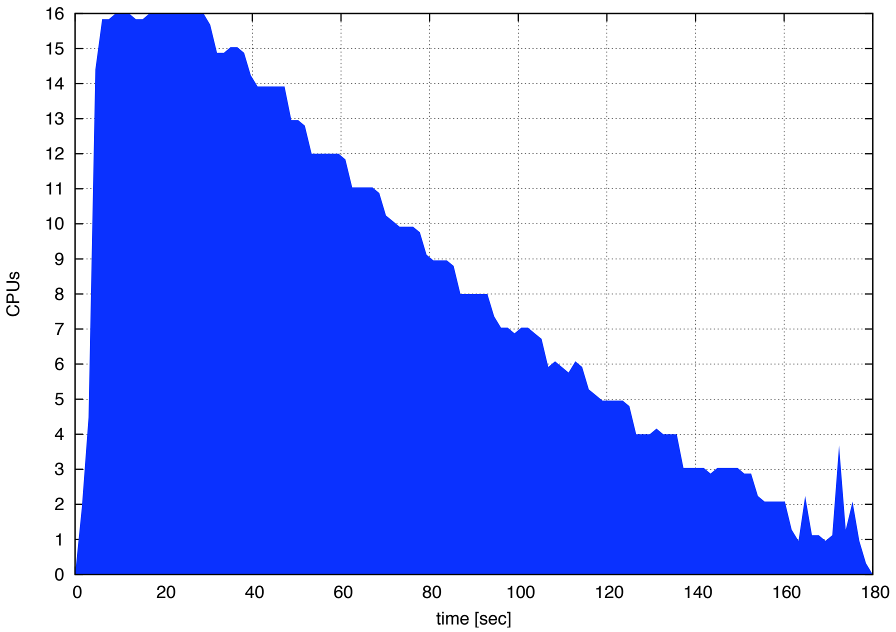


Figure 7.5: Matrix power processor load.

n-queen problem. This benchmark is well suited because it can be expressed as one expression. Note, there is no caching for the novel expression template approach. The regular manipulation algorithm uses computed tables. In comparison to Section 5.3 we will just measure the building time.

Table 7.7 compares the traditional approach with the novel generic multi-operand `APPLY` algorithm. The number of peak nodes illustrates the efficiency of this new approach. The peak nodes include the nodes representing the board positions and the resulting function. The overhead (including board position representations) is illustrated in Figure 7.6. It can be seen that there is no overhead for the new algorithm while keeping the efficient recursive structure (compared to [51, 27]). The run-time improvements are a result from early termination and the optimal number of nodes. For the largest benchmark instance it was not possible to fit into $4Gb$ of memory[6] with the traditional approach. The new approach reduce the necessity to use caching mechanisms or garbage collection. This is important as caching and garbage collection is inefficient in an multi-threaded environment.

[6] no garbage collection took place to illustrate the overhead of the traditional approach

			Traditional APPLY		Multi-Operand APPLY	
n	Solutions	Nodes	Time(s)	Peak Nodes	Time(s)	Peak Nodes
4	2	31	0,01	423	0,01	170
5	10	169	0,01	1.629	0,01	459
6	4	131	0,01	5.183	0,01	678
7	40	1.101	0,01	19.216	0,01	1.995
8	92	2.453	0,04	66.421	0,02	3.831
9	352	9.559	0,20	251.989	0,09	11.571
10	724	25.947	0,92	995.789	0,35	28.759
11	2.680	94.824	4,59	4.367.315	1,72	98.626
12	14.200	435.172	29,79	21.057.439	9,51	440.169
13	73.712	2.044.396	306,09	109.672.282	56,10	2.050.818
14	365.596	9.572.420	∞	>4Gb	380,58	9.580.511
15	2.279.184	51.889.031	∞	>4Gb	2.922,75	51.899.061

Table 7.7: Comparison between traditional and generic multi-operand APPLY

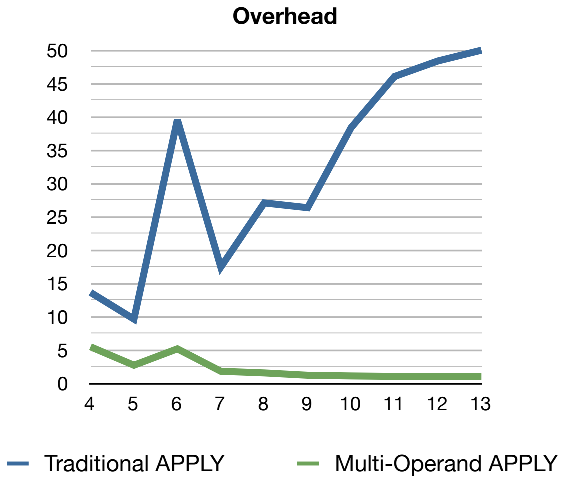


Figure 7.6: Overhead comparison including board representation

Chapter 8 Conclusion

The framework of OWDDs unifies the different OBDD variants to reason about canonicity and to identify the similarities of several variants. In various applications, it has been shown that OBDD variants are well suited for a wide range of applications. The special requirements for matrix representation and model checking yield the basis to develop a new, more generic framework. The development of HOWDDs made it possible to implement an efficient OBDD library and to develop a new OBDD variant, TADD, that is more compact for matrix representation and with which renaming of variable pairs can be performed in constant time.

The implementation of JINC follows traditional approaches and extends them to support the idea of HOWDDs. With this design it is possible to implement new instances rapidly. JINC makes use of multi-core architectures to speed-up function manipulations, the garbage collection phase and reordering algorithms. The novel approaches are implemented in a way that JINC's multi-threaded version is superior to the single-threaded version published in [88]. If JINC is used on a single-core architecture it is as fast as the single-threaded version. Parallel packages like [78, 111] had significant overhead compared to their single process basis implementation. This is due to the fact that their parallel approach changes the algorithmic structure. JINC has not altered the recursive structure of the algorithms, benefits from shared data structures and uses almost no blocking algorithms.

Many steps were necessary to achieve JINC's goal to provide a highly efficient OBDD library. JINC is written in C++ which has been shown to be an efficient programming language for high performance computing. The realization of JINC's design concept has been eased with the use of templates, policy-based design and state of the art programming techniques. In several benchmarks, it has been shown that the single-threaded version of JINC is faster than CUDD. The difference can be explained by the efficient implementation and the extensive use of templates.

The new multi-threading approaches speed-up the OBDD manipulations and reordering algorithms by several factors compared to JINC's single-threaded version. The multi-operand `APPLY` algorithm eliminates the creation of temporary nodes. This is especially important in a multi-threaded environment as caching and garbage collection is not as efficient as in a single-threaded environment. The elimination of temporary nodes reduces the need for caching and garbage collection. The reduction of the required memory for manipulations enables the computation of larger models. Because of the space-efficiency of this new approach, the run-time performance is also increased compared to the traditional `APPLY` algorithm.

The benchmarks indicate that the new approaches speed-up the computation and reordering algorithms. This together with the space-efficiency leads to the conclusion that many applications (e.g., unbounded model checking [122, 61, 94]) should be reviewed if they could benefit from the new multi-threading multi-operand approaches introduced and discussed in this thesis.

List of Figures

List of Tables

Listings

Bibliography

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[2] A. Agarwal. Performance tradeoffs in multithreaded processors. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1991.

[3] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27:509–516, 1978.

[4] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, February 2001.

[5] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[6] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990.

[7] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proc. IEEE/ACM International Conference on CAD*, pages 188–191. IEEE Computer Society Press, 1993.

[8] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2-3):171–206, 1997.

[9] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, 1977.

[10] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[11] F.W.M. Bezzel. Eight queens problem. *Berliner Schachzeitung*, 3:636, 1848.

[12] F. Bianchi, Fulvio Corno, Maurizio Rebaudengo, Matteo Sonza Reorda, and Roberto Ansaloni. Boolean function manipulation on a parallel system using bdds. In *HPCN Europe '97: Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, pages 916–928, London, UK, 1997. Springer-Verlag.

[13] Richard Bird, Geraint Jones, and Oege De Moor. More haste, less speed: lazy versus eager evaluation. *J. Funct. Program.*, 7(5):541–547, 1997.

[14] Tobias Blechmann and Christel Baier. Checking equivalence for reo networks. *Electron. Notes Theor. Comput. Sci.*, 215:209–226, 2008.

[15] Robert D. Blumofe. *Executing multithreaded programs efficiently.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995.

[16] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *In Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS*, pages 356–368, 1994.

[17] Beate Bollig, Martin Lobbing, and Ingo Wegener. Simulated annealing to improve variable orderings for obdds. In *In Int'l Workshop on Logic Synth*, pages 5–5, 1995.

[18] Beate Bollig and Ingo Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Comput.*, 45(9):993–1002, 1996.

[19] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a bdd package. In *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 40–45, New York, NY, USA, 1990. ACM.

[20] Ulrich Breymann. *Designing Components with the C++ STL.* Pearson Education Ltd., 3rd (electronic) edition, 2002.

[21] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[22] R. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with applications to integer multiplication. *IEEE Transactions on Computers*, 40:205–213, 1991.

[23] Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits with binary moment diagrams. In *Design Automation Conference*, pages 535–541, 1995.

[24] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective.* Prentice Hall, us ed edition, August 2002.

[25] Kenneth M. Butler, Don E. Ross, Rohit Kapur, and M. Ray Mercer. Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 417–420, New York, NY, USA, 1991. ACM.

[26] Shan-Tai Chen, Shun-Shii Lin, Li-Te Huang, and Chun-Jen Wei. Towards the exact minimization of bdds—an elitism-based distributed evolutionary algorithm. *Journal of Heuristics*, 10(3):337–355, 2004.

[27] Yirng-An Chen, Bwolen Yang, and Randal E. Bryant. Breadth-first with depth-first BDD construction: A hybrid approach. In *Proceedings of the 34st Conference on Design Automation*, 1997. Submitted to 1997 ACM/IEEE Design Automation Conference.

[28] Pi-Yu Chung, Ibrahim N. Hajj, and Janak H. Patel. Efficient variable ordering heuristics for shared ROBDD. In *ISCAS*, pages 1690–1693, 1993.

[29] G. Ciardo and M. Tilgner. On the use of Kronecker operators for the solution of generalized stocastic Petri nets. ICASE Report 96-35, Institute for Computer Applications in Science and Engineering, 1996.

[30] E. Clarke, M. Fujita, P. McGeer, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *Proc. International Workshop on Logic Synthesis (IWLS)*, 1993.

[31] E. Clarke, M. Fujita, and X. Zhao. Multi-terminal binary decision diagrams and hybrid decision diagrams. *in [104]*, pages 93–108, 1996.

[32] J. W. Counts. ACM Algorithm 126: Gauss' method. *Communications of the ACM*, 5(10):511, October 1962.

[33] Armin B. Cremers and Thomas N. Hibbard. Mutual exclusion of n processors using an o(n)-valued message variable (extended abstract). In *Proceedings of the Fifth Colloquium on Automata, Languages and Programming*, pages 165–176, London, UK, 1978. Springer-Verlag.

[34] Armin B. Cremers and Thomas N. Hibbard. Axioms for concurrent processes. In *New Results and New Trends in Computer Science*, pages 54–68, London, UK, 1991. Springer-Verlag.

[35] Y. Orlarey D. Fober, S. Letz. Lock-free techniques for concurrent access to shared objects. In GMEM, editor, *Actes des Journées d'Informatique Musicale JIM2002, Marseille*, pages 143–150, 2002.

[36] R. Drechsler and N. Gockel. Minimization of bdds by evolutionary algorithms, 1997.

[37] Rolf Drechsler, Nicole Gockel, and Bernd Becker. Learning heuristics for obdd minimization by evolutionary algorithms. In *In Parallel Problem Solving from Nature, LNCS 1141*, pages 730–739, 1996.

[38] Rolf Drechsler and Wolfgang Gunther. Using lower bounds during dynamic BDD minimization. In *Design Automation Conference*, pages 29–32, 1999.

[39] John English. Multithreading in c++. *SIGPLAN Not.*, 30(4):21–28, 1995.

[40] John E. Faust and Henry M. Levy. The performance of an object-oriented threads package. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 278–288, New York, NY, USA, 1990. ACM.

[41] J. Franel. *n*- Queens solution. *Intermédiaire des mathématiciens*, page 140/141, 1894.

[42] Hiroshige Fujii, Goichi Ootomo, and Chikahiro Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 38–41, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[43] M. Fujita, P. McGeer, and J. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2-3):149–169, 1997.

[44] Masahiro Fujita, Yusuke Matsunaga, and Taeko Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *EURO-DAC '91: Proceedings of the conference on European design automation*, pages 50–54, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[45] S. Gai, M. Rebaudengo, and M. Sonza Reorda. A data parallel algorithm for boolean function manipulation. In *FMPSC: Frontiers of Massively Parallel Scientific Computation*. National Aeronautics and Space Administration (NASA), IEEE Computer Society Press, 1995.

[46] Anders Gidenstam, Marina Papatriantafilou, Hakan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *ispan*, 00:202–207, 2005.

[47] J.W.L. Glaisher. On the problem of the eight queens. *Edin. Philosophical Magazine Sec. 4*, 48:457–467, 1874. In 1874 J. W. Glaisher proposed expanding the Eight Queens Problem to the n-queens problem, that is, solving the queens' puzzle for the general nxn chessboard. For example, the well-known n-queens problem can be tackled by noting that the eight geometric symmetries of the problem translate into an invariance group of the set of clauses; this reduces the search space, as was noted already by Glaisher.

[48] R. Govindarajan, S. S. Nemawarkar, and P. LeNir. Design and performance evaluation of a multithreaded architecture. In *HPCA '95: Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, page 298, Washington, DC, USA, 1995. IEEE Computer Society.

[49] Orna Grumberg, Shlomi Livne, and Shaul Markovitch. Learning to order bdd variables in verification. *Journal of Artificial Intelligence Research*, 18:2003, 2003.

[50] Justin E. Harlow and Franc Brglez. Design of experiments in BDD variable ordering: lessons learned. In *ICCAD*, pages 646–652, 1998.

[51] A. Hett, R. Frechsler, and B. Andbecker. More: Alternative implementation of bdd-packages by multi-operand synthesis. In *Proceedings of the European Design Automation Conference*, 1996.

[52] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[53] S. Höreth and R. Drechsler. Dynamic minimization of word-level decision diagrams. In *DATE '98: Proceedings of the conference on Design, automation and test in Europe*, pages 612–617, Washington, DC, USA, 1998. IEEE Computer Society.

[54] Alan J. Hu, David L. Dill, Andreas Drexler, and C. Han Yang. Higher-level specification and verification with bdds. In *CAV '92: Proceedings of the Fourth International Workshop on Computer Aided Verification*, pages 82–95, London, UK, 1993. Springer-Verlag.

[55] William Hung and Xiaoyu Song. Bdd variable ordering by scatter search. In *ICCD '01: Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, page 368, Washington, DC, USA, 2001. IEEE Computer Society.

[56] Shin ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *DAC '93: Proceedings of the 30th international conference on Design automation*, pages 272–277, New York, NY, USA, 1993. ACM.

[57] Shin ichi Minato. *Binary decision diagrams and applications for VLSI CAD.* Kluwer Academic Publishers, 1996.

[58] A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1), 1990.

[59] Mizuho Iwaihara and Yusaku Inoue. Bottom-up evaluation of logic programs using binary decision diagrams. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 467–474, Washington, DC, USA, 1995. IEEE Computer Society.

[60] T. Kam, T. Villa, R. Brayton, and A. SagiovanniVincentelli. Multi-valued decision diagrams: Theory and applications. *Multiple-Valued Logic: An International Journal*, 4(1-2):9–62, 1998.

[61] Hyeong-Ju Kang and In-Cheol Park. Sat-based unbounded symbolic model checking. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 840–843, New York, NY, USA, 2003. ACM.

[62] Sascha Klüppelholz and Christel Baier. Symbolic model checking for channel-based component connectors. *Electron. Notes Theor. Comput. Sci.*, 175(2):19–37, 2007.

[63] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In *Computer Performance Evaluation / TOOLS*, pages 200–204, 2002.

[64] Oxford University Computing Laboratory. Prism case studies.

[65] K. Lampka. *A symbolic approach to the state graph based analysis of high-level Markov Reward Models*. PhD thesis, University Erlangen-Nuremberg, 2007.

[66] K. Lampka and M. Siegle. Symbolic Activity-Local State Graph Generation in the Context of Moebius. In *Proc. of the Satelite Workshop on Stochastic Petri Nets and related Formalisms at the 30'th International Colloquium on Automata, Languages and Programming, Eindhoven, Netherlands*, June 28-29 2003.

[67] Kai Lampka, Markus Siegle, Jörn Ossowski, and Christel Baier. Partially-shared zero-suppressed multi-terminal bdds: Concept, algorithms and applications. Technical report, 2008.

[68] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, vol. 38:985–999, 1959.

[69] Wolfgang Lenders and Christel Baier. Genetic algorithms for the variable ordering problem of binary decision diagrams. In *FOGA*, volume 3469 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2005.

[70] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, page 2, New York, NY, USA, 2007. ACM.

[71] Tong Li, Alvin R. Lebeck, and Daniel J. Sorin. Quantifying instruction criticality for shared memory multiprocessors. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 128–137, New York, NY, USA, 2003. ACM.

[72] Jørn Lind-Nielsen. Buddy - a binary decision diagram package version 2.2. `http://sourceforge.net/projects/buddy`, 2002.

[73] Yibei Ling, Tracy Mullen, and Xiaola Lin. Analysis of optimal thread pool size. *SIGOPS Oper. Syst. Rev.*, 34(2):42–55, 2000.

[74] Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo. *C++ Primer (4th Edition)*. Addison-Wesley Professional, 2005.

[75] Elsa Loekito and James Bailey. Are zero-suppressed binary decision diagrams good for mining frequent patterns in high dimensional datasets? In Peter Christen, Paul J. Kennedy, Jiuyong Li, Inna Kolyshkina, and Graham J. Williams, editors, *Sixth Australasian Data Mining Conference (AusDM 2007)*, volume 70 of *CRPIT*, pages 139–150, Gold Coast, Australia, 2007. ACS.

[76] Cameron McNairy and Rohit Bhatia. Montecito: A dual-core, dual-thread Itanium processor. *IEEE Micro*, 25(2):10–20, March/April 2005.

[77] Maged M. Michael and Michael L. Scott. Implementation of atomic primitives on distributed shared memory multiprocessors. In *Proc. First Symp. on High Performance Computer Architecture*, pages 222–231, 1995.

[78] Kim Milvang-Jensen and Alan J. Hu. BDDNOW: A parallel BDD package. In *Formal Methods in Computer-Aided Design*, pages 501–507, 1998.

[79] Masaaki Mizuno and Liubo Chen. A structured methodology to develop concurrent programs for a thread-pool model in object-oriented systems. Technical report, Kansas State University.

[80] Nauck. First complete solution of eight queens problem. *Leipziger Illustrierte Zeitung*, 361:352, 1850. Franz Nauck outlined the first complete solution of the 8x8 chessboard, consisting of 92 solutions, in the Leipzig Illustrierte Zeitung in 1850.

[81] Ziv Nevo and Monica Farkash. Distributed dynamic bdd reordering. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 223–228, New York, NY, USA, 2006. ACM.

[82] Ziv Nevo and Monica Farkash. Distributed dynamic bdd reordering. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 223–228, New York, NY, USA, 2006. ACM.

[83] Hiroyuki Ochi, Nagisa Ishiura, and Shuzo Yajima. Breadth-first manipulation of sbdd of boolean functions for vector processing. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 413–416, New York, NY, USA, 1991. ACM.

[84] Jörn Ossowski. Symbolic representation and manipulation of discrete functions. Master's thesis, University of Bonn, 2004.

[85] Jörn Ossowski. Jinc is a fast, object-oriented and multi-threaded obdd library. `http://www.jossowski.de`, 2009.

[86] Jörn Ossowski. Promoc a symbolic probabilistic model checker based on jinc. `http://www.jossowski.de`, 2009.

[87] Jörn Ossowski and Christel Baier. Symbolic reasoning with weighted and normalized decision diagrams. In *Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus 2005)*, volume 151, pages 39–56. Electronic Notes in Theoretical Computer Science, 2006.

[88] Jörn Ossowski and Christel Baier. A uniform framework for weighted decision diagrams and its implementation. *STTT*, 10(5), 2008.

[89] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, August 12 1985.

[90] Shipra Panda and Fabio Somenzi. Who are the variables in your neighborhood. In *ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 74–77, Washington, DC, USA, 1995. IEEE Computer Society.

[91] Gregory M. Papadopoulos and Kenneth R. Traub. Multithreading: a revisionist view of dataflow architectures. In *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture*, pages 342–351, New York, NY, USA, 1991. ACM.

[92] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems.* PhD thesis, University of Birmingham, 2002.

[93] Thuan Quang Pham. The experimental migration of a distributed application to a multithreaded environment. Thesis (m.s.), Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, USA, 1991.

[94] Florian Pigorsch, Christoph Scholl, and Stefan Disch. Advanced unbounded model checking based on aigs, bdd sweeping, and quantifier scheduling. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 89–96, Washington, DC, USA, 2006. IEEE Computer Society.

[95] K. Randall. Cilk: Efficient multithreaded computing. Technical report, Cambridge, MA, USA, 1998.

[96] Ram Rangan. *Pipelined Multithreading Transformations and Support Mechanisms.* PhD thesis, Princeton University, June 2007.

[97] Rajeev K. Ranjan, Wilsin Gosti, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Dynamic reordering in a breadth-first manipulation based BDD package: Challenges and solutions. In *International Conference on Computer Design*, pages 344–351, 1997.

[98] Gabriel Dos Reis and Bjarne Stroustrup. Specifying c++ concepts. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308, New York, NY, USA, 2006. ACM.

[99] Juan Rodríguez-Hortalá and Jaime Sánchez-Hernández. Functions and lazy evaluation in prolog. *Electron. Notes Theor. Comput. Sci.*, 206:153–174, 2008.

[100] Don E. Ross, Kenneth M. Butler, Rohit Kapur, and M. Ray Mercer. Fast functional evaluation of candidate obdd variable orderings. In *EURO-DAC '91: Proceedings of the conference on European design automation*, pages 4–10, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[101] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[102] N. Ishiura S. Minato and S. Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 52–57, 1990.

[103] Jagesh V. Sanghavi, Rajeev K. Ranjan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. High performance bdd package by exploiting memory hierarchy. *dac*, 00:635–641, 1996.

[104] T. Sasao and M. Fujita, editors. *Representation of Discrete Functions*. Kluwer Academic Publishers, 1996.

[105] Minato Shin-ichi. Zero-suppressed bdds and their applications. *STTT*, 3(2):156–170, 2001.

[106] Jeremy Siek and Walid Taha. A semantic analysis of c++ templates. In Thomas [116], pages 304–327.

[107] Detlef Sieling. The nonapproximability of obdd minimization. *Information and Computation*, 172:103–138, 1998.

[108] Detlef Sieling. Variable orderings and the size of obdds for random partially symmetric boolean functions. *Random Struct. Algorithms*, 13(1):49–70, 1998.

[109] Fabio Somenzi. Cudd: Colorado university decision diagram package release 2.4.2. `http://vlsi.colorad.edu/~fabio/CUDD`, 2009.

[110] Jeffrey M. Squyres. Processes, processors, and MPI, oh my! *ClusterWorld Magazine, MPI Mechanic Column*, 2(1), January 2004.

[111] Tony Stornetta and Forrest Brewer. Implementation of an efficient parallel bdd package. *dac*, 00:641–644, 1996.

[112] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[113] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[114] P. Tafertshofer and M. Pedram. Factored edge-valued binary decision diagrams. *Formal Methods in System Design*, 10(2-3):243–270, 1997.

[115] Seiichiro Tani, Kiyoharu Hamaguchi, and Shuzo Yajima. The complexity of the optimal variable ordering problems of shared binary decision diagrams. In *ISAAC '93: Proceedings of the 4th International Symposium on Algorithms and Computation*, pages 389–398, London, UK, 1993. Springer-Verlag.

[116] Dave Thomas, editor. *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*. Springer, 2006.

[117] Christel Baier Tobias Blechmann. Checking equivalence for reo networks. In *Proc. of the 4th Internation Workshop on Formal Aspects of Component Software (FACS 2007)*, 2007.

[118] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[119] S. Vrudhula, M. Pedram, and Y. Lai. Edge-valued binary decision diagrams. *in [104]*, pages 109–132, 1996.

[120] John Whaley. *Context-sensitive pointer analysis using binary decision diagrams*. PhD thesis, Stanford University, Stanford, CA, USA, 2007. Adviser-Monica Lam.

[121] Wikipedia. Thread-local storage — wikipedia, the free encyclopedia, 2009. [Online; accessed 15-February-2009].

[122] Poul Frederick Williams, Armin Biere, Edmund M. Clarke, and Anubhav Gupta. Combining decision diagrams and sat procedures for efficient symbolic model checking. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 124–138, London, UK, 2000. Springer-Verlag.

[123] Bwolen Yang and David R. O'Hallaron. Parallel breadth-first BDD construction. In *In Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 145–156, June 1997.