The JJS-BDD Library 0.1

Sascha Klüppelholz
klueppel@cs.uni-bonn.de
&
Jörn Ossowski
ossowsk@cs.uni-bonn.de

Institute of Computer Science
Department I
Prof. Dr. Christel Baier
Rheinische Friedrich-Wilhelms-Universität Bonn
Germany

27th May 2004

Contents

Foreword

We started programming a BDD library 2002 as a part of a practical. After that practical we tried to improve the performance of the library and implemented a graphical user interface (GUI). With the concept of that library we reached the limits of the library fast. Instead of trying to resolve the problems within this structure, we decided to restructure the whole library. The main attention was turned to make the library expandable easily. With the new structure we achieved a big step in performance. The drawback of the restructuring was that we could not use the code of the old GUI.
With this library an user should have a good tool for manipulating discrete functions.

Sascha Klüppelholz and Jörn Ossowski

Chapter 1

Introduction

In this manual the JJS binary decision diagram library will be introduced. The most important development goals were to keep it clearly arranged, efficient, and user friendly. A very simple API was designed, which is easy to understand and use. The JJS-BDD package is an object oriented C++ library. It is open source and freely available, so everyone should easily be able to install, use, and even extend it.
Decision diagrams, in particular ordered binary decision diagrams [Bry86], have a long tradition in computer science; e.g. in the context of VLSI design, verification of reactive systems and complexity theory [McM93, Min96]. The more BDDs became important for academic research and industrial applications, the more BDDs have been established in the curriculum for computer science at universities (c.f. the variety of textbooks that treat BDDs [Min96, CM98, EC00]).
Many problems in the design and verification of discrete functions can be expressed as sequences of boolean functions. The performance of manipulating boolean functions depends on the underlying data structure. It is thus a benchmark for that data structure.
Binary Decision Diagrams (BDDs) are an efficient data structure for symbolic boolean manipulation and are commonly used for boolean function representation. In this chapter we want to introduce the basic concept of BDDs.

1.1 Binary decision diagrams

A BDD is a directed acyclic graph with two kinds of terminal nodes (which we will call the zero drain and the one drain). Each non terminal node has two outgoing edges ($succ_0$ and $succ_1$) and a corresponding input variable. An ordered BDD (OBDD) is a BDD where input variables appear in a fixed order on all paths of the graph. No variable appears more than once on a path (e.f. see figure 1.1).

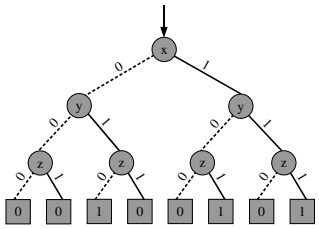


Figure 1.1: A binary decision diagram

Notations: Let $\mathcal{Z} = \{z_1, \ldots, z_n\}$ be a finite set of pairwise disjunctive boolean variables. The variable ordering of $\mathcal{Z}$ is an ordered tuple

$$\pi = (z_1, \ldots, z_n),$$

that contains every variable exactly once. For every two variables in π the following holds:

$$z_i \leq z_j \Leftrightarrow i \leq j$$

An evaluation of $\mathcal{Z}$ is a map

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

that assigns every variable $z \in \mathcal{Z}$ to a value $\eta(z) \in \{0, 1\}$.

$$\text{Eval}(\mathcal{Z})$$

identifies the set of all evaluations of $\mathcal{Z}$.
Let $\overline{a} = a_1, \ldots, a_n \in \{0, 1\}$ and $\overline{z} = z_1, \ldots, z_n \in \mathcal{Z}$ then $[\overline{z} := \overline{a}]$ represents the evaluation $\eta \in \text{Eval}(\mathcal{Z})$ with

$$\eta(z_i) = a_i,\ i = 1, \ldots, n.$$

A switching function (over $\mathcal{Z}$) is a map

$$f : \text{Eval}(\mathcal{Z}) \to \{0, 1\}.$$

The set of all switching functions over $\mathcal{Z}$ will be called $\mathbb{B}(\mathcal{Z})$.

Definition: Let $f \in \mathbb{B}(\mathcal{Z}), \overline{z} = \{z_{i_1}, \ldots, z_{i_r}\} \subseteq \mathcal{Z}$ and $\overline{b} \in \{0,1\}^r$. The cofactor of function f with respect to $\overline{z}$ is defined by

$$f|_{\overline{z}:=\overline{b}}(\eta) = f(\eta\left[\overline{z} := \overline{b}\right])$$

with

$$\eta\left[\overline{z} := \overline{b}\right](z) = \eta\left[z_{i_1} := b_1, \ldots, z_{i_r} := b_r\right](z) = \begin{cases} b_j & \text{if } z = z_{i_j} \\ \eta(z) & \text{if } z \notin \{z_{i_1}, \ldots, z_{i_r}\} \end{cases}.$$

Let $f \in \mathbb{B}(\mathcal{Z})$ and $z \in \mathcal{Z}$. The function f can be represented by the Shannon expansion:

$$\boxed{f = \neg z \cdot f|_{z=0} \vee z \cdot f|_{z=1}}$$

OBDDs can contain redundance. With additional conditions we can get a more compact representation of a function. A redundance-free OBDD is called reduced OBDD (ROBDD).

Definition: An OBDD $\mathcal{B}$ is called reduced if the following holds for every two nodes v_1 and v_2 ($v_1 \neq v_2$) of $\mathcal{B}$:

$$f_{v_1} \neq f_{v_2}$$

The following reduction rules ensure that we can create a ROBDD from an OBDD

- Let v be an inner node with

  $$\text{succ}_0 = \text{succ}_1 = w$$

  than remove v and link all incomming edges to w (elimination rule).

- Let v_1 and v_2 be two (different) nodes, that
  v_1 and v_2 are terminal nodes with the same value
  or v_1 and v_2 are inner nodes on the same level with $f_{\text{succ}_b(v_1)} = f_{\text{succ}_b(v_2)}$, $b \in \{0,1\}$
  than remove v_1 or v_2 and link all incoming edges to the remaining node (isomorphism rule).

If two functions (with the same variable ordering) have isomorphic subtrees (cofactors) it is a good idea to store both functions in one BDD. BDDs that share cofactors are called SOBDDs. In the following we will call SROBDDs SOBDDs.
Algorithms on SOBDDs should not create redundancies. The elimation rule

$$f\,|_{z=0} = f\,|_{z=1}$$

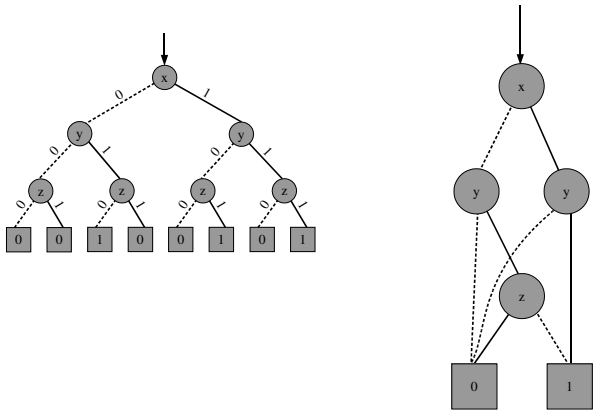


Figure 1.2: reduction on an OBDD

can be checked before creating a new node. The check for isomorphic subtrees can be done by a unique table. An algorithm should check if a node with the same successors already exists on its level. In that case a reference to this node will be used instead of creating a new (redundant) node. Otherwise a new node has to be created. We will call the function that finds and returns isomorphic subtrees or creates a new node the findOrAdd function.
To negate one function in a SOBDD an algorithm has to traverse the complete tree of that function. To negate functions in constant time complement bits were invented. This may result in a further reduction.
The JJS-BDD library can handle different types of BDDs (SOBDDs (with complement bit), ZBDDs and ADDs). This is the reason why we are talking about BDDs in the following.

1.2 The variable ordering problem

The chosen variable ordering is very important for the size of a BDD. E.g. let $\pi = (x_0, x_1, y_0, y_1)$ be the variable ordering and $f = (x_0 \wedge x_1) \vee (y_0 \wedge y_1)$. The corresponding BDD has 6 nodes. A BDD with variable ordering $\pi' = (x_0, y_0, x_1, y_1)$ representing the same function has 8 nodes (see figure 1.3).

We differ several function classes depending on their behaviour with respect to variable ordering changing (variable ordering problem). Symmetric functions have always the same size for every variable ordering. The size of 'linear' functions ranges from linear to exponential (in reference to the number of variables in the variable ordering). An example for such a function is the

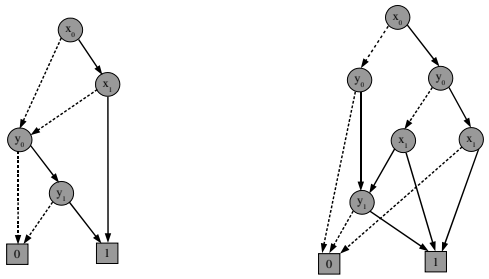


Figure 1.3: The same function with two different variable orderings

symbolic representation of the sum (see chapter 4 for more details). Another class of functions is the class where every function has expontial size for every possible variable ordering.

Chapter 2

Getting started

The source code of the library is accessible by CVS. To check out the latest version just type the following lines:

```
export CVSROOT=:pserver:anonymous@cvs.sourceforge.net:/cvsroot/jjs-bdd
  cvs login
  cvs -z4 co -P jjs-bdd
  cd jjs-bdd
  ./autogen.sh
  ./configure --prefix=/usr --enable-final
  make
  make install
```

The password phrase is empty, so just type enter. To install the library you will need to have root permissions on your system. While developing the JJS-BDD library it might be better not to use the enable-final configure option. This will speed up compilation and slow down execution. As an user of the library you should not forget using this option. It results in high optimized binaries while warnings will not be printed[1]. Because of the XML input/output format you need to have the libxml2-devel library installed. If you have any problems installing our library, just have a look in the FAQs on the website (www.jjs-bdd.de) and feel free to contact us. Once you installed the library you can check its functionality with the included testsuite by calling the jjsbdd executable. But even if the user compiled the library without installing it the testsuite can be executed. Just change to the source directory and execute the jjs-bdd testsuite:

```
[jjs-bdd@hera jjs-bdd]$ cd src
[jjs-bdd@hera src]$ ./jjsbdd
```

The first time the testsuite will be started, the prototypes of standard size will be created and stored to your home directory (∽/.jjs-bdd). This may take

[1]Though no warnings should appear anyway, because our code is ANSI C++ consistent.

some time depending on your system performance. You can also download the precomputed prototypes from the projects website. For more details about prototypes we refer to chapter 4. The testsuite will test the most important library features. All tests should be passed successfully. Please report any kind of errors to bugs@jjs-bdd.de. First the testsuite will create the switching function for the n-queens problem of the default size and print out the resulting BDD size, number of solutions and all of the satisfying assignments represented as a chessboard. Afterwards the prototypes will be created or loaded from your home directory. All prototypes will then be tested by calculating and comparing the results from container arithmetic to normal integer arithmetic for a large number of values. All input values up to a certain limit are calculated and compared. The library check will fail if any of those results differs from the expected value. Finally input/output is being tested loading one of the prototypes from harddisk and comparing the resulting container to the one computed before. If and only if all tests were passed successfully, the library will print out: "Library is working fine.". The output of the testsuite should look like this:

```
   JJS-BDD Testsuite
-----------------------
4-queens problem #### [  OK  ]
BDD nodes: 30
Solutions: 2
0 0 X 0
X 0 0 0
0 0 0 X
0 X 0 0

BDD nodes: 17
Solutions: 1
0 X 0 0
0 0 0 X
X 0 0 0
0 0 X 0

Creating prototypes (this may take some time) [DONE]
LSH ########## [  OK  ]
RSH ########## [  OK  ]
ADD ########## [  OK  ]
SUB ########## [  OK  ]
MUL ########## [  OK  ]
DIV ########## [  OK  ]
MOD ########## [  OK  ]
```

```
IO  [  OK  ]

Library is working fine.
```

Chapter 3

Library features

The JJS library offers an efficient elementary set of operations for synthesis and modification of discrete function representations.
In this chapter we want to give an overview of the most important features users might be interested in. The JJS-BDD library in version 0.1 supports several kinds of functionality on functions, containers, variable orderings, groups and the BDDs themselves. Besides this, some nice helper classes, such as the assignments, are also part of our library. We will restrict ourselves describing the SOBDD classes and common classes, which provide the basic data structures and algorithms for all kinds of BDDs. Working with ZBDDs, ADDs, etc. will be the same. For more details please take a look at the corresponding header files.

3.1 Important common classes

The common part of the library provides the shared features for all kinds of BDDs. The folder contains the following files:

- assignment.cpp
- assignment.h
- constants.h
- reordering.h
- varset.cpp
- mempool.h
- rootnode.h
- varset.h

- hashmap.h
- node.h
- ulongarray.cpp
- uniquetable.h
- input.h
- nodeinfo.h
- ulongarray.h
- varorder.h
- configreader.h
- configreader.cpp

We will give a short overview of files an user should know more about.

3.1.1 The assignment class

A very important task for BDDs is to check for satisfiability of a given function. The assignment class was designed to store such an assignment. The user can get one satisfiable assignment from a function with

```
Assignment assignment=function.satisfyOne()
```

This class provides the expected methods for accessing, finding, inserting and removing variables. The class can also be used for building cofactors.

```
function.cofactor(assignment)
```

3.1.2 The varSet class

VarSets are a simple data structure to create ordered sets of variables. They are used by BDDs for renaming variables from one set to another, for quantification and computing the relational product[1]. This class provides the expected methods for accessing, finding, inserting and removing variables. Please have a look at the header file for more details.

[1] $\exists \bar{x}[\bigwedge_{i \in I} f_i]$

3.2 Important SOBDD classes

In this section we will give a summary of the features provided on SOBDDs. Because of using templates, other kinds of BDDs are treated in the same manner - even your own (new) BDDs types would perfectly fit in here.

3.2.1 The SOBDDFunction class

An user not knowing much of BDDs, who wants to deal with switching functions will pretty fast be capable to work with our library. Functions can be created as a projection on an existing or non existing variable[2] passing the constructor a string reference. Functions can be created from assignments and they can also be copied from another function:

```
SOBDDFunction(const string&);
SOBDDFunction(const Assignment&);
SOBDDFunction(const SOBDDFunction&);
```

Some static members will be omnipresent and can easily be used for computations. These three functions are the constant one and zero function and also a function signaling an error during computations:

```
SOBDDFunction a=SOBDDFunction::ONE();
SOBDDFunction b=SOBDDFunction::ZERO();
SOBDDFunction c=SOBDDFunction::ERROR();
```

The SOBDDFunction class supports the most important logical operators, namely:

$$\text{NOT}(!),\ \text{AND}(\&, *),\ \text{OR}(|, +),\ \text{XOR}(\hat{})$$

And their assignment counterparts:

$$!=, \&=, *=, |=, +=, \hat{}=$$

Functions can be compared on (in)equality with the == and ! = operator. Besides applying the most important static operator ITE[3], cofactors and generalized cofactors (constrain, restrict) can be computed, quantification ($\forall, \exists$) can be applied and functions may be composed. Another important method is rename which provides renaming variables having a certain prefix to be renamed. Using the varSet template rename can also be called with two sets of variables. Besides this calling satisfyOne() passes one satisfiable assignment

[2] This may also be done by SOBDDFunction::projection(variablename) or SOBDDFunction::projection(variable_number) passing the variable name or number in the variable ordering

[3] SOBDDFunction::ITE(f1,f2,f3)

back to the users hand if countSatisfiableAssignments() is greater zero. Some more information like if a function is constant, the number of nodes needed to represent the current handled function, or even the variables being essential can be accessed:

```
unsigned long size=a.size();
bool constant=a.isConstant();
string essentials=a.getEssentialVariables();
```

Functions may be stored and loaded using the SOBDD methods storeFunction(filename, function) and loadFunction(filename, function_number).

3.2.2 The SOBDDFunctionContainer class

The SOBDDFunctionContainer class provides several features. It is not just a simple container class, where functions may be accessed, added and removed. It provides the same operators as functions do and some additional like left shift(<<) and right shift(>>) with its corresponding assignment operators (<<=, >>=). Computation of a relational product, where conjunction and exists quantification will be done at the same time, is also provided by this class. Users will need to know more about variable sets when using this feature. Containers may be padded, trimmed and cropped to a specified length. Also symbolic representation of integer arithmetic and logic can be done with the help of our containers and speeded up by prototype precomputation. For details on this see chapter 4 and the header file in the projects source directory. It is also possible to store and load containers or single functions from a container to harddisk. Please use the SOBDD methods storeFunctions(filename, container), loadFunction(filename, function_number) and loadFunctions(filename) to do so.

3.2.3 The SOBDDGroup class

Grouping variables together may be interesting for reordering algorithms. Grouped variables are not being seperated during reordering. That is why we wrote a common group class with the basic functionality and a SOBDDGroup. SOBDDGroups always have an unique name. Trying to create a group with a group name that is already in use, will pass the group, corresponding to the specified name, back to the user without creating a new group. Groups can be created empty or copied from another group:

```
SOBDDGroup(void);
SOBDDGroup(const string&);
SOBDDGroup(const SOBDDGroup&);
```

Adding variables to a group by groupVariable(varname) (<<varname) is just as easy as removing a variable by ungroupVariable(varname) (>>varname). Our SOBDDGroup tries to group variables if they are not belonging to another group. Removing variables works fine if the variable belongs to the current group. Unlike the group template class the SOBDDGroup swaps ungrouped variables to the nearby border of the group to add them. Calling removeVariable() will cause the variable to be removed from the group and if no function needs this variable, it will be removed from the variable ordering. Besides group joining (join(group)) are some more methods the header file does list. Please have a look on src/common/group.h and src/sobdd/sobddgroup.h for more details.

3.2.4 The SOBDDVarOrder class

The SOBDDVarOrder class is based on the common variable ordering template (src/common/varorder.h). The SOBDDVarOrder offers methods to iterate on non empty groups. It also contains methods for adding, inserting and removing variables, methods for removing unused variables and empty groups (clear(), clearGroups()) etc.

3.2.5 The SOBDD class

The SOBDD class provides the global functionality on the BDD structure. Variables may be added, inserted and removed. Functions may be stored and loaded and reordering algorithms may be called. Setting the garbage collection delay and even starting garbage collection by hand are also supported features.

Chapter 4

The Function Container Class

The SOBDDFunctionContainer class was designed for several purposes. On the one hand it implements a standard container class for SOBDDFunctions, where functions can be accessed, added and removed by the user. As we show in the next section, containers can on the other hand be used for (symbolic representation of) arithmetic and logic on unsigned integers. In addition each container has an extra function representing the status of the corresponding container instance.

4.1 Integer value container

Besides further standard operations on containers like the conjunction, disjunction, etc. of all contained functions, the class can be used as a data structure for arithmetic and logical operations. Our containers are able to represent a simple unsigned integer value. Let b be the binary representation of an integer value n, b_i the i-th bit of b and x a function container. Then the i-th function of the container is defined by

$$x[i] := \begin{cases} \text{SOBDDFunction::ONE()} & \text{if } b_i = 1 \\ \text{SOBDDFunction::ZERO()} & \text{if } b_i = 0 \end{cases}$$

Creating a new container representing an integer value by

SOBDDFunctionContainer x=SOBDDFunctionContainer(12);

or simply

SOBDDFunctionContainer x=12;

the user gets a new function container of size $\lceil \log_2(x) \rceil$. On the other hand, if the container functions are constant, the container might be interpreted as an integer, which can be calculated by the getNumber() method. The result of getNumber() will be the number represented by the container. Every non constant function will be interpreted as one.

4.2 Arithmetic and logic on containers

This previously mentioned interpretation of a container enables us to apply standard arithmetic operators like $+, -, \cdot, /$ to containers. But our containers are capable to deal with any kind of functions, which leads to symbolic representations of standard integer arithmetic. For example, adding $(+)$ two containers, which just contain projection functions results in a new container - a symbolic representation of the sum of two binary numbers of length n. We call this a prototype for the sum[1] and denote it with ADD_n. Let x and y be two containers of size n containing the projection functions on $(x_{n-1}, x_{n-2}, \ldots, x_0)$ and $(y_{n-1}, y_{n-2}, \ldots, y_0)$ respectively. The resulting container $s = x + y$ is also of size n and the n-th carry bit (function) is stored to the containers status. Calling disableCutOff()[2] before applying the "+" operator causes the resulting container to be of size $n + 1$ if $c[n]$ is not equal to zero. Let

$$c[i] := \begin{cases} (x[i] \cdot y[i]) + c[i-1] \cdot (x[i] + y[i]) & i \in \{1, 2, \ldots, n\} \\ 0 & \text{else} \end{cases}$$

the i-th carry bit. Then holds:

$$s[i] = x[i] \hat{} y[i] \hat{} c[i],$$

where "$+, \cdot, \hat{}$" are the logical OR, AND and XOR.

4.2.1 Prototype containers

Our containers can be stored and loaded in two different formats - XML and binary (see chapter 7 for more details):

```
Storing a container:
SOBDD::storeFunctions("file.xml",container,XML_FORMAT);
SOBDD::storeFunctions("file.jjs",container,BINARY_FORMAT);

Loading a container:
SOBDD::loadFunctionsXML("file.xml");
SOBDD::loadFunctions("file.jjs");
```

When using an arithmetic operator on containers, the corresponding prototype has to be created. Instead of creating those prototypes every time an operator is applied, we support using precomputed containers. Those containers will be generated up to a specified size n and stored in binary format in the users home directory ($\backsim$/.jjs-bdd) when calling:

[1] All prototypes: LSH, RSH, ADD, SUB, MUL, DIV and MOD
[2] SOBDDFunctionContainer::disableCutOff()

```
SOBDDFunctionContainer::createPrototypes(n);
```

This causes all containers (ADD_2,ADD_3,...,ADD_n), (SUB_2,SUB_3,...,SUB_n),..., (MOD_2,MOD_3,...,MOD_n) to be precomputed and stored. Because creating those prototypes takes some time - especially with increasing n, we put those containers up to a size of 16 on the projects website (www.jjs-bdd.de) for download. The creation of the left and right shift prototypes (LSH_n, RSH_n) is fast enough so they will not be stored. Once a user has created (or downloaded) the prototypes, createPrototypes(n) will preload those containers for later usage. This results in an enormous speedup of symbolic computations.

4.2.2 Logical operators

Additionally logic operators like AND, OR and XOR but also comparison operators are supported features. There are several ways comparisons may be intended to be interpreted. If the user wants to know if two containers x and y have exactly the same content, using $x.eq(y)$, which results in a boolean value will be the right choice. Applying one of the defined operators to containers in general results in a symbolic representation. Let 0 denote the constant zero function and 1 denote the constant one function in the following examples. The first example shows a simple comparison resulting in a symbolic representation: Let $x = (1, x[2], 0, x[0])$ and $y = (1, y[2], 0, y[0])$.

$$(x = y) = (x[2] \leftrightarrow y[2]) \wedge (x[1] \leftrightarrow y[1])$$

When comparing two integers with the same values but a different number of bits for their representation (leading zeros), this causes the result to be the constant zero function. To prevent this the container class offers a method to switch to numeric comparisions, where leading zeros are going to be ignored. Switching can be done by calling the static SOBDDFunctionContainer::enableNumericCheck() method of the container class. Comparing $x = (0, 0, 0, 1, 0, 1)$ and $y = (1, 0, 1)$ leads to:

$$(x = y) = \begin{cases} 1 & \text{if enableNumericCheck() was called before} \\ 0 & \text{else} \end{cases}$$

Calling SOBDDFunctionContainer::disableNumericCheck() will switch back to default.

Chapter 5

Dynamic reordering

One of the most important things to know when dealing with BDDs is the variable ordering problem. As it was shown in chapter 1 the resulting BDD size measured by the number of nodes can vary from linear to exponential in the input size n depending on the variable ordering. This affects both, memory and computation time and results in functions, which are not presentable because of a bad choice for the ordering. The first approach to get this under control is to check for a good ordering before synthesising the function. That is what we did in the XML file input case, where we used the Fanin or Weigths heuristic to measure the importance of each variable. Symmetric variables can be swapped without changing the BDD size. In version 0.1 of the library we do not test for symmetric variables and do not group them together. See chapter 10 for more details. Another important approach is to reorder an existing BDD to minimize its size. This can be done on any kind of BDD providing a swap function, which swaps two adjacent variable layers without changing the represented switching function(s).

Algorithm swap

Input: BDD with variable ordering $\pi = (x_1, , \ldots, x_i, x_{i+1}, \ldots, x_n)$ and index i
Output: BDD with variable ordering $\pi = (x_1, \ldots, x_{i+1}, x_i, \ldots, x_n)$

WHILE $\exists$ x_i node n with level$(n) = i$ without successors on level $i+1$ **DO**
  level$(n) := i + 1$
OD

WHILE $\exists$ x_i node n with level$(n) = i$ **DO**
  $w_{00} := x|_{x_i=0,\ x_{i+1}=0}$
  $w_{01} := x|_{x_i=0,\ x_{i+1}=1}$
  $w_{10} := x|_{x_i=1,\ x_{i+1}=0}$
  $w_{11} := x|_{x_i=1,\ x_{i+1}=1}$

  IF$(w_{00} = w_{10})$ **THEN** $t_0 := w_{00}$
  ELSE t_0 :=findOrAdd$(i+1, w_{10}, w_{00})$ **FI**

  IF$(w_{01} = w_{11})$ **THEN** $t_1 := w_{11}$
  ELSE t_1 :=findOrAdd$(i+1, w_{11}, w_{01})$ **FI**

  $n|_{x_{i+1}=0} := t_0$
  $n|_{x_{i+1}=1} := t_1$
  level$(n) := i$
OD

WHILE $\exists$ x_{i+1} node n with level$(n) = i+1$ without predecessors on level i **DO**
  level$(n) := i$
OD

Using a swap operator on BDDs we could create any desired variable ordering. But since there are $n!$ permutations of a variable ordering $\pi = (z_1, z_2, \ldots, z_n)$ we will not be able to build all of those $n!$ BDDs and find the smallest one within an acceptable time. That is why a lot of reordering methods do exist. In version 0.1 of our library we do support the following list of reordering algorithms:

- Window permutation with window sizes $2, 3$ and 4
- Sifting
- Genetic minimize

The way reordering was implemented makes extending the library very easy. We decided to implement reordering as a template. The effect is, that any

kind of BDD added by a developer implementing a swap function provides the mentioned reordering algorithms automatically. On the other hand when adding a new reordering technique all other kinds of BDDs having a swap function benefit from this without any extra work. A good example is the ADD:

```
new Reordering<ADDVarOrder,ADDGroup,ADD>(&ADD::swap,this);
```

Before giving a short overview on the mentioned reordering algorithms in the next section we will show how groups of variables are used and handled by our reordering algorithms.

5.1 Grouping variables

All reordering algorithms can deal with variable groups. Variable groups are sets of adjacent variables in the variable ordering which belong together. The user may define groups of variables which should not be seperated from each other by a swap, shift or any other reordering algorithm. Groups within the variable ordering are signified by brackets $[\ldots]$. For example

$$\pi_1 = (x, [y_1, y_2], z_1, z_2, [y_3, y_4])$$

contains two groups - each of them contains two y variables. Before executing any reordering algorithm ungrouped variables are temporarily grouped together. In our example the variable ordering would look like this:

$$\pi_1' = ([x], [y_1, y_2], [z_1, z_2], [y_3, y_4])$$

After that any reordering technique described in the next section will be applied. Variables within each group and groups may be shifted to other positions in the variable ordering, but variables belonging together are never seperated. In our example the new variable order might look like this:

$$\pi_2' = ([z_1, z_2], [y_2, y_1], [y_4, y_3], [x])$$

Afterwards all temporary groups will be removed and the resulting variable ordering will look like this:

$$\pi_2 = (z_1, z_2, [y_2, y_1], [y_4, y_3], x)$$

5.2 Reordering algorithms

We implemented some reordering algorithms, which will be explained in the next sections. All kinds of BDDs providing a swap function automatically inherit those algorithms for free.

5.2.1 Window Permutation $2, 3$ and 4

When applying window permutation of window size $k \in \{2, 3, 4\}$ to a BDD the first k variables are going to be examined and all $k!$ permutations are build with $(k! - 1)$ swaps. Afterwards the best ordering of those three variables is known and created before continuing. The window moves forward to examine the second to the $(k + 1)$-th variable and builds all BDDs corresponding to the next $k!$ permutations of the variable ordering. The window permutation terminates after processing the last k variables. Because of a possibly existing group structure variables which are not in the same group as the first one of the actual window are going to be ignored by window permutation k. We decided to implement window permutation up to a size of 4, because when using a larger window size the number of permutations and swaps increases dramatically and thus computation time does too. For all three implemented sizes, we do use the minimal number of swaps to create all permutations and a minimum number of swaps to create the best ordering found.

5.2.2 Sifting

Sifting consists of two parts which are quite the same. The only diffence is, that the first is working on variables and the second is working on groups. Starting with the variables, all variables in the current group are taken successively, swapped to all possible positions in the variable ordering. The best position is kept in mind and restored after all positions have been tested. Sifting takes an double value as parameter, which we call the max growth value. The default value of this parameter can be found in constants.h. See chapter 8 for more details. When reaching a BDD size, which is max growth times larger then the best known BDD size, sifting stops swapping. This is to speed up sifting, but it is possible that sifting does not find an existing better variable ordering because of a local minimum in the size of the current BDD.

Sifting on Variables

Sifting variables will be the first step in sifting, but can also be called by hand:

```
SOBDD:variableSifting(max_growth)
```

Sifting on Groups

Sifting on groups will be executed as a second step in sifting and can be called by hand just as the variable sifting described before:

```
SOBDD:groupSifting(max_growth)
```

5.2.3 Genetic minimize

We implemented a hybrid genetic algorithm that treats different variable orderings as individuals with a fitness value reciprocal to number of nodes in the corresponding BDD in a population of fixed size. Parents are picked by roulette wheel selection with probabilities according to the individual's fitnesses. For obtaining the next generation of individual variable orders we can choose among several mating operators[Gol89]. Additionally mutation is applied to resulting offspring by a chance of 0.05. Mutation swaps a small number of random variables. After mutation we conduct a "minimum sifting", i.e. a fast sifting algorithm with a maximum growth factor of 1.0 as a problem specific heuristic (Hybridization). Continuing this way, we create new generations until no further improvement in terms of fitness was obtained for a given number of iterations.

Genetic minimize on Variables

Mating operates on variable level, therefore the searchspace is the entire space of variable permutations.

Genetic minimize on Groups

Mating operates on variable groups, thus restricting the searchspace to permutations of variable groups.

Chapter 6

Memory

Memory management is important while working with BDDs. Without any memory technique manipulating would be very slow. We want to give a short overview of the techniques we use and why they are so efficient.

6.1 Computed tables

Computed tables are used to store results. Before an algorithm traverses a subtree of a BDD a look up in the computed table will be done. If there is an entry no more calculation is needed for this subtree. Without computed tables every node would be visited on every possible path.
Computed tables speed up algorithms. A very difficult question is how much memory should a computed table use. This depends on the given situation. We implemented computed tables as hashmaps to have a fast look up and control of the reserved memory.

6.1.1 Hashmap

The user can vary the parameters p, q and c of the hashmap:

- the number of collision lists $(p \cdot q)$
- the size of the collision lists (c)
- the hashfunction $f(x) := q \cdot (x \bmod p) + (x \bmod q)$

See chapter 8 for more details.

6.1.2 Computed table container

We designed the library with the view to simplify implementing new types of computed tables (e.g. with more parameters). A developer implements a

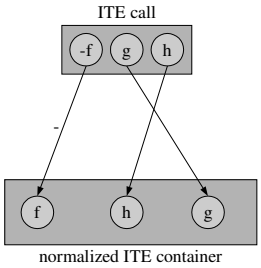


Figure 6.1: Normalized entry for the computed table

container that stores the parameters of an operator (e.g. ITE stores all three functions). The container can manipulate the behaviour of the hashmap. The container has to implement a function to get the key for this container. The hashmap will use this key to calculate the collision list in which the container will be stored. The hashmap also needs an operator to decide whether an 'equal' entry exists.
This is all a developer needs to implement for a new computed table. This is a powerful and easy concept. The ITE algorithm is a very good example for this. We are handling standard triples directly in the container. To increase the number of hits in the computed table we normalize ITE calls (ITE$(-f, g, h)$ =ITE(f, h, g)). The normalization is done by our ITE container (see figure 6.1).

6.2 Unique tables

The computation time for manipulating BDDs depends on the size of the used BDDs. So it is wanted to work with reduced BDDs. The elimination rule

$$f\mid_{z=0} = f\mid_{z=1}$$

can be checked by every algorithm directly. The more complicated part is to check for isomorphic subtrees. For this reason we included unique tables. Before creating a new node we look up in the unique table if such a node already exists. In this case no new node will be created. We have a unique

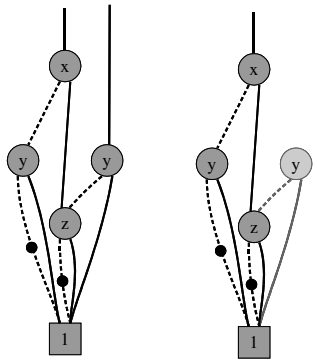
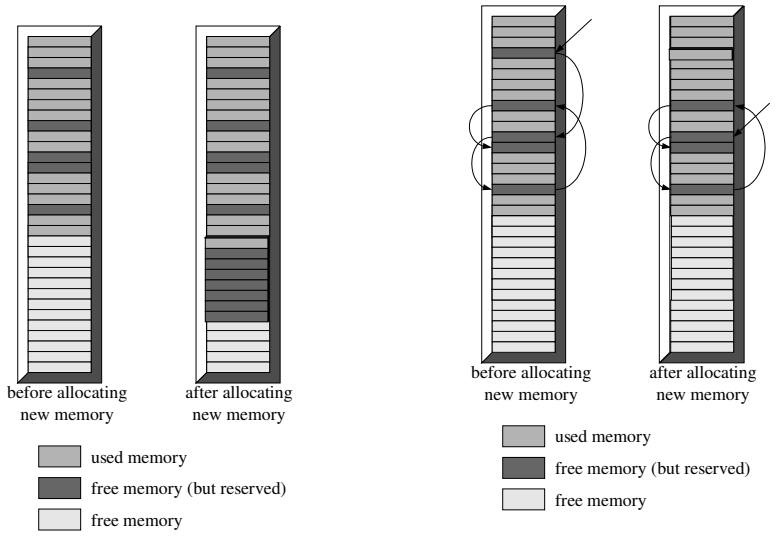


Figure 6.2: Before and after removing a BDD function

table for every variable to speed up the search for nodes.
Our implementation of unique tables is designed to handle complement bits. There are two tables that are combined to one. The tables store a disjunctive set of nodes. One table is used for nodes where no complement bit is set and the other for nodes where the complement bit is set. Every table is represented by a map that can store maps itself. If a node with succ0 and succ1 (the complement bit will be ignored because we already work on the corresponding table) will be searched the result from this table will be taken to search for the next successor. The result of this search is the corresponding node if such node exists.

6.3 Garbage collection

Whenever a function is deleted nodes can become unnecessary. In most cases it is a good idea not to erase the nodes immediatly. Whenever a node becomes unnecessary it will be left unreferenced in the unique table. Those nodes are called 'zombie' nodes. We are using a garbage collection delay. If the number of zombie nodes is above that delay we erase all unnecessary nodes to free memory. If an algorithm is requesting a zombie node, the node and its corresponding subtree will be 'reincarnated'.
The main advantage of the garbage collection is based on the computed tables. With garbage collection we do not have to clear the computed tables so often. This increases the number of hits and that results in a reduction of computation



without memory pool

with memory pool (and its linked list)

Figure 6.3: Memory fragmentation

time.
We implemented this with a reference counter for every node. If no node is pointing to a node (reference count=0) it becomes a zombie and will be inserted in a set. If a node (or subtree) is requested to be 'reincarnated' it is removed from the set and is no longer a zombie. The garbage collection (deleting all zombie nodes) is done by removing all nodes in the set instead of traversing all unnecessary subtrees.

6.3.1 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. We are just using a memory pool for nodes. We are using the reference to one of the successors to link a node (a deleted (not zombie) node does not need information about successors anymore). So no additional memory is used to prevent memory fragmentation.

Chapter 7

File format/IO

We are providing two ways to store and load BDDs. One XML based and one binary input/output format.

7.1 XML format

The XML format is for synthesising small functions by hand. It is not recommended to use this format for large functions or function containers. The file size is very large and loading is extremely slow. We are using the libxml2 to read a XML file. We can only load circuits without cycles. Before building the function (or container) we have to check for cycles. The XML format is variable ordering independent. This is a big advantage because an user can have only one BDD at the same time. An user does not have to reorder the BDD to load a new function. This is done by high level functions. This advantage is paid with a high computation time.

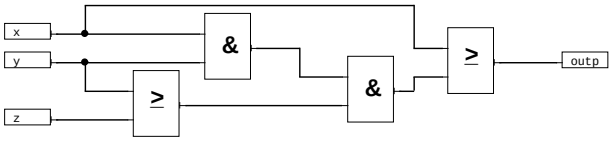
7.2 Binary format

The binary format was developed to load BDDs fast and to store them in a compact way. There are no checks (except for the header) if a file is correct. A disadvantage is that situations where a correct file can not be loaded (variable orderings can't be joined) can occure. The BDD is build with low level functions. For every node just one findOrAdd(...) will be called. This leads to a compact and fast readable file format. The file format consists of three parts:

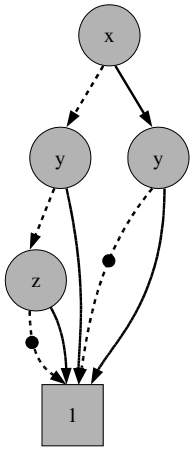
- header
- body
- outputs

```
<?xml version="1.1" ?>
<!DOCTYPE circuit>

<circuit>
  <and id="and1">
    <input>x</input>
    <input>y</input>
  </and>

  <or id="or1">
    <input>y</input>
    <input>z</input>
  </or>

  <and id="and2">
    <input>and1</input>
    <input>or1</input>
  </and>

  <or id="or2">
    <input>x</input>
    <input>and2</input>
  </or>

  <output id="output">
    <input>or2</input>
  </output>
</circuit>
```

XML file



circuit



BDD

Figure 7.1: XML file and corresponding circuit and BDD

7.2.1 Header

The header contains information that is needed before creating nodes. This information is file type, the variable ordering and the number of nodes. The first line of the header contains

- file type (8 bytes)
- version number (5 bytes)
- BDD type (5 bytes)

```
#JJS-BDD 1.1SOBDD
```

The following lines represent the variable ordering (in the 'right' order). Every line is terminated by '$\backslash n$'. One empty line signals the end of the variable ordering.

```
#JJS-BDD 1.1SOBDD
x
y
z

```

Nodes do not need a separator because the size for a node can be calculated during loading or storing. A first idea could be to store the number of nodes (n), calculate the number of bits needed to store the successors ($\lceil log(n) \rceil$) and then store every node with this **fixed** size. But in our special case this would not be very compact. A node just can have succesors on levels below its own. So we need the information how many nodes are on levels below the current. We do this by storing the number of nodes on every level. But instead of storing this information in plain text we want to store it more compact. We calculate the number of bits needed to store the largest number and store this information in plain text.

```
#JJS-BDD 1.1SOBDD
x
y
z

2

```

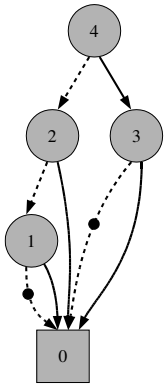


Figure 7.2: BDD with numbering

After that we store the number of nodes on each level (in the 'right' order) with the calculated length.

```
011001 → 01 10 01 → 1 2 1
1 x node
2 y nodes
1 z node
```

7.2.2 Body

Let n be the number of nodes and V be the set of nodes. Every node gets a unique number ($\forall v \in V$ number$(v) \in [0, n-1]$). Additionally for two nodes $v_0, v_1 \in V$ the following must hold (see figure 7.2):

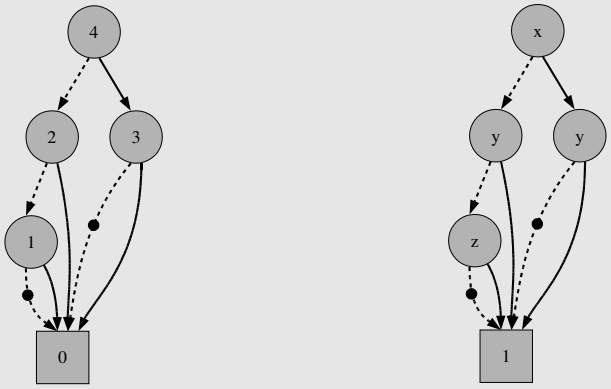
$$\text{number}(v_0) < \text{number}(v_1) \Rightarrow \text{level}(v_0) \geq \text{level}(v_1)$$

The one drain has always number 0 and does not need to be stored. For all nodes $(1, \ldots, n-1)$ we just need to store the numbers of the two successors and the complement bit.
Let k be the number of levels of the BDD and c_i the number of nodes on this level ($i \in [1, k]$). For all nodes v (except the one drain) we need

$$2 \cdot \left\lceil \log \left(\sum_{i=\text{level}(v)+1}^{k} c_i \right) \right\rceil + 1$$

bits to store the two successors and the complement bit.

```
00110000110110 → 001 100 001 10110 → (0,0,1) (1,0,0)
(0,0,1) (2,3,0)
```



Better compression

But in some cases we are still wasting memory. Let v be a node with successors s_0 and s_1 and complement bit c. With

$$b = \left\lceil \log \left(\sum_{i=\text{level}(v)+1}^{k} c_i \right) \right\rceil$$

bits we can store numbers from 0 to $2^b - 1$. If

$$l = \left(\sum_{i=\text{level}(v)+1}^{k} c_i \right) < 2^b - 1$$

we can use $(2^b - l - 1)$ remaining numbers to store more information.
If $(\text{number}(s_0) < 2^b - l - 1)$ or $(\text{number}(s_1) < 2^b - l - 1)$ we do not have to store the bit for the negation (complement bit). Without loss of generality let $\text{number}(s_0) < 2^b - l - 1$. Now we can store the information of the complement bit with

$$\text{number}(v) := \begin{cases} \text{number}(s_0) & c = 0 \\ 2^b - \text{number}(s_0) - 1 & c = 1 \end{cases}$$

and so we do not have to waste the bit for the negation.

```
11 10000110110 → 11 100 001 10110 →
(0,0,1) (1,0,0) (0,0,1) (2,3,0)
```

There are functions where such a situation occurs in almost every node. The n-queens problem is a good example for that.

7.3 Outputs

Now we could build the complete BDD. But we do not have the information which nodes are used as an output and if these outputs are negated or not. The next line stores the number of outputs followed by the output nodes (number and complement bit)

```
1 (number of outputs)
1001 ⇝ (4,1)
```

7.4 Example file

Here is the complete example file which we used to show how the file format works.

```
00100011 01001010 01001010 01010011 00101101 01000010    #JJS-B
01000100 01000100 00100000 00100000 00110001 00101110    DD  1.
00110001 01010011 01001111 01000010 01000100 01000100    1SOBDD
00001010 01111000 00001010 01111001 00001010 01111010     x y z
00001010 00001010 00110010 00001010 01100111 10000110      2 g
11000000 00110001 00001010 10010000                       A1
```

Chapter 8

JJS-BDD constants

The JJS-BDD package provides the user several ways to have a share in the performance of the library. Most of the options are bunched in one file and can thus easily be changed. After changing one or more of those options the package needs be recompiled and installed. Instead of changing the options in the file, every user can set the values in a config file (∽/.jjs-bdd/config). This is the recommended way to change the options, because the user does not have to recompile the library. In this chapter we will give a short overview of these options stored in the constants.h file, which can be found in the src/common directory. Besides some internal definitions like bad characters which are not allowed for group names, this file also contains some constants developers should know about.

8.1 IO constants

The file input and output part of our library defines two kinds of constants. The first determines the different file formats and their default, while the second defines the heuristic to be used for a good variable ordering before reading an XML input file.

8.1.1 IO format

Our library supports two file formats. Because of the better performance the binary format is chosen to be the default. Developers should add their own file format to this list:

```
const short XML_FORMAT= 0;
const short BINARY_FORMAT= 1;
const short OUTPUT_DEFAULT=BINARY_FORMAT;
```

8.1.2 Input heuristics

As we saw in chapter 7 the XML file format is independent from the existing variable ordering because of using a high level synthesis function to generate the BDD nodes. As shown before in chapter 1, the variable ordering of the BDD is of great importance for the BDD's size. Besides the reordering techniques described in chapter 5 it might be interesting to have an advantageous ordering before function synthesis. This is done by several heuristics. Developers should append constants for their own heuristic to this list:

```
const short NONE_HEURISTIC= 0;
const short FANIN_HEURISTIC= 1;
const short WEIGHT_HEURISTIC= 2;
```

8.2 Reordering constants

As one can see in chapter 5 our package provides a large range of reordering algorithms. Their default options are also defined within the constants.h file.

```
const float MAX_GROWTH_DEFAULT=
   ConfigReader::getFloat("max growth",1.3f);
const unsigned long POPULATION_SIZE=5;
const unsigned long ELITARISM_WINNERS=
   (POPULATION_SIZE/5 < 2?2:POPULATION_SIZE/5);
const unsigned long RANDOM_REPLACEMENTS=POPULATION_SIZE/5;
```

- The MAX_GROWTH_DEFAULT constant is used as a default by the sifting algorithm on groups and variables. Whenever the size of the variable ordering gets larger than 1.3 times the size of the best tested ordering, sifting will stop swapping the current group or variable in the current direction.
- The POPULATION_SIZE constant defines the size of the population, i.e. the number of individual variable orderings per generation, for the genetic reordering algorithm. For larger variable orderings choose a smaller value for performance reasons, for smaller orderings choose a larger population size to maintain diversity.
- The ELITARISM_WINNERS constant for the genetic reordering algorithm defines the number of high performance individuals that enter the next generation unaltered plus an additional mutation of the best individual. E.g. a value of 2 implies that the best individual gets copied to the next generation and a mutation of the same one does so as well.

- The RANDOM_REPLACEMENTS constant for the genetic reordering algorithm defines the number of individuals for the next generation that get replaced by random variable orderings to prevent premature convergence ("incest").

8.3 Memory constants

Constants concerning memory are also defined here. Besides the size of computed tables the memory management options can be found in this file.

8.3.1 Memory management

As we saw in chapter 6 a two level memory management is used by our library. The MEM_POOL_SIZE_DEFAULT constant determines how many nodes should fit to the allocated mem pool whenever new memory is needed. While the
GARBAGE_COLLECTION_DELAY_DEFAULT value decides how many zombie nodes are kept in memory before garbage collecting them to free memory:

```
const unsigned long GARBAGE_COLLECTION_DELAY_DEFAULT=
   ConfigReader::getULong("garbage collection delay",100000);
const unsigned int MEM_POOL_SIZE_DEFAULT= 100000;
```

8.3.2 Computed table constants

It was also shown in chapter 6 that computed tables can speed up all kinds of algorithms on BDDs. All computed table parameters can be changed separately. We just show the section for the ITE computed table of constants.h. All other options are defined according to expectations:

```
const unsigned long ITE_P_DEFAULT=
   ConfigReader::getULong("ite p",7);
const unsigned long ITE_Q_DEFAULT=
   ConfigReader::getULong("ite q",9);
const unsigned short ITE_COLLISION_LIST_SIZE_DEFAULT=
   ConfigReader::getULong("ite collision list size",5);
```

8.4 Prototype constants

The default size (number of bits) of the prototypes ADD, SUB, MUL, DIV and MOD can also be found here. See chapter 4 for more details.

```
const unsigned long PROTOTYPE_SIZE_DEFAULT=
  ConfigReader::getULong(”prototype size”,8);
```

Chapter 9

Implementing new BDD types

One of the main goals while developing the library was to make implementing new BDD types easy. We implemented many classes as templates to reuse datastructures and algorithms. Let's see how to implement a new BDD type. We will show this on ADDs.

We have a node class from which we can derive and implement an inner node (ADDNode) and a terminal node (ADDTerminalNode) class. This is important because we do not want to store values in inner nodes and we do not want to store successors in terminal nodes. With this construct we do not have to waste memory for drains in inner nodes. Every node type has to implement the node interface. After implementing the nodes we can use the unique table, memory pool, variable ordering, root nodes, ... without modifying anything. Now we can start writing the findOrAdd($\cdot$) function.

Algorithm findOrAdd(u, s_1, s_0)
Input: Nodes s_1 and s_0 and unique table u
Output: Node v with successors s_0 and s_1 with unique table u
IF $\exists v | u(s_1, s_0)$ **THEN** return v
ELSE
  v =memPool→alloc()
  v →setUniqueTable(u)
  v →setSucc(s_0, s_1)
  return v
FI

With these classes and functions we can create an ADDFunction class and implement the first synthesis functions.

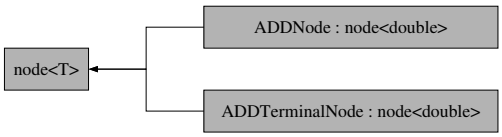


Figure 9.1: class structure

Algorithm addition(u, v)
Input: Nodes u and v
Output: Node w with $f_w = f_u + f_v$

case 1a: $(0 + f)$
IF $(u \rightarrow$isDrain()$) \wedge (u \rightarrow$getValue()$= 0)$ **THEN** return v **FI**

case 1b: $(f + 0)$
IF $(v \rightarrow$isDrain()$) \wedge (v \rightarrow$getValue()$= 0)$ **THEN** return u **FI**

case 2: $(*, *)$
IF$(u \rightarrow$isDrain()$) \wedge (v \rightarrow$isDrain()$)$ **THEN**
  return findOrAddDrain($u \rightarrow$getValue()$+v \rightarrow$getValue())
FI

$x := \min\{\text{var}(u), \text{var}(v)\}$

$w_0 :=$addition$(u\,|_{x=0}, v\,|_{x=0})$
$w_1 :=$addition$(u\,|_{x=1}, v\,|_{x=1})$

IF $w_0 = w_1$ **THEN**
  return w_0
ELSE
  return findOrAdd()
FI

A wide range of reordering algorithms can immediatly be used after implementing the swap function.

```
void ADD::sifting(const float maxGrowth){
  ADD::variableSifting(maxGrowth);
  ADD::groupSifting(maxGrowth);
}

void ADD::variableSifting(const float maxGrowth){
  reordering→variableSifting(maxGrowth);
}

void ADD::groupSifting(const float maxGrowth){
  reordering→groupSifting(maxGrowth);
}

void ADD::windowPermutation2(void){
  reordering→windowPermutation2();
}

void ADD::windowPermutation3(void){
  reordering→windowPermutation3();
}

void ADD::windowPermutation4(void){
  reordering→windowPermutation4();
}

void ADD::geneticMinimize(void){
  reordering→geneticMinimize();
}
```

All the developer has to do is to create a new reordering object.

```
new Reordering<ADDVarOrder,ADDGroup,ADD>(&ADD::swap,this);
```

This should give a short impression how to implement new BDD types.

Chapter 10

Perspective

After discussing the features we have already completed we want to give a short overview of some features we still want to implement:

- linear sifting, another reordering algorithm on SOBDDs
- symmetric groups to find coherences between variables
- input negation to reduce the BDD size
- ISOP algorithm for implicit cube set representation
- dynamic garbage collection for a more efficient memory usage
- graphical user interface (GUI) for debugging and teaching purposes

The JJS-BDD library provided a GUI in the release 0.01. After that release we changed the whole structure of the library. The old GUI code was written especially for one BDD type (SOBDD). With the new BDD types it is more important to write a GUI that can be used for every type. That is why we removed the GUI from this release. The new GUI core code will be written as a template and be the main focus of our next release.
More information about the progress and further code examples can be found on our webpage (www.jjs-bdd.de).

Bibliography

[Bry86] R. Bryant. Graph-based algorithms for boolean function manipulation, pages pp 677–691. IEEE Transaction on Computers. 1986.

[CM98] T. Theobald C. Meinel. Algorithmen und Datenstrukturen im VLSI Design. Springer Verlag, 1998.

[EC00] D. Peled E. Clarke, O. Grumberg. Model Checking. MIT Press, 2000.

[Gol89] D.E. Goldberg. Genetic algorithms in search, optimization, and machine learning. 1989.

[McM93] K. McMillan. Symbolic Model Checking. Kluwer Academic Press, 1993.

[Min96] S. Minato. Binary Decision Diagrams and Application for VLSI Design. Kluwer Academic Press, 1996.

Index