An Efficient ROBDD Package

Jesper Møller
c937140

Christian Østergaard
c929676

Department of Information Technology
Technical University of Denmark
June 1996

Abstract

Many interesting problems in computer science can be modelled using Boolean expressions. In formal verification for example, a common task is to determine whether a given model of a system satisfies some properties supposed to hold for the system. Such problems are often modelled as transition systems over Boolean variables, and thus a computer-aided solution requires some efficient techniques for manipulating Boolean expressions.

Reduced Ordered Binary Decision Diagrams (ROBDDs) provide an efficient way of representing Boolean expressions and many Boolean functions can be implemented as graph algorithms on ROBDDs. This report shows how an ROBDD package can be designed and efficiently implemented in C/C++.

Preface

This report and the enclosed program is the result of a bachelor project carried out at the Department of Information Technology, Technical University of Denmark. The purpose of the report is to describe how to design and implement an efficient ROBDD package. The report also shows how to use the implemented ROBDD package to solve problems which can be modelled using Boolean variables.

Thus, the report can either serve as a specification of how to implement an efficient ROBDD package, or as a reference manual to the package we have developed. It is assumed that the reader has a basic knowledge of C/C++ programming, and is acquainted with Boolean algebra.

The report consists of six chapters and three appendices. In chapter 1 we give a short introduction to ROBDDs and we state the requirements specification to the ROBDD package. Chapter 2 is concerned with different design issues, and chapter 3 investigates our implementation. In chapter 5, Milner's Scheduler is used to examine the efficiency of the ROBDD package. Finally, chapter 6 is a conclusion where we summarize the important results. Appendix A shows how to install the ROBDD package under Unix or Linux.

Jesper Møller Christian Østergaard

Contents

Chapter 1

Introduction

In many areas of computer science Boolean algebra can be used to model a given system. When a model has been developed, it is often desirable to use computers to facilitate the many and usually tedious calculations. Therefore, it is important to be able to represent and manipulate Boolean expressions efficiently.

In section 1.1 we give a short and informal introduction to the ROBDD data structure which provide an efficient representation of Boolean expressions. A more detailed description can be found in [1] or [3]. In section 1.2 we discuss the problems of designing and implementing an efficient ROBDD package. Finally, in section 1.3 we state the requirements to the ROBDD package we will implement.

1.1 Reduced Ordered Binary Decision Diagrams

An obvious way of representing a Boolean expression would be a binary tree, but this representation requires $\Theta(2^n)$ nodes, where n is the number of variables in the expression. Even for relatively small n this representation is infeasible. By observing that only two external nodes are necessary and exploiting the posibility of sharing subtrees the size can be dramatically reduced. Such a representation is known as a *Binary Decision Diagram* (BDD). A BDD is a rooted, directed, acyclic graph (DAG) where each node represents a variable *var* and has two outgoing edges, the *high* and *low* edge.

If the variables are arranged with respect to a given ordering an *ordered* BDD (OBDD) is obtained. In an OBDD all paths from the root to a terminal node will visit the variables in this order. By introducing an ordering, efficient algorithms for manipulating OBDDs can be developed. The ordering has a great impact on the size of an OBDD and should therefore be chosen carefully. However, for some expressions the size of the OBDD becomes exponential for all variable orderings. Fortunately, an acceptable ordering can be established relatively easy for most OBDDs.

A *Reduced Ordered Binary Decision Diagram* (ROBDD) is an OBDD where redundant nodes and isomorphic sub-DAGs have been removed. A node is redundant if and only if it has identical *high* and *low* edges. Two sub-DAGs are isomorphic, if and only if they have

the same truth value for all variable assignments. Thus, in an ROBDD Boolean expressions have a *canonical* representation.

This property makes it possible in constant time to check whether two ROBDDs are equivalent. Moreover, the question of tautology and satisfiability can be verified in time $O(1)$. This is a co-NP-complete problem for other representations, such as binary trees and Karnaugh maps. In section 2.3 we show how to manipulate Boolean expressions using graph algorithms on ROBDDs.

1.2 Problem Statement

In designing an efficient and usable ROBDD package it is necessary first to consider an internal representation of ROBDD nodes. Next, an operator for constructing ROBDD nodes should be developed. This constructor must take care of preserving the canonical form in the internal representation. Then, a set of manipulation operators should be provided allowing composition of ROBDDs. These operators should use dynamic programming to improve running time.

In order to be efficient regarding storage requirements, some memory management stategies must be considered. Among other things, these strategies should take recycling of unused ROBDD nodes into consideration. Finally, an interface to the ROBDD package must be developed.

1.3 Requirements Specification

Our primary goal is to develop an efficient and usable ROBDD package which can be included in other programming projects.

The package will provide an abstract data type which can be used to manipulate Boolean expressions. This data type will be implemented as a C++ class providing the basic Boolean operators ($\neg, \wedge, \vee$, etc.) and the universal and existential quantifiers ($\forall$ and $\exists$).

The kernel of the package will be based on a few core operators (CREATE-NODE, ITE and RESTRICT) implemented in C. These operators provide the neccesary means by which the Boolean operators and quantifiers can be implemented. We will also provide a function to fix the variable ordering, as well as the auxiliary functions NODESIZE and ANYSAT.

We will use dynamic programming and randomization to improve the running time of the core operators. Furthermore, a garbage collection technique will be developed in order to automatically recycle redundant nodes.

We will test the package on a problem known as Milner's Scheduler. This problem is scalabe and uses most of the functions and operators in the ROBDD package.

Chapter 2

Design

In section 1.3 we stated the requirements specification to the ROBDD package. In this chapter we will discuss which building blocks (i.e. programming techniques and data structures) are needed to construct an ROBDD package from this specification.

In section 2.1 the discussion will concentrate on a suitable ROBDD representation. This basic representation will be extended as the ROBDD package is developed and all building blocks are put together. In section 2.2 we present an operator to create ROBDD nodes, and in section 2.3 we then show how to construct a large set of manipulation operators ($\wedge, \vee, \neg, \exists x.t$, etc.) from just a few general and very efficient operators. The efficiency of these operators will be crucial to the overall perfomance of the ROBDD package, and in section 2.4 we describe the necessary programming techniques (memoization) to improve their running time. Finally, in section 2.5 we give an outline of an automatic garbage collector to reduce the memory consumption of the ROBDD package.

2.1 Representing ROBDDs

In this section we first introduce a compact representation for ROBDDs, which will form the basis of the final ROBDD representation in C. We then show how to construct ROBDDs within this representation, and finally we give some examples to illustrate the importance of the variable ordering.

Recall from section 1.1 that an ROBDD is a DAG where each node has a variable identification (var) and two edges ($high$ and low) pointing to two sub-DAGs. We will therefore use the following representation for an ROBDD:

Each node is represented as a $(var, high, low)$ entry in an array indexed from 1. The $high$ and low edges are indices to the entry of the corresponding $high$ or low node, and var is a (not further specified) variable identification. Thus, each node has a unique number (the index in the array) that other nodes can refer to. If we also, for all ROBDDs, impose the same variable ordering in advance, we can represent the nodes of *all* ROBDDs in one single array of $(var, high, low)$ entries, the robdd_table. Each ROBDD now simply becomes a *root* identification (i.e. an index to the robdd_table).

This *global* representation has several advantages. First, it dramatically reduces the memory requirements by allowing sharing of not only sub-DAGs *within* an ROBDD, but also sub-DAGs *between* ROBDDs. Furthermore, it makes it possible to perform all memory allocation at once when the package is initialized, as opposed to the *local* approach where each ROBDD has its own robdd_table that must be allocated and de-allocated dynamically. The only disadvantage is, that some advanced garbage collection techniques must be developed. This will be discussed in section 2.5.

Assume that some ROBDD contains a sub-DAG d representing the expression e, and suppose that some other ROBDD contains a sub-DAG representing $\neg e$. Unfortunately, the proposed representation does not allow sharing of such sub-DAGs, unless we introduce an extra refinement: *complement edges*. A complement edge is simply a regular (*high* or *low*) edge, but with a sign. If an edge is negative, the DAG d pointed to is interpreted as $\neg d$. Thus, we only need one terminal node. We choose to use the entry with index 1 in the robdd_table to represent $true$, and consequently -1 will represent $false$.

Before we present an algorithm for creating nodes in the robdd_table, we discuss some alternatives for deciding where a new node should be inserted in the robdd_table. The obvious solution is to have counter to the next free entry in the robdd_table, and to increment this counter whenever a new node is created. This solution works only if the free entries in the robdd_table are contiguous. Another solution is to introduce a new variable *next* in each entry b_i in the robdd_table. *next* points to the next free entry or zero if b_i is the last free entry. This solution works independently of how the entries are organized in the robdd_table and as we develop a garbage collector it turns out to be the best alternative.

To summarize, our ROBDD representation consists of an array of nodes (robdd_table), where each node consists of four variables: a variable identification *var* not further specified, a *high* edge holding the index to the sub-DAG corresponding to $var = true$, and a *low* edge holding the index to the sub-DAG corresponding to $var = false$. If an edge to a sub-DAG is negative, the edge is a complement edge and the truth value of that sub-DAG is negated when the node is evaluated. Finally, *next* is a pointer to the next free entry in the robdd_table. A global variable $freelist$ is used to hold the index of the first free entry in the robdd_table. In section 2.5 this representation is extended with further two variables.

2.2 Creating ROBDDs

We are now ready to present the operator for creating nodes in the robdd_table, CREATE-NODE (see figure 2.1). Recall from the definition of an ROBDD, that in order to be a reduced OBDD, no two nodes should denote isomorphic sub-DAGs. This means, that some precaution must taken when a node is to be constructed: *a node should only be created if an equivalent node has not been created before* – this will maintain a canonical form in the robdd_table.

This introduces two new problems: node equivalence check and checking whether a node has been created before. The latter problem (node memoization) will be discussed in section 2.4. Concerning the former problem, let $b = (var, high, low)$ be a node in the

```
CREATE-NODE(var, high, low)
1    neg ← high < 0
2    if high = low then return high
3    if high < 0 then high ← −high
4                    low  ← −low
5    res ← lookup(unique_table, var, high, low)
6    if res = 0 then
7       if freelist = 0 then error("out of memory")
8       res ← freelist
9       freelist ← next(freelist)
10      robdd_table[res] ← (var, high, low)
11      insert(unique_table, res, var, high, low)
12   if neg then return −res else return res
```

Figure 2.1: The CREATE-NODE operator.

robdd_table. Then the following four pairs of nodes are equivalent:

$$(var, high, low) \equiv -(var, -high, -low) \quad (2.1)$$

$$(var, high, -low) \equiv -(var, -high, low) \quad (2.2)$$

$$-(var, high, low) \equiv (var, -high, -low) \quad (2.3)$$

$$-(var, high, -low) \equiv (var, -high, low) \quad (2.4)$$

As shown in [2] the canonical representation can be maintained if we always choose a node where the *high* edge is a regular edge (i.e the four nodes on the left in the equivalences).

CREATE-NODE works as follows: In line 1 we store the sign of the *high* edge. In line 2 we ensure, that no immediate redundant nodes are created. In line 3 and 4 we choose a node with a regular *high* edge, according to the equivalences (2.1) - (2.4). In line 7-11 we only create the node if it has not been created before. This is determined using the unique_table which memoizes all previously created nodes. In line 8 $freelist$ points to the next free entry in the robdd_table which is where the new node is to be stored. In line 9 $freelist$ is updated, and in line 10 the node is inserted in the robdd_table. In line 11 we store the node and its robdd_table entry in the unique_table, and in line 12 the sign of the result is set according to (2.1) - (2.4).

Assuming that *insert* and *lookup* are constant time operations the running time of CREATE-NODE is $O(1)$.

Figure 2.2(a) shows the ROBDD for the expression[1] $(x_1 \leftrightarrow y_1) \wedge \neg(x_2 \leftrightarrow y_2) \leftrightarrow (x_2 \leftrightarrow y_2)$ using the variable ordering $x_1 < x_2 < y_1 < y_2$. Figure 2.2(b) shows the ROBDD for the same expression, but using a different variable ordering $x_1 < y_1 < x_2 < y_2$.

[1] In section 2.3 we show how to construct such an expression. The actual tables in figure 2.2 are produced using the `print_table` function.

$root = -15$

i	var	$high$	low
1	–	–	–
2	x_1	1	-1
3	x_2	1	-1
4	y_1	1	-1
5	y_2	1	-1
6	x_2	5	-5
7	x_1	4	-4
8	y_1	1	-5
9	y_1	1	5
10	x_2	9	8
11	y_1	5	-1
12	y_1	5	1
13	x_2	12	-11
14	x_1	13	10
15	x_1	10	13

(a)

$root = -11$

i	var	$high$	low
1	–	–	–
2	x_1	1	-1
3	y_1	1	-1
4	x_2	1	-1
5	y_2	1	-1
6	x_2	5	-5
7	x_1	3	-3
8	y_1	1	6
9	y_1	6	1
10	x_1	9	8
11	x_1	8	9

(b)

Figure 2.2: The ROBDD representing $(x_1 \leftrightarrow y_1) \wedge \neg(x_2 \leftrightarrow y_2) \leftrightarrow (x_2 \leftrightarrow y_2)$. (a) With ordering $x_1 < x_2 < y_1 < y_2$. (b) With ordering $x_1 < y_1 < x_2 < y_2$.

Note, that the ROBDD in (a) contains eight internal nodes (5, 8, 9, 10,11,12,13 and 15) whereas the ROBDD in (b) only contains five internal nodes (5,6,8,9 and 11).

Obviously, the ordering in (b) is better than the ordering in (a). The rule of thumb is, that variables close to each other in an expression (e.g. x_1 and y_1 in figure 2.2) should also be close to each other in the variable ordering. Note also, that both ROBDDs contain some redundant nodes (6,7 and 14 in (a), and 7 and 10 in (b)). This will be discussed in section 2.5 where we construct a garbage collector.

2.3 Manipulating ROBDDs

We now introduce the core manipulation operator of the ROBDD package, the ITE operator [2]. Formally, the ITE operator is defined as follows[2]:

$$\text{ITE}(f, g, h) = f \wedge g \vee \neg f \wedge h \tag{2.5}$$

Shannon's decomposition theorem states:

$$f = v \wedge f_v \vee \neg v \wedge f_{\neg v} \tag{2.6}$$

where f_v is f evaluated at $v = true$, and $f_{\neg v}$ is f evaluated at $v = false$.

[2]In the following we assume that $\wedge$ has higher precedence than $\vee$.

```
ITE(f, g, h)
1   if terminal case then return res
2   res ← lookup(ite_table, f, g, h)
3   if res = 0 then
4      v ← top_variable({f, g, h})
5      res ← CREATE-NODE(v, ITE(f_v, g_v, h_v), ITE(f_¬v, g_¬v, h_¬v))
6      insert(ite_table, res, f, g, h)
7   return res
```

Figure 2.3: First version of the ITE operator.

Now, let $f = (w, t, e)$ be a node with variable w and *high* edge t and *low* edge e. If $v < w$, then f does not depend on v (according to the variable ordering), and $f_v = f_{\neg v} = f$. Thus we have

$$f_v = \begin{cases} f & \text{if } v < w, \\ t & \text{if } v = w. \end{cases} \qquad \text{and} \qquad f_{\neg v} = \begin{cases} f & \text{if } v < w, \\ e & \text{if } v = w. \end{cases} \tag{2.7}$$

Combining equation (2.6) and (2.7) we obtain the following recurrence to compute $z = \text{ITE}(f, g, h)$:

$$\begin{aligned} z &= v \wedge z_v \vee \neg v \wedge z_{\neg v} \\ &= v \wedge (f \wedge g \vee \neg f \wedge h)_v \vee \neg v \wedge (f \wedge g \vee \neg f \wedge h)_{\neg v} \\ &= (v \wedge (f_v \wedge g_v \vee \neg f_v \wedge h_v)) \vee (\neg v \wedge (f_{\neg v} \wedge g_{\neg v} \vee \neg f_{\neg v} \wedge h_{\neg v})) \\ &= v \wedge \text{ITE}(f_v, g_v, h_v) \vee \neg v \wedge \text{ITE}(f_{\neg v}, g_{\neg v}, h_{\neg v}) \\ &= (v, \text{ITE}(f_v, g_v, h_v), \text{ITE}(f_{\neg v}, g_{\neg v}, h_{\neg v})) \end{aligned} \tag{2.8}$$

Thus, we have $\text{ITE}(f, g, h) = (v, \text{ITE}(f_v, g_v, h_v), \text{ITE}(f_{\neg v}, g_{\neg v}, h_{\neg v}))$, where v is the top variable of $\{f, g, h\}$. The terminal cases for the recurrence are:

$$\text{ITE}(1, f, g) = \text{ITE}(f, 1, -1) = \text{ITE}(-1, g, f) = \text{ITE}(g, f, f) = f \tag{2.9}$$

and

$$\text{ITE}(f, -1, 1) = -f \tag{2.10}$$

Figure 2.3 shows the first version of the ITE operator. In line 1 we return if we have reached one of the five terminal cases given in (2.9) and (2.10). In line 2 we check whether the ite_table already holds the result. If the result has not been computed before, we use the recurrence given in (2.8) to create a new node. Finally the result is stored in the ite_table and the entry to the robdd_table is returned. In chapter 3 we present a more concrete algorithm and discuss the running time of ITE. The ITE operator can be used to construct all Boolean operators as shown in figure 2.4.

Table	Name	Expression	Equivalent form
0000	false	$false$	-1
0001	and(f,g)	$f \wedge g$	$\textsc{Ite}(f, g, -1)$
0010	gt(f,g)	$f \wedge \neg g$	$\textsc{Ite}(f, -g, -1)$
0011	f	f	f
0100	lt(f,g)	$\neg f \wedge g$	$\textsc{Ite}(f, -1, g)$
0101	g	g	g
0110	xor(f,g)	$f \otimes g$	$\textsc{Ite}(f, -g, g)$
0111	or(f,g)	$f \vee g$	$\textsc{Ite}(f, 1, g)$
1000	nor(f,g)	$\neg(f \vee g)$	$\textsc{Ite}(f, -1, -g)$
1001	iff(f,g)	$f \leftrightarrow g$	$\textsc{Ite}(f, g, -g)$
1010	not(g)	$\neg g$	$\textsc{Ite}(g, -1, 1)$
1011	gte(f,g)	$f \leftarrow g$	$\textsc{Ite}(f, 1, -g)$
1100	not(f)	$\neg f$	$\textsc{Ite}(f, -1, 1)$
1101	lte(f,g)	$f \rightarrow g$	$\textsc{Ite}(f, g, 1)$
1110	nand(f,g)	$\neg(f \wedge g)$	$\textsc{Ite}(f, -g, 1)$
1111	true	$true$	1

Figure 2.4: All Boolean functions constructed by means of the Ite operator

The next operator we will consider is Restrict. The *restriction* of an ROBDD b with respect to a given variable assignment $v := v_0$, $v_0 \in \{true, false\}$, is the new ROBDD where all occurences of v have been replaced by v_0 written as $b[v \mapsto v_0]$. The algorithm in figure 2.5 shows how a restriction of an ROBDD f can be done. In line 1 we simply return f if it does not depend on the variable v. In line 2 the actual restriction is performed, and because of the variable ordering we simply return either the *high* or *low* edge depending on the value of val. If the restriction has not been computed before we use Create-Node to construct a new ROBDD where all occurences of v in the two sub-DAGs are recursively restricted to val. We must use Create-Node since restricting an ROBDD might result in both redundant nodes and isomorphic sub-DAGs.

Restrict and Ite can be used to create existential and universal quantifiers, as shown

```
Restrict(f, v, val)
1   if var(f) > v or f = ±1 then return f
2   if var(f) = v then if val then return high(f) else return low(f)
3   res ← lookup(restrict_table, f, v, val)
4   if res = 0 then
5      res ← Create-Node(var(f), Restrict(high(f), v, val), Restrict(low(f), v, val))
6      insert(restrict_table, res, f, v, val)
7   return res
```

Figure 2.5: The Restrict operator.

```
NODESIZE(f)
1    function ns(f)
2      if f = −1 then return 0
3      if f = 1 then return 1
4      res ← lookup(nodesize_table, f)
5      if res = 0 then
6        res ← 2^(var(high(f))−var(f)−1) × ns(high(f)) + 2^(var(low(f))−var(f)−1) × ns(low(f))
7        insert(nodesize_table, res, f)
8      return res
9    end ns
10
11   var(1) ← robdd_vars + 1
12   res ← 2^(var(f)−1) × ns(f)
13   var(1) ← 0
14   return res
```

Figure 2.6: The NODESIZE function.

below:

$$\exists x.t \equiv t[x \mapsto false] \vee t[x \mapsto true] \tag{2.11}$$

$$\forall x.t \equiv \neg \exists x. \neg t \tag{2.12}$$

The last two functions we will consider are NODESIZE and ANYSAT. NODESIZE(b) returns the number of *satisfying* variable assignments (i.e. the number of assignments for which the ROBDD b is $true$). If an ROBDD is satisfiable, then ANYSAT(b) returns an arbitrary satisfying variable assignment. In figure 2.6 an algorithm for computing the nodesize of an ROBDD is given. Again, a table (nodesize_table) is used to memoize all the previously computed results.

NODESIZE works as follows: In line 2 and 3 we have reached a terminal case and the corresponding nodesize is returned. In line 6 we recursively calculate the nodesize for the two sub-DAGs making use of the variable ordering.

An algorithm for ANYSAT(b) can be constructed by doing the following observation: If the top node of an ROBDD b is not $false$ then either $high(b)$ or $low(b)$ (or both) will point to a sub-DAG $d \neq \pm 1$, and thus the truth value of $var(b)$ can be determined accordingly and ANYSAT can move down one level in the ROBDD. The ANYSAT function can be useful in proofs by contradiction.

2.4 Dynamic Programming

In section 2.1 we used a unique_table to memoize all the nodes that had been created. Similary, in section 2.3 we introduced an ite_table, a restrict_table and a nodesize_table, to

memoize all previously computed results. Such memoization techniques are called *dynamic programming* [4], and serve two main purposes:

- In all operators (CREATE-NODE, ITE, RESTRICT and NODESIZE) memoization is used to *improve running time.*
- In CREATE-NODE the unique_table is also used to *maintain a canonical form* in the robdd_table.

Of course, the canonical form also results in an improved running time, since the operators can benefit from this property, but the primary purpose of using memoization in CREATE-NODE is to maintain a canonical form in the unique_table. The unique_table can thus be viewed as a mapping from (*var*, *high*, *low*) to entries in the robdd_table. From the pseudo code of the four operators it follows that some standard operations (*lookup*, *insert* etc.) should be available.

In chapter 3 we will develop some data structures implementing these memoization techniques. As we will show, two types of memoization are interesting in this context: *total memoization* (hash tables) and *partial memoization* (caches). Hash tables can be used to maintain a canonical form (and to improve running time), whereas caches can be used to improve running time by memoizing only *a part* of the computed results. For now it is important to understand, that the unique_table memoizes *all* the nodes, that are created by CREATE-NODE. As we will see, there are many ways of implementing these memoization data structures, but all of them make extensively use of *hash functions.*

2.5 Garbage Collection

Using the global ROBDD representation introduced in section 2.1 with a common robdd_table some memory management strategies must be developed. Consider the problem of deciding what to do if (or when) the robdd_table runs out of entries. The immediate solution would be simply to give up and demand that the user should provide more resources the next time he uses the package.

But perhaps we can do better: as indicated in figure 2.2 the robdd_table might contain nodes, that were created under the construction of an ROBDD, but turned out to be redundant (i.e. do not belong to any ROBDD). Or the user might have constructed a temporary ROBDD that is not needed anymore, and therefore can be $freed$. Thus, there is hope for a solution employing a mechanism to recycle these redundant entries in the robdd_table. Such a mechanism is called a *garbage collector.*

The garbage collector must be able to distinguish the redundant (or *dead*) nodes from the nodes that should not be recycled. One way of doing this is by means of a technique called *reference counting.* Before this technique is described, we introduce some notions.

If a user creates an ROBDD with root r_e then r_e is said to be *externally referenced.* As opposed to this, a node r_i is *internally referenced* if there exists a node whose *high* or *low* edges are $\pm r_i$.

```
GARBAGE-COLLECTOR
1    for i ← robdd_entries downto 2 do
2        if ref(i) > 0 then MARK(i)
3    unique_table ← empty
4    freelist ← 0
5    for i ← robdd_entries downto 2 do
6        if mark(i) then mark(i) ← false
7                        insert(unique_table(i, var(i), high(i), low(i))
8                    else robdd_table[i] ← (0, 0, 0)
9                        next(i) ← freelist
10                       freelist ← i
```

Figure 2.7: The GARBAGE-COLLECTOR.

Now, suppose that we extend each node with a reference count ref of the number of internal and external references. This count can be maintained incrementally: When a new node r is created $ref(high(r))$ and $ref(low(r))$ are both incremented by one, and when a user creates an ROBDD b, $ref(b)$ is also incremented by one. When a user frees an ROBDD b we decrement $ref(b)$, and if ref becomes zero, the $high$ and low nodes are recursively freed.

With this approach reference counting is used to maintain an accurate count of the number of dead nodes in the robdd_table. Therefore, garbage collection becomes fairly simple as described in [2], and can be run safely at any time.

The garbage collector we have developed uses a similar reference counting technique, but requires much less bookkeeping. Instead of counting both the internal and external references, we only count the external references. This means that we cannot run the garbage collector at any time, because this might recycle some nodes used in an ROBDD under construction. Therefore, we only invoke the garbage collector at "safe places" in the program. In section 3.6 we show how this is done by means of C++ constructors and destructors. For now, assume that the garbage collector is always called at "safe places".

Figure 2.7 shows how our garbage collector works. In line 1 we run through the robdd_table and if a node is externally referenced, it is *marked*. The MARK(b) function called in line 2 marks all the nodes that are reachable from b. This function is implemented in section 3.5. In line 3 the unique_table is cleared, and in the actual implementation the ite_table, restrict_table and nodesize_table are also cleaned up. In line 6 we determine whether a node is alive and therefore should be re-inserted in the unique_table or dead and therefore can be recycled (put on the $freelist$).

Recycling the dead nodes in a top-down manner in the robdd_table will make the $freelist$ run bottom-up, and new nodes will therefore also be created bottom-up. This makes the output from the functions that print the robdd_table (or part of it as `print_table`) more readable for humans. Thus, substituting line 1 and 5 with **for** $i \leftarrow 2$ **to** $robdd\_entries$ **do** will not change the correctness of the garbage collector.

2.6 Summary

In this chapter we have gradually constructed an ROBDD representation. To begin with, the representation contained an array of records with three variables *var*, *high* and *low*. For each node in the ROBDD these three variables hold the information about the variable identification and the high and low edges. When we constructed the CREATE-NODE operator, we extended the representation with a variable *next* to point to the next free entry in the robdd_table. As we developed a garbage collector the representation was extended with two extra variables in each record: *ref* and *mark*. *ref* holds the number of external references to the node and *mark* is a bit field used in the marking process.

Along with this representation we constructed some operators for creating ROBDDs (CREATE-NODE) and manipulating them (ITE, RESTRICT). These three operators will form the core of the ROBDD package. Finally, we discussed some dynamic programming techniques to both improve runnning time of the core operators, and to maintain a canonical form in the robdd_table.

Chapter 3

Implementation

In the previous chapter we developed an ROBDD representation (the robdd_table) together with a constructor (CREATE-NODE) and some manipulators (primarily ITE and RESTRICT). We also described some programming techniques and discussed how to construct a garbage collector. In this chapther we will show how we have implemented an ROBDD package in C/C++ from these building blocks. The complete source code is listed in appendix B (section B.1).

In section 3.1 we show how the ROBDD package is structured and the modules it consists of. In section 3.2 we introduce an efficient randomized hash function that is used in the unique_table, ite_table, restrict_table and nodesize_table. In section 3.3 we show how the robdd_table is implemented and we show how it can be merged with the unique_table. In section 3.4 we discuss different memoization techniques, and in section 3.5 we state how we have implemented the manipulation operators. Finally, in section 3.6 we show how an automatic garbage collector can be implemented by means of the garbage collection algorithm given in section 2.5 and C++ constructors and destructors.

3.1 Structure

Figure 3.1 shows the modules of the ROBDD implementation. The hash module is a collection of hash functions used in both the unique_table and in the three modules, ite_table, restrict_table and nodesize_table. The robdd_table module implements the ROBDD node representation and also provides some useful methods for accessing the representation. The robdd_functions module contains the implementation of the node constructor and the ROBDD manipulators. This module also implements the garbage collector described in section 2.5.

The `robdd` module implements the user interface to the ROBDD package: a function to initialize the package and set the variable ordering, a set of Boolean operators (see figure 2.4), and some functions to print the ROBDDs. These functions will be described in chapter 4. As the figure shows, the kernel of the ROBDD package is implemented in C, whereas the user interface in implemented in C++. C++ was primarily chosen because

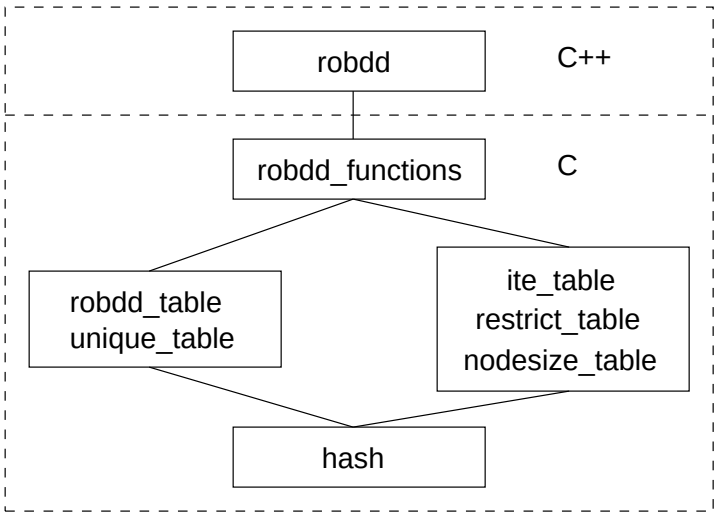

Figure 3.1: The Structure of the ROBDD package.

it provides a concept of constructors and destructors, which can be used in the reference counting process. How this is done is shown in section 3.5.

3.2 Hash Functions

Recall from section 2.4 that the unique_table can be viewed as a mapping from (var, $high$, low) to entries in the robdd_table. Such a mapping can be realized as a hash table and we thus need to develop some efficient hash functions.

The hash function we will use is described in [7] and is based on a programming technique called *randomization*. We will not go into detail about the properties of randomized algorithms but instead refer to [6]. The basic idea behind the randomized hash function is the following: First an array T with 2^{16} entries is constructed, such that for each entry i, $T[i] = i$, and then a *random* permutation of the entries in T is performed.

Now, an algorithm for calculating the hash value for a 32-bit integer x is shown in figure 3.2. First, in line 1 and 2 we lookup two "random" 16-bit integers s_1 and s_2 in T based on the lower 16 bits in x and $x+1$. Then, in line 3 we perform a bitwise exclusive or of s_1 and the upper 16 bits of x and use this 16-bit integer as an entry in T. Similary with s_2 and $high\_word(x)$ in line 4. Finally, in line 5 and 6 s_2 and s_1 are used as the high and low words in the hash value, which is masked with $size - 1$. If $size = 2^n$ for some n then the hash values will be in the range $0, 1, \ldots, size - 1$. If $size$ is not chosen as a power of 2, the hash values will not be uniformly distributed over the range $0, 1, \ldots, size - 1$.

Hashing a pair (x_1, x_2) of 32-bit integers is done by running the described algorithm twice as shown in figure 3.3. Note, that the intermediate value of s_1 based on x_1 in line 5 is xor'ed with $low\_word(x_2)$. Similary with s_2 and $low\_word(x_2 + 1)$. Thus it can be shown [7], that the hash value of (a, b) with high probability is different from (c, d) if $(a, b) \neq (c, d)$. This is one of the main advantages of using the randomization in the hash functions.

```
HASH(x, size)
1    s_1 ← T[low_word(x)]
2    s_2 ← T[low_word(x + 1)]
3    s_1 ← T[s_1 bitwise-xor high_word(x)]
4    s_2 ← T[s_2 bitwise-xor high_word(x)]
5    low_word(s) ← s_1
6    high_word(s) ← s_2
7    return s bitwise-and (size − 1)
```

Figure 3.2: The basic HASH function.

```
HASH2(x_1, x_2, size)
1    s_1 ← T[low_word(x_1)]
2    s_2 ← T[low_word(x_1 + 1)]
3    s_1 ← T[s_1 bitwise-xor high_word(x_1)]
4    s_2 ← T[s_2 bitwise-xor high_word(x_1)]
5    s_1 ← T[s_1 bitwise-xor low_word(x_2)]
6    s_2 ← T[s_2 bitwise-xor low_word(x_2 + 1)]
7    s_1 ← T[s_1 bitwise-xor high_word(x_2)]
8    s_2 ← T[s_2 bitwise-xor high_word(x_2)]
9    low_word(s) ← s_1
10   high_word(s) ← s_2
11   return s bitwise-and (size − 1)
```

Figure 3.3: A HASH2 function to hash pairs of integers.

A function for hashing a triple of 32-bit integers has also been developed using the same principle.

The implementation in [7] only uses a randomized array with 2^8 entries. This involves some extra shifting and exclusive or operations, but the principle is the same. In chapter 5 we will analyze which of the two approaches give the best performance.

In our implementation we use a precomputed randomized array with either 2^8 of 2^{16} entries depending on the presence of a C macro `HASH16` (this is described in appendix A). It would also be possible to compute a randomized array T each time the ROBDD package is initialized. This would change T from session to session, and possibly changing the performance of the package. This might be desirable in some situations, but in order to be able to perform internal comparisons, we have decided to use a fixed T.

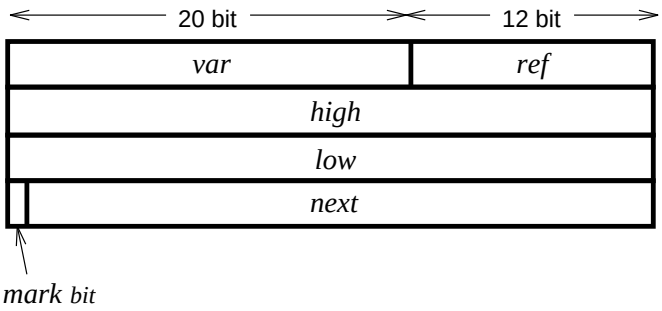

Figure 3.4: Representation of an ROBDD node.

3.3 Representing ROBDDs

Figure 3.4 shows how each ROBDD node in the robdd_table is implemented. A C struct robdd_entry taking up 16 bytes of memory is used to hold the var, $high$, low, ref, $next$ and $mark$ fields we introduced in chapter 2. We have chosen to represent variables as integers at this level in the ROBDD package. In section 4.1 we show how a mapping from these variable numbers to user-defined identifiers (strings) can be established. Note that var and ref are fit into one double word. Thus, in our implementation we can have upto 2^{20} variables, and a node can have upto 2^{12} external references. The robdd_table is an array of robdd_entrys.

The unique_table is implemented as a hash table using the hash function described in section 3.2. When a node (var, $high$, low) and a key k_1 is inserted, the hash value h for the triple (var, $high$, low) is first computed, and then k_1 is stored at entry h in the unique_table. If another key k_2 has previously been stored at this entry, a collision has occured and must be resolved.

Since the $next$ fields are only used to chain $unused$ nodes, we can store k_2 in $next(k_1)$. Thus, the $next$ fields of all the used entries in the robdd_table are used to maintain a linked list where each $next$ field points to the index of the robdd_table entry of the next node in the chain. Looking up a node in the unique_table is simply searching through this linked list. If the hash values are uniformly distributed over the number of entries in the unique_table, then few collisions will occur, and thus lookup will be an $O(1)$ operation.

We have also implemented two extra update functions $delete$ and $resize$, and a function for examining the performance of the unique_table. These functions are not used in the ROBDD package at this moment, but are include to be used in future work.

Amalgamating the robdd_table and the unique_table into a hybrid data structure results in both a lower memory consumption, and faster running time because all memory allocations can be done at once when the package is initialized.

3.4 Caches and Hash Tables

To improve running time, the core operators CREATE-NODE, ITE and RESTRICT and the NODESIZE operator use memoization. The unique_table, ite_table, restrict_table and nodesize_table could therefore be implemented as hash tables with ordinary linked lists resolving collisions (unlike the trick with the *next* field in the unique_table). Another solution could be a *cache* [2], i.e. a data structure similar to a hash table, but with only one slot for each entry in the table instead of linked list. This saves us from dynamic memory allocation, and makes insertion, deletion and looking up extremely simple. But insertion may now result in overwriting an old entry, and such an *eviction* may be expensive. Thus, the hash functions' ability to separate or spread the hash values becomes crucial. This will be examined in section 5.2.

In order to find out which of the two data structures have the best performance we implement both. In fact, we have also implemented a third solution, an $m \times n$ cache, i.e. a cache with m slots for each of the n entries. This introduces some interesting questions of which one of the m elements to evict, if all the slots are occupied. Our solution simply uses the m slots in a FIFO manner, but as shown in [6] randomization can be used to achieve a better performance.

Now, suppose that we decide to allocate a fixed amount of memory to a $m \times n$ cache. Then one can argue, that if the hash function is "good" (i.e. the hash values are spread as uniformly as possible over a given range) then a cache with mn entries will outperform an $m \times n$ cache, due to the extra time needed to maintain and search through the m elements. In chapter 5 we will examine how good the hash functions in figure 3.2 and 3.3 are in practice, but for now we will discard the $m \times n$ cache and concentrate on ordinary hash tables and caches.

The ite_table and restrict_table are compiled as either hash tables or caches, depending on the presence of the C macros `IT_CACHE` and `RT_CACHE` (this will be further explained in appendix A). nodesize_table is implemented as a cache which will be justified in chapter 5. All these tables are tailored versions of the generic cache and hash_table modules which can be found section B.3 in appendix B.

3.5 Manipulating ROBDDs

We are now ready to use the auxiliary data structures described in section 3.2, 3.3 and 3.4 in the implementation of the core operators.

As shown in section B.1.14 the CREATE-NODE operator is implemented directly from the algorithm given in figure 2.1. The ITE operator outlined in figure 2.3 requires some extra refinements. First, we choose a *standard triple* (see figure 3.5) to reduce the number of operations on the ite_table and eliminating the potential recomputation of equivalent results.

CHOOSE-STANDARD-TRIPLE(f, g, h)
1 **if** $f = \pm g$ **then return** $(f, \pm 1, h)$
2 **if** $f = \pm h$ **then return** $(f, g, \mp 1)$

CHOOSE-EQUIVALENT-TRIPLE(f, g, h)
1 **if** $g = \pm 1$ **then if** $f >_v h$ **then return** (h, g, f) **else return** $(-h, g, -f)$
2 **else if** $h = \pm 1$ **then if** $f >_v g$ **then return** $(-g, -f, h)$ **else return** (g, f, h)
3 **else if** $g = -h$ **and** $f >_v g$ **then return** $(g, f, -f)$ **else return** (f, g, h)
4 **else return** (f, g, h)

Figure 3.5: Two auxilary functions used in the ITE operator. The following abbreviation is used: $f >_v g \equiv var(f) > var(g) \vee \big(var(f) = var(g) \wedge |f| > |g|\big)$.

Next, consider the following *equivalent triples* [2]:

$$\text{ITE}(f, 1, g) \equiv \text{ITE}(g, 1, f) \tag{3.1}$$
$$\text{ITE}(f, g, -1) \equiv \text{ITE}(g, f, -1) \tag{3.2}$$
$$\text{ITE}(f, g, 1) \equiv \text{ITE}(-g, -f, 1) \tag{3.3}$$
$$\text{ITE}(f, -1, g) \equiv \text{ITE}(-g, -1, -f) \tag{3.4}$$
$$\text{ITE}(f, g, -g) \equiv \text{ITE}(g, f, -f) \tag{3.5}$$

Among these five pairs, we choose the triple whose first argument has the smallest top variable, or – in case of a tie – is numerically smallest.

Now, ITE can be implemented from the pseudo code listed in figure 2.3. The final version of the ITE operator is shown in figure 3.6. In line 1 we choose a standard triple, and in line 2-4 we check for the five terminal cases given in (2.9). In line 5 we choose a unique triple among the five pairs in (3.1)-(3.5).

In line 8-37 the recurrence (2.8) is used. There are five main cases: $g = \pm 1$, $h = \pm 1$, $var(f) > var(g)$, $var(f) < var(g)$ and $var(f) = var(g)$, and for each of these five cases there are three sub-cases. In line 10, for instance, $g = \pm 1$ and $var(f) > var(h)$, and thus $var(h)$ is the smallest variable of $\{f, g, h\}$ and we create a new node with variable $var(h)$. Because of the variable ordering, $f_{var(h)} = f_{\neg var(h)} = f$ and similary for g and finally $h_{var(h)} = high(h)$ and $h_{\neg var(h)} = low(h)$, see (2.7).

Note, that in all of the recursive calls, at least one of the arguments is reduced to its *high* or *low* edge. This guarantees termination of the algorithm. Furthermore, if ite_table holds all previously computed results the running time of ITE is $O(|f||g||h|)$, where $|\cdot|$ denotes the number of nodes in an ROBDD. Note, that if the three arguments have many variables in common, two or three levels may be processes in each recursive call (as e.g. in line 35-36). Thus, we might hope for a running time which is close to the size of the resulting ROBDD.

According to figure 2.5 the restrict_table should contain three variables in each entry: the root f on which the restriction has been performed, a variable identification v and

```
ITE(f, g, h)
1   (f, g, h) ← CHOOSE-STANDARD-TRIPLE(f, g, h)
2   if f = 1 then return g
3   if g = ±1 and h = ∓1 then return ±f
4   if g = h or f = −1 then return h
5   (f, g, h) ← CHOOSE-EQUIVALENT-TRIPLE(f, g, h)
6   res ← lookup(ite_table, f, g, h)
7   if res = 0 then
8     if g = ±1 then
9       if var(f) > var(h) then
10        res ← CREATE-NODE(var(h), ITE(f, g, high(h)), ITE(f, g, low(h)))
11      else if var(f) = var(h) then
12        res ← CREATE-NODE(var(h), ITE(high(f), g, high(h)), ITE(low(f), g, low(h)))
13      else res ← CREATE-NODE(var(f), ITE(high(f), g, h), ITE(low(f), g, h)))
14    else if h = ±1 then
15      if var(f) > var(g) then
16        res ← CREATE-NODE(var(g), ITE(f, high(g), h), ITE(f, low(g), h))
17      else if var(f) = var(g) then
18        res ← CREATE-NODE(var(g), ITE(high(f), high(g), h), ITE(low(f), low(g), h))
19      else res ← CREATE-NODE(var(f), ITE(high(f), g, h), ITE(low(f), g, h)))
20    else if var(f) < var(g) then
21      if var(f) < var(h) then
22        res ← CREATE-NODE(var(f), ITE(high(f), g, h), ITE(low(f), g, h))
23      else if var(f) = var(h) then
24        res ← CREATE-NODE(var(f), ITE(high(f), g, high(h)), ITE(low(f), g, low(h)))
25      else res ← CREATE-NODE(var(h), ITE(f, g, high(h)), ITE(f, g, low(h))))
26    else if var(f) > var(g) then
27      if var(g) < var(h) then
28        res ← CREATE-NODE(var(g), ITE(f, high(g), h), ITE(f, low(g), h))
29      else if var(g) = var(h) then
30        res ← CREATE-NODE(var(g), ITE(f, high(g), high(h)), ITE(f, low(g), low(h)))
31      else res ← CREATE-NODE(var(h), ITE(f, g, high(h)), ITE(f, g, low(h))))
32    else if var(f) < var(h) then
33        res ← CREATE-NODE(var(f), ITE(high(f), high(g), h), ITE(low(f), low(g), h))
34      else if var(f) = var(h) then
35        res ← CREATE-NODE(var(f), ITE(high(f), high(g), high(h)),
36                                  ITE(low(f), low(g), low(h)))
37      else res ← CREATE-NODE(var(h), ITE(f, g, high(h)), ITE(f, g, low(h))))
38    insert(ite_table, res, f, g, h)
39  return res
```

Figure 3.6: The final version of the ITE operator.

the truth value assignment val. But if we simply represent $(v, true)$ as v and $(v, false)$ as $-v$, we only need to have two fields in each restrict_table entry, and thus hashing can be performed faster. Apart from this trick, the implementation directly follows the pseudo code.

Since the *insert* and *lookup* operations on the restrict_table take time $O(1)$ the running time of RESTRICT(f, v, val) is $O(|f|)$. The source code in section B.1.14 also shows how NODESIZE and ANYSAT have been realized in C.

The last function to be implemented is the garbage collector. Apart from the MARK function the algorithm given in figure 2.7 can be realized directly. Now, *recursion* would be the immediate approach in constructing such a MARK function, but unfortunately this does not quite work. The observation is, that when the garbage collector is called it is highly possible, that all resources have been allocated. Then, it might be impossible to allocate enough stack space to complete the recursion, if for example the robdd_table consists of one single (very large) ROBDD.

Thus, we need a marking algorithm, that neither uses the stack, nor any large data structures. This introduces a new problem: When we move down an edge in an ROBDD to mark its sub-DAG, we need to know where we came from, when the sub-DAG has been marked. The problem is unsolvable unless we allow the MARK function to change the state of the robdd_table temporarily. The key observation to the solution is, that when moving down say a *high* edge to a sub-DAG h, we can use the *high* edge of h to point back to where we came from. An algorithm implementing this idea is given in figure 3.7. It is based on a general marking algorithm by Knuth (see p. 417 in [5]), but has been tailored to work on ROBDDs. The running time of MARK(f) is $\Theta(|f|)$ since each edge is traversed twice.

3.6 Garbage Collection

This section describes how the garbage collector designed in section 2.5 can be incorporated to work automatically in the core manipulation functions that create ROBDD nodes by means of C++ constructors and destructors. We first briefly summarize the problems of constructing an automatic garbage collector with the reference counting technique we have chosen. We then recapitulate the properties of C++ constructors and destructors. And finally we show how to put all these building blocks together to implement an automatic garbage collection.

Recall from section 2.5 that since we only count the number of the external references, the garbage collector should only be invoked when the robdd_table is in a state where all non-dead nodes can be reached from the externally referenced nodes. Of course, we could demand that the user should keep track of all this, but this would make the package tedious to use: every time a calculation had been completed, the user should manually invoke the garbage collector if necessary.

Consequently, we want the package to perform the garbage collection automatically when a given memory criterion is exceeded. This means, that during the construction of

```
MARK(P_0)
1    T ← 0
2    P ← P_0
3    E1: Q ← high(P)
4        if Q ≠ ±1 and ¬mark(Q) then
5            high(P) ← T
6            T ← P
7            P ← Q
8            goto E1
9    E2: Q ← low(P)
10       mark(P) ← true
11       if Q ≠ ±1 and ¬mark(Q) then
12           low(P) ← T
13           T ← P
14           P ← Q
15           goto E1
16   while T ≠ 0 do
17           Q ← T
18           if ¬mark(Q) then T ← high(Q)
19               high(Q) ← P
20               P ← Q
21               goto E2
22           else T ← low(Q)
23               low(Q) ← P
24               P ← Q
```

Figure 3.7: The MARK algorithm.

an ROBDD we need to temporarily protect its (not yet reachable) sub-DAGs from garbage collection. This can be achieved by means of C++ constructors and destructors.

Thus, we have implemented the ROBDDs as a C++ class `robdd` with a set of C++ operators, constructors and destructors. An ROBDD is still an integer *root* which is the index to the robdd_table, but now two extra methods become available: a constructor and a destructor. A class constructor is a method that is called for each object instantiated. Similarily, a destructor is a function that is called whenever an object goes out of scope.

We now let the constructors and destructors do the reference counting. When a user assigns an ROBDD to a variable b of type `robdd` the class constructor is called, and $ref(b)$ is incremented indicating an external reference. When the variable goes out of scope the destructor is called and $ref(b)$ is decremented.

Suppose that we wrap the original ITE operator in a C++ enhanced ITE operator taking three `robdd`s as arguments, and returning an `robdd` as result. Then, if we in this operator construct an `robdd` from the result of calling the original ITE operator with the roots of the `robdd` arguments, the reference count of this new `robdd` is incremented (by the constructor) and we can safely call the garbage collector, if it is neccesary.

The condition which triggers the garbage collector is called `GC_GRIT` and is defined as follows:

$$\frac{|freelist|}{free\_mem} < 0.2 \tag{3.6}$$

where $|freelist|$ is the length of the $freelist$, and $free\_mem$ is the number of free entries in the robdd_table after the previous garbage collection (initially set to the total number of entries in the robdd_table). The value 0.2 is not a magic number, but chosen after running some tests. The length of the $freelist$ can be maintained incrementally as shown in the source code for the CREATE-NODE operator and the GARBAGE-COLLECTOR. Using this adaptive condition, we prevent the garbage collector from being called too frequently when the robdd_table is say 80 percent full containing mainly non-dead nodes.

The core operators are now responsible for calling the garbage collector ("cleaning up after themselves"), and thus the garbage collection will be totally transparent to the user. There are several advantages of doing garbage collection at this level in the package:

- a minimum of bookkeeping is done since the actual `robdd` construction is done by core C functions that do not have to update reference counts.
- a minimum of memory is needed by the ref variable, since it must only count the number of external reference, and thus more variables can be managed (see figure 3.4).

A parallel can be drawn to the garbage collection technique used in [2]. Recall, that in this solution ref holds both the internal and external references. This approach corresponds to letting CREATE-NODE work on `robdd`s rather than just integers. This makes it possible to do the garbage collection in CREATE-NODE (and not in the manipulator functions), since at this level all non-dead nodes are references.

We agree, that this solution is more elegant than our solution, since all memory related tasks (node allocation and recycling) can be performed in CREATE-NODE. But we are

certain that our solution is more efficient due to both less bookkeeping and less memory requirements. The running time of the GARBAGE-COLLECTOR is $\Theta(N)$ where N is the number of entries in the robdd_table.

3.7 Summary

In this chapter we have shown how an ROBDD package can be implemented in C/C++. In section 3.1 we gave an overview of the modules needed in an ROBDD package. In 3.2 we introduced a randomized hash function to be used in the hash tables and caches implemented in section 3.4. In section 3.3 we stated a compact representaion for ROBDD nodes and we showed how to merge the unique_table and the robdd_table into a single data structure.

The ITE operator and the non-recursive MARK algorithm was developed and implemented in section 3.5. In section 3.6 some different garbage collection approaches were elaborated, and we also introduced a C++ class `robdd` to be used as the final representation for ROBDDs. This representation and its manipulation will be described in the next chapter.

Chapter 4

Reference Manual

In this chapter we will describe the interface to the ROBDD package. The chapter consists of three sections: In section 4.1 we describe two functions for initializing the ROBDD package and give some examples of how to use them. In section 4.2 we give an overview of the `robdd` functions and operators, and in section 4.3 we specify the usage of some functions for printing `robdds`.

4.1 Initialization

In this section we first describe how to initialize the ROBDD package. After that, we show how to declare an array of variables and how to set a variable ordering.

Syntax

```
void package_init(uint bt, uint btv, uint ut, uint it, uint rt, uint nt)
```

Description

This function initializes the ROBDD package and allocates memory for the internal `robdd` representation and the various hash tables and caches. The parameter `bt` specifies the number of entries in the robdd_table. Thus, the value of `bt` sets the limit for the maximum number of `robdd` nodes. `btv` specifies the maximum number of variables to be used. The parameter `ut` specifies the number of entries in the unique_table. This number should be at least half the size of `bt` and a power of 2 as we stated in section 3.2.

The parameters `it` and `rt` specify the number of entries in the ite_table and the restrict_table respectively and should be approximately of the same size. The parameter `nt` specifies the number of entries in the nodesize_table. The three parameters `it`, `rt` and `nt` indirectly determine the performance of the ROBDD package, and should not be set too low. As for the unique_table the `it`, `rt` and `nt` parameters should be chosen as powers of 2.

Example

If you have 32 MB of memory available you can initialize the package as follows

```
package_init(1<<20, N, 1<<20, 1<<18, 1<<18, 1<<12);
```

This will allocate space for 2^{20} `robdd` nodes and 2^{20} entries in the unique_table. The ite_table and restrict_table will both have 2^{18} entries, and the nodesize_table will get 2^{12} entries. `N` specifies the number of variables.

Syntax

```
void package_done(void)
```

Description

This function will shut down the ROBDD package and free the allocated memory.

Syntax

```
robdd* order_init(uint N, uint from,int step, char* prefix, uint offset)
```

Description

This function is used to instantiate an array of `robdd`s and give them an ordering. The parameter `N` is the number of variables to be initialized and set into the ordering. Recall from chapter 2, that variables internally are represented as positive integers. The parameter `from` specifies which variable number the first `robdd` variable should have. The parameter `step` specifies the (positive or negative) step between the variable numbers.

To assign a name to each of the `N` `robdd` variables, the `prefix` parameter holds a string, that will prefix all the `N` variables. The `offset` specifies the initial value of a counter that will be concatenated with each `prefix`.

Examples

In order to specify the ordering $x_0 < x_1 < x_2 < x_3$ you would write

```
robdd* x = order_init(4, 1, 1, "x_", 0);
```

The reversed ordering $x_3 < x_2 < x_1 < x_0$ is obtained by

```
robdd* x = order_init(4, 4, -1, "x_", 3);
```

The ordering $x_0 < y_0$ is obtained by

```
robdd* x = order_init(1, 1, 1, "x_", 0);
robdd* y = order_init(1, 2, 1, "y_", 0);
```

The ordering $x_0 < x_1 < y_0 < y_1$ is obtained by

```
robdd* x = order_init(2, 1, 1, "x_", 0);
robdd* y = order_init(2, 3, 1, "y_", 0);
```

Operator	Description	Symbol	Usage
!	not	$\neg$	!p
*	and	$\wedge$	p * q
%	xor	$\otimes$	p % q
+	or	$\vee$	p + q
>	gt		p > q
<	lt		p < q
>=	gte	$\leftarrow$	p >= q
<=	lte	$\rightarrow$	p <= q
==	iff	$\leftrightarrow$	p == q
=	assignment	$=$	p = q

Figure 4.1: The robdd operators. The operator! has the highest precedence and the operator= has the lowest precedence. The operator* and operator% have the same precedence, and similarily for the >, <, >= and <= operators.

The ordering $x_0 < y_0 < z_0 < x_1 < y_1 < z_1$ is obtained by

```
robdd* x = order_init(2, 1, 3, "x_", 0);
robdd* y = order_init(2, 2, 3, "y_", 0);
robdd* z = order_init(2, 3, 3, "z_", 0);
```

4.2 Manipulators

In this section we describe the usage of the robdd operators and functions.

Syntax

```
robdd operator!(const robdd& f)
robdd operator*(const robdd& f, const robdd& g)
robdd& operator=(const robdd& b)
```

Description

All Boolean functions described in figure 2.4 (except for nand and nor) are also available as robdd operators. Figure 4.1 shows the symbols and precedence of these operators. The $\neg$ operator (operator!) is prefix and the other operators are infix. We admit that the symbol of the exclusive or operator (operator%) is not intuitive and that this is not in the spirit of C++ programming [8]. But the advantage is that the robdd operators have the same precedence as their corresponding Boolean operators.

Example

We will construct the robdd for the Boolean expression $f = x_0 \wedge \neg x_1 \leftrightarrow x_2$. Using the operators from figure 4.1 this is quite easy. Assume the variables are declared and an ordering has been set, then the robdd f is constructed as follows

```
robdd f = x[0] * !x[1] == x[2];
```

Syntax

```
robdd exists(const robdd& f, const robdd& x)
```

Description

This function performs an existential quantification of the top variable of the `robdd x` over the `robdd f`. Note that an `robdd` is returned. Per definition, if this `robdd` is `false` then there does not exist an `x` such that `f` is true.

Example

Let the `f` be the `robdd` for the expression $x_0 \wedge (x_1 \leftrightarrow x_2)$ with the variable ordering $x_0 < x_1 < x_2$. We want to find out whether a variable assignment of x_0 exists such that `f` is satisfiable.

This can be done by writing

```
robdd* x = order_init(3, 1, 1, "x_", 0);
robdd f = x[0] * (x[1] == x[2]);
robdd r = exists(f, x[0]);
print_table(r);
```

The result of the existential quantification is shown in the table below. It follows from the table that $r = x_1 \leftrightarrow x_2$, and thus the satisfying variable assignments are: $x_1 = x_2 = true$ and $x_1 = x_2 = false$.

$root = 5$

i	var	$high$	low
1	$-$	$-$	$-$
2	x_0	1	-1
3	x_1	1	-1
4	x_2	1	-1
5	x_1	4	-4

Syntax

```
robdd forall(const robdd& f, const robdd& x)
```

Description

This function performs a universal quantification of the top variable of the `robdd x` over the `robdd f`. Per definition, if this `robdd` is different from `false` then `f` is `true` for all values of the top variable of `x`.

Syntax

```
robdd restrict(const robdd& f, const robdd& v, int val)
```

Description

This function computes the `robdd` where all occurences of `v` in `f` have been replaced by `val`. Note, that `val=0` is interpreted as `false` and all other values are interpreted as `true`.

Syntax

```
double nodesize(const robdd& f)
```

Description

This function computes the number of satisfying variable assignments.

Example

To compute the nodesize for the expression $(x_0 \leftrightarrow x_1) \vee (x_2 \otimes x_3) \vee (x_4 \rightarrow x_5)$ with the variable ordering $x_0 < x_1 < x_2 < x_3 < x_4 < x_5$ we write:

```
robdd* x = order_init(6, 1, 1, "x_", 0);
robdd f = (x[0]==x[1]) + (x[2]%x[3]) + (x[4]<=x[5]);
double n = nodesize(f);
```

The result is `n = 60` which can be verified easily.

Syntax

```
void anysat(const robdd& f)
```

Description

This function computes a satisfying variable assignment and prints it in either plain ASCII format or as a LaTeX tabular depending on the how the ROBDD package was compiled (see appendix A).

Example

Let the `f` be the `robdd` for the expression $x_0 \wedge (x_1 \leftrightarrow x_2)$ with the variable ordering $x_0 < x_1 < x_2 < x_3$. A satisfying variable assignment can be found by writing

```
robdd* x = order_init(4, 1, 1, "x_", 0);
robdd f = x[0] * (x[1] == x[2]);
anysat(f);
```

The result is

$root = 6$

var	$value$
x_0	$true$
x_1	$true$
x_2	$true$

Variables not present in the table are don't cares (x_3 in the example).

Syntax

```
int equal(const robdd& f, const robdd& g);
```

Description

This function returns true if and only if the `robdd f` and `robdd g` represent the same Boolean expression.

Example

Let the `f` be the `robdd` for the expression $\neg(x_0 \vee x_1) \vee (x_2 \wedge x_3)$ and let `g` be the `robdd` for the expression $(x_1 \vee x_0) \rightarrow (x_3 \wedge x_2)$. The variable ordering is set to $x_0 < x_1 < x_2 < x_3$. To verify, that `f` and `g` are equal we write:

```
robdd* x = order_init(4, 1, 1, "x_", 0);
robdd f = !(x[0]+x[1]) + (x[2]*x[3]);
robdd g = (x[1]+x[0]) <= (x[3]*x[2]);
int eq = equal(f,g);
```

The result is `eq = 1`, and thus `f` and `g` denote the same Boolean expression.

Syntax

```
int tautology(const robdd& f);
```

Description

This function returns true if and only if the `robdd f` is a tautology, i.e. `f` is true for all variable assignments.

Syntax

```
int satisfiable(const robdd& f);
```

Description

This function returns true if and only if the `robdd f` is satisfiable, i.e. there exists a variable assignment such that `f` is true.

4.3 Printing Functions

Syntax

```
void print_table(const robdd& f)
```

Description

This function prints the entire robdd_table and the root of the `robdd f`. The output is in either plain ASCII format or as a LaTeX tabular.

Example

This function has been used in the example under the description of the function `exists`.

Syntax

```
void print_info(void)
```

Description

This function prints (static as well as dynamic) information about the ROBDD package. It displays information about the garbage collector and the hash tables and caches.

Example

This function is demonstrated in section 5.2.

Syntax

```
void print(const robdd& f)
```

Description

This function prints the `robdd f` in parenthetical form. Unless a parser is developed we suggest that the function `print_table` is used.

Chapter 5

Efficiency

During the development of the ROBDD package our implementations have been tested thoroughly to ensure both correctness and efficiency. Many of the proposed data structures and algorithms were discarded as they turned out to be either inefficient (e.g. the $m \times n$ cache) or insufficient (e.g. the first versions of the internal representation).

In this chapter we will discuss the efficiency of some of the implementations in the ROBDD package. In section 5.1 we briefly describe the problem we have used in the test of the ROBDD package. In section 5.2 we present the results of the tests we have performed, and in section 5.3 we analyze the results.

5.1 Milner's Scheduler

Milner's Scheduler is a concurrent programming problem that is usefull as a test example because it is scalable and its representation requires very large ROBDDs. Furthermore, it uses many of the functions and operators we have implemented in the ROBDD package.

In Milner's Scheduler, N cyclers are connected in a ring and co-operates on starting and detecting termination of N tasks. The tasks must be initiated in order, but can terminate in any order. This property is fulfilled by introducing a token which is passed between the cyclers. A cycler is only allowed to start its task if it holds the token. Initially, no cycler holds the token.

The state of each cycler can be described by three Boolean variables t_i, h_i and c_i. t_i indicates whether cycler i is running its task, h_i is *true* if and only if cycler i holds the token, and thus blocks the other cyclers from initiating their task. Finally, c_i is *true* if and only if cycler $i-1$ has put down the token, and cycler i has not yet picked it up.

With these definitions Milner's Scheduler can be modelled as a transistion system over Boolean variables. Such a modelling is carried out in [1].

In figure 5.1 only the main function of our implementation of Milner's Scheduler is shown. The complete listing can be found in appendix B.2. The first six lines in the figure set the following variable ordering

$$c_1 < c'_1 < t_1 < t'_1 < h_1 < h'_1 < \cdots < c_N < c'_N < t_N < t'_N < h_N < h'_N$$

```
robdd* c  = order_init(N, 1, 6, "c", 1);
robdd* cp = order_init(N, 2, 6, "c'", 1);
robdd* t  = order_init(N, 3, 6, "t", 1);
robdd* tp = order_init(N, 4, 6, "t'", 1);
robdd* h  = order_init(N, 5, 6, "h", 1);
robdd* hp = order_init(N, 6, 6, "h'", 1);

robdd I  = initial_state(t, h, c);
robdd T  = transitions(t, tp, h, hp, c, cp);
robdd R  = reachable_states(I, T, t, tp, h, hp, c, cp);
```

Figure 5.1: Modelling Milner's Scheduler with the ROBDD package.

In the following two lines the initial state and all possible transistions are computed. In the last line a *fix point iteration* computes the reachable states.

By the following argumentation the number of reachable states is $N2^{N+1}$ where N is the number of cyclers: Each cycler can be in one of four states: (1) The cycler has not yet picked up the token: $(c_i, h_i, t_i) = (1, 0, 0)$. (2) The cycler has picked up the token: $(c_i, h_i, t_i) = (0, 1, 0)$. (3) The cycler starts its task: $(c_i, h_i, t_i) = (0, 1, 1)$. (4) The cycler releases the token: $(c_i, h_i, t_i) = (0, 0, 1)$.

When cycler i is in one of its four states, the other cyclers are either running their task or not, that is $t_j = 0$ or $t_j = 1$ for $j \neq i$. Thus, the number of reachable states are $4N2^{N-1} = N2^{N+1}$. Note, that although the number of states is $O(2^N)$ we can hope for a more compact `robdd` representation.

5.2 Results

We will now use Milner's Scheduler to examine the efficiency of the ROBDD package we have implemented. We will first study the running time for calculating the reachable states in Milner's Scheduler using four different versions of the ROBDD package. Then, we will examine the performance of the hash functions.

In order to determine which of the hash functions and memoization data structures have the best performance, we have compiled four versions of the ROBDD package:

Version 1 uses a 16-bit hash function, ite_table and restrict_table are caches.

Version 2 uses a 16-bit hash function, ite_table is a cache and restrict_table is a hash table.

Version 3 uses a 16-bit hash function, ite_table is a hash table and restrict_table is cache.

Version 4 uses an 8-bit hash function, ite_table and restrict_table are caches.

Figure 5.2 shows the running times for Milner's Scheduler with $N \in \{10, 20, 30, 40, 50\}$ cyclers on a computer with RISC architecture. From column 1 and 4 it follows that an 8-bit

N	Version 1	Version 2	Version 3	Version 4
10	13	15	14	13
20	88	110	94	88
30	362	523	411	365
40	1155	1720	1331	1147
50	8671	12742	9393	8425

Figure 5.2: Running time in seconds for Milner's Scheduler with N cyclers on a 40 MHz DEC MIPS 3000 with 128 MB memory. Initial table sizes: robdd_table $= 2^{21}$, unique_table $= 2^{21}$, ite_table $= 2^{20}$ and restrict_table $= 2^{20}$.

N	Version 1	Version 4
10	4	5
20	59	77
30	930	1224
40	3134	4138

Figure 5.3: Running time in seconds for Milner's Scheduler with N cyclers on a 120 MHz 486 with 16 MB memory. Table sizes: robdd_table $= 2^{18}$, unique_table $= 2^{18}$, ite_table $= 2^{17}$ and restrict_table $= 2^{17}$.

hash function performs just as well as a 16-bit hash function. It can also be concluded, that caches perform better than hash tables. The number of reachable states was computed as `nodesize(R)`$/2^{3N}$ and was exactly $N2^{N+1}$ as expected.

Figure 5.3 shows the running times for Milner's Scheduler on a PC running Linux. On this CISC architecture it also turns out that caches preform better that hash tables, and thus we concentrate on the hash functions. As opposed to the results on the DEC MIPS the overall running time of the ROBDD package is about 25 percent faster using a 16-bit hash function instead of an 8-bit hash function.

To examine the hash functions' ability to separate hash values we have run Milner's Scheduler with 20 cyclers requirering about 1.5×2^{20} nodes. We allocated 2^{21} nodes for the robdd_table and 2^{21} entries in the unique_table. Figure 5.4 shows how the length of the linked lists in the unique_table are distributed. Note, that more than 90 percent of entries have three or less elements in their chain.

5.3 Analysis

From figure 5.2 it follows that a cache based memoization has a better performance than memoization based on hash tables. It also follows that the 8-bit and 16-bit hash functions have the same performance on the DEC MIPS. This might be due to the RISC architecture. On the PC with a CISC processor, however, the 16-bit hash function has a significantly better performance (a profiling reveals that 16-bit hashing runs approximately twice as fast as 8-bit hashing).

l	% of entires
0	24
1	34
2	24
3	12
4	4
5	1
≥ 6	1

Figure 5.4: Percentage of entries with chain-length l in a unique_table with about 1.5×2^{20} elements and 2^{20} entries.

The running times of the four versions of the ROBDD package in figure 5.2 and 5.3 are polynomial in N and a simple curve fitting shows a running time of $O(N^3)$.

A profiling of running Milner's Scheduler with 30 cyclers shows that more than half of the time is spent in the hash functions. This is not a surprising result, since almost all functions and operators in the ROBDD package use memoization which requires hashing. The running time of the package can thus be improved if a better hash function is developed. Furthermore, the profiling reveals that less than 5 percent of the total running time is used to do garbage collection which is acceptable.

From figure 5.4 it follows that the hash values produced by the hash function are well spread over the entries in the unique_table: only 24 percent of the entries contain no elements and 70 percent of the entries hold no more more than three elements. Thus, a *lookup* operation will approximately take constant time.

We realize that the ROBDD package ought to be tested on other problems in order to conclude that the package is efficient in general. But we believe that Milner's Scheduler gives a good hint on the efficiency of the ROBDD package.

Chapter 6

Conclusion

In this chapter we will first recapitulate the main points of the previous five chapters and then discuss the usability and efficiency of the final product of this bachelor project, the ROBDD package.

In chapter 1 we gave a short introduction to Reduced Ordered Binary Decision Diagrams, and formulated the problem statement — to develop an efficient ROBDD package. Then, in the requirements specification we stated that the ROBDD package should be a library providing an abstract datatype `robdd` with associated functions and operators. Furthermore, we required an automatic garbage collection mechanism to be developed.

In chapter 2 we discussed the design of the ROBDD package and developed both a compact internal representation of `robdd` nodes and outlined the core algorithms. In chapter 3 this design was realized in C/C++ and we provided different implementations of the data structures for memoization. Chapter 4 contained a reference manual to the interface of ROBDD package, and chapter 5 presented the results of running Milner's Scheduler on the ROBDD package.

6.1 Usability

A *usable* ROBDD package should provide an abstract datatype with an intuitive interface to manage Boolean expressions. We decided on implementing a C++ class `robdd` with a set of functions and operators:

The function `equal` makes it possible determine whether two Boolean expressions are equivalent. `satisfiable` and `tautology` are used to check whether a Boolean expression is true for some or all variable assignments, respectively. These three functions are all constant time operations when the `robdd`s have been build. The function `nodesize` is used to count the number of satisfying variable assignments and `anysat` returns one of these assignments.

The overloaded `robdd` operators (`operator*`, `operator+` `operator==`, etc.) implement the standard binary Boolean operators ($\wedge$, $\vee$, $\leftrightarrow$, etc.), and `exists` and `forall` enable existential and universal quantification of Boolean expressions.

The ROBDD package also provides an automatic garbage collector that — transparently to the user — takes care of the memory management. Thus, the user can concentrate on writing simple and compact code without having to worry about allocating and de-allocating `robdds`, as well as recycling unused memory.

Thus, the ROBDD package is a usable tool for representing systems that can be modelled with Boolean expressions.

6.2 Efficiency

In practice, *efficiency* and *storage requirements* are two desicive factors for the usability of the ROBDD package. Thus, fast algorithms and advanced programming techniques as well as a compact internal representation must be implemented. A compact representation was designed in chapter 2 and benefits from taking up only 16 bytes of memory.

The overall perfomance of the ROBDD package depends on three core operators: CREATE-NODE, ITE and RESTRICT, as all other operators and functions in the ROBDD package (except the printing functions) utilize these three operators.

To improve running time ITE and RESTRICT use either partial memoization (caches) or total memoization (ordinary hash tables) of the previously computed results. CREATE-NODE uses a hash table to memoize all previously created nodes to preserve a canonical form. This table was merged with the robdd_table to minimize storage requirements.

We have adopted a randomized hash function described in [7] and implemented an 8-bit and a 16-bit version hoping that the 16-bit version would be faster. This turned out to be the case on a PC running Linux, but on a RISC computer no significant difference was observed.

As we decided on a global ROBDD representation a garbage collection mechanism was developed. In practice, the garbage collector uses less than five percent of the total running time which we believe is acceptable.

The ROBDD package was tested on Milner's Scheduler showing that a system with 40 cyclers can be handled easily. Assuming that Milner's Scheduler is a good test example, we consider the implementation of the ROBDD package to be efficient.

6.3 Summary

Reviewing the requirements specification we conclude to have fulfilled the task of implementing an efficient ROBDD package. We also believe, that we have implemented a product which is usable in practice and can be included in larger programming projects.

In a prospective view, developing a faster hash function will improve the overall efficiency of the package. Another technique to improve the package is *dynamic variable ordering*. This technique uses heuristics to dynamically change the global variable ordering expecting the new ordering to result in smaller ROBDDs. Furthermore, the package can be extended with extra `robdd` functions, e.g. SIMPLIFY and ALLSAT described in [1].

These function are fairly simple to implement using the CREATE-NODE operator and the memoization techniques provided by the ROBDD package.

Bibliography

[1] Henrik Reif Andersen. *An Introduction to Binary Decision Diagrams.* Department of Information Technology, Technical University of Denmark, September 1995.

[2] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient Implementation of a BDD Package. *27th Design Automaton Conference*, pages 40–45, 1990.

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

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* McGraw-Hill, 1994.

[5] Donald E. Knuth. *The Art of Computer Programming, Fundamental Algorithms.* Addison-Wesley, 2nd edition, 1968.

[6] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University, 1995.

[7] Peter K. Pearson. Fast Hashing of Variable-Length Text Strings. *Communications of the ACM*, 33(6):677–680, June 1990.

[8] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edition, 1991.

Appendix A

Installation

This appendix shows how to install the ROBDD package under Linux or Unix. Note, that since the ROBDD library will be installed in `/usr/local/lib/` you must be super-user during the installation process[1].

A.1 Installing the ROBDD package

First, copy the `robdd-1.0.tgz` file from the enclosed disk to your home directory. Alternatively, you can download the file from `http://www.it.dtu.dk/~jmr/robdd-1.0.tgz`. Then type `tar zxvf robdd-1.0.tgz` to extract the contents of the archive. This will create a directory called `robdd/` where all the source files and examples are placed. Finally, change to the `robdd` directory and run `make` to complete the installation. This command will first compile all the source files and then install the `librobdd.a` library in `/usr/local/lib/` and the `robdd.h` header file in `/usr/local/include/`.

After you have installed the ROBDD package, you are ready to run the Milner's Scheduler example in the `bdd/ex/` directory: type `make milner` and then simply follow the on-screen instructions.

A.2 Configuring the ROBDD package

There are four C macro definitions, you can change before compiling the ROBDD package. If the `HASH16` macro is defined the hash functions will use a randomized table with 16-bit entries, otherwise a table with 8-bit entries is used. The difference between these tables is discussed in chapter 3.

If `IT_CACHE` and `RT_CACHE` are defined the ite_table and restrict_table modules are implemented as caches rather than ordinary hash tables where the chain resolution is solved using singly linked lists. Normally, a cache will be more efficient regarding both memory consumptions and running time than an ordinary hash table.

[1] Alternatively, you can change `LIB` and `INC` in the `Makefile`.

If `LATEX` is defined the printing functions will produce output that can be inserted into a LaTeXdocument.

The values of these four macros at compile time can be determined by calling the `print_info` function.

Appendix B

Source Code

B.1 ROBDD package

B.1.1 types.h

B.1.2 types.c

B.1.3 hash.h

B.1.4 hash.c

B.1.5 robdd_table.h

B.1.6 robdd_table.c

B.1.7 ite_table.h

B.1.8 ite_table.c

B.1.9 restrict_table.h

B.1.10 restrict_table.c

B.1.11 nodesize_table.h

B.1.12 nodesize_table.c

B.1.13 robdd_functions.h

B.1.14 robdd_functions.c

B.1.15 robdd.h

B.1.16 robdd.c

B.2 Milner's Scheduler

B.2.1 milner.h

B.2.2 milner.c

B.3 Auxiliary Modules

Appendix C

Diary

January

25th Initiation of project; first considerations about data structure; discussion on the approach and which subjects we find relevant.

29th Meeting at the Department of Information Technology; present Henrik Reif Andersen, Jesper Møller, Christian Østergaard; discussion of project: descision on litterature, global approach with garbage colletion, data structure, efficiency parameters: storage requirement and time consumption during computation, ordering given once and for all;

February

26th Meeting at the Department of Information Technology; present Henrik Reif Andersen, Jesper Møller, Christian Østergaard; further discussion of data structure, and garbage collection.

March

1st Final version of data structure — flag bit introduced, list of functions to be implemented.

4th Specification of functions to be implemented.

5th Considerations about hash tables, and hash functions. Randomization.

11th unique_table done

13th Discussion of different implementations of the needed tables; FIFO cache, randomize cache, dynamic array, linked list.

15th Implementation of the ITE operator and all basic functions on ROBDDs

24th Setting of deadlines for the basic algorithms.

25th Implementation of the RESTRICT function.

28th Discussion of general table versus specialized.

April

2nd Marking algorithm (Knuth) for garbage collection done

9th Algorithms for garbage collection done; discussion on approaches to clean up garbage.

11th Outline of report.

16th Report.

18th Report.

22nd Work on RESTRICT; Criterion for invoking GARBAGE-COLLECTOR decided.

23rd Discussion on implementation of automatic garbage collection.

26th All memory on freelist; speeds up CREATE-NODE.

29th Meeting at the Department of Information Technology; present Henrik Reif Andersen, Jesper Møller, Christian Østergaard; discussion about garbage collection; C++ interface solves problem concerning temporary nodes. Garbage collection; RESTRICT done. restrict_table is deleted during garbage collection. `exist` and `forall` functions done; temp flag removed; agreed on implementing ANYSAT and Milner's Scheduler.

May

3rd Work on C++ front end and garbage collection

5th Restructuring of modules; work on C++ interface and garbage collection; compilation and execution now possible.

6th C++ refinements, NODESIZE initiated

7th ANYSAT initiated, NODESIZE done, Milner's Scheduler initiated.

9th Milner's Scheduler.

13th Milner's Scheduler debugging.

16th Milner's Scheduler done.

19th Debugging.

20th Set order.

22nd Experiments regarding table structures; implementation of choises.

June

3rd-21st Report.