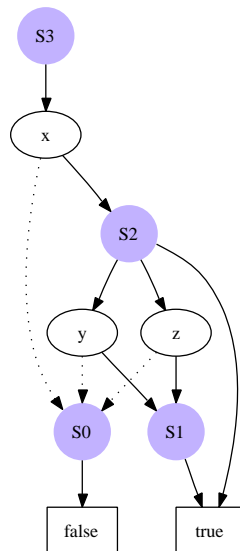


SBDD - Representing Sets of Boolean Functions

Patrick Wischnewski <wischnewski@react.cs.uni-sb.de>



Bachelor Thesis

Prof. Bernd Finkbeiner
Naturwissenschaftlich-Technische Fakultät I
Fachrichtung 6.2 – Informatik
Universität des Saarlandes, Saarbrücken, 2006



This thesis introduces *SBDDs*. *SBDDs* are an efficient data structure for representing sets of Boolean functions that are needed in many fields of computer aided verification. We will present an implementation of this data structure as well as all the operations that we need in order to deal with sets of Boolean functions. Additionally, we will present a *language inclusion checker* as an application of *SBDDs*. We can save exponentially many nodes compared to the standard approach for sets of *BDDs*.

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbst erstellt und dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Saarbrücken, den 27. September 2006

Acknowledgement

I would like to thank Prof. Finkbeiner for the helpful and friendly guidance I received from him during my time doing my Bachelor Thesis. I learned a lot whilst dealing with this topic and the talks and ideas given by him.

Contents

1	Introduction	9
2	Formal Definition of SBDDs	13
2.1	Syntax of SBDDs	13
2.2	Semantics of SBDDs	15
2.3	SBDD: Data Structure or Tree-Automaton?	16
2.4	Summary	20
3	Operations on SBDDs	21
3.1	Set Operations	21
3.1.1	UNION	22
3.1.2	INTERSECTION	22
3.2	Boolean Operations	22
3.2.1	OR	23
3.2.2	NOT	24
3.2.3	RESTRICT	25
3.2.4	FILTER	25
3.3	Building canonical form	29
3.3.1	DETERMINIZE	29
3.3.2	ELIMINATE-REDUNDANT-TESTS	30

3.3.3	MINIMIZE	31
3.4	Equality check	33
3.5	Summary	34
4	Implementing SBDDs	35
4.1	Storing Nodes	36
4.2	Making nodes unique	36
4.3	Computed table	38
4.4	Collect garbage	38
4.5	Design	39
5	Language containment checking	43
5.1	Symbolic Labeled Transition System	43
5.2	Language containment checking	44
5.3	Design	46
6	Experimental Results	49
6.1	Dining Philosophers	49
6.2	Mutual exclusion with manager	51
6.3	Semaphores	52
6.4	Summary	53
7	Summary	57
7.1	Future work	58

Chapter 1

Introduction

Binary decision diagrams (BDD) [And97] are an efficient data structure for representing Boolean functions. BDDs are a common tool in many applications, in particular in the area of formal verification. The main idea of this graph-based data structure is to reuse parts of the structure wherever possible.

A lot of effort has been spent on implementing packages that provide the functionality of BDDs very efficiently, for example [Som] or [Jan01]. A comparison between different implementations can be found at <http://www.bdd-portal.org>. These packages are already successfully used for developing *symbolic-model-checker*, for example [McM93].

Furthermore, there exist a variety of extensions or modifications of the BDD structure for special purposes such as *zero-suppressed binary decision diagrams* (ZBDD) [Min93, LSW95] that are designed in order to represent and efficiently manipulate sets of combinations. ZBDD is the underlying structure for clause encoding in the *Backtrack Search SAT Solver* presented in [AMS02].

Another extension of the conventional BDDs are the *Non-deterministic binary decision diagrams* (NBDD). They consist of two different kinds of nodes: The decision nodes like BDDs, and the structural nodes with an arbitrary number of successors. This non-determinism is interpreted as disjunction. Thus, NBDDs give a often more compact representation of Boolean formula than BDDs. NBDDs are successfully used for language containment checking of Büchi automaton [Fin01].

SBDDs were developed by [Fin], inspired by the need for an efficient data structure for representing and manipulating *sets of Boolean functions*, especially in the area of formal verification. This structure corresponds syntactically to NBDDs but with a different semantic. Thus, we have two kinds of nodes decision nodes, like in the BDD case, and structural nodes with an arbitrary number of successors that represent the union of its successors rather than the disjunction of them.

BECAUSE OF THEIR HIGH COMPRESSION SBDDs, ARE VERY USEFUL FOR PRACTICAL APPLICATIONS especially in formal verification. Since there has not been an implementation of this data structure so far, we will develop in this thesis a package efficiently implementing this data structure and the functionality for manipulating SBDDs. The implementation is based on the ideas of a pointerless BDD implementation [Jan01].

Furthermore, we will show the practical applicability concerning as an example a language inclusion checker for non-deterministic reactive systems. The underlying data structure for symbolically representing the reachable state space is the SBDD implementation we have developed.

In the second chapter we will give a formal definition of what a SBDD actually is. First we introduce the syntax of the SBDD and show how they are represented as a graph structure. Then we will show the semantical meaning of this data structure and how we can extract the represented set of Boolean functions. In the last section of the second chapter we show the equivalence between SBDDs and binary tree automaton. This implies that we can use both well known concepts of the graph theory of generic BDDs and well known concepts of tree automaton.

In the third chapter we introduce the algorithms for SBDD. We give the Boolean Operations (*and*, *or*, *not*), the set operations (*union*, *intersection*) and the operation needed for building a canonical form that base on algorithms of theory of binary tree automaton.

The fourth chapter shows how SBDDs can be implemented. First we show the graph structure as a pointerless data structure based on arrays. Secondly, we give the structures representing the nodes of the graph as well as the memory organisation. Then we will introduce important concepts for efficiently implementing the operations on SBDDs. The last part of this chapter treats the mark-sweep-clean garbage collector used for erasing obsolete nodes.

The fives chapter introduces a language inclusion checker for reactive systems based on the above methods and implementations of SBDDs. We will present a fixpoint computation method for exploring the reachable state space of the composition of two systems, namely implementation and specification. Analyzing the fixpoint answers the question of whether the language of implementation is a subset of the language of specification.

The sixth chapter relates the SBDDs to equivalent sets of BDDs and shows the exponential amount of memory saved. We will give evidence by considering certain language inclusion instances of the mutual exclusion problem. The first example will be an instance of the problem of dining philosophers. Given a system with observable events we will verify that the property holds that not all philosophers are able to eat

at the same time. Additional examples will be: *mutual exclusion with manager* and *mutual exclusion with semaphores*.

Chapter 2

Formal Definition of SBDDs

In this chapter we are dealing with the question: "How can we store a set of Boolean functions memory efficiently". We introduce the syntactic structure of the SBDD data structure as well as their semantic interpretation.

As introduced in [Fin] the syntax of SBDDs is similar to that of BDDs except that we have two different kinds of nodes in the SBDDs. We extend the concept of BDDs by introducing nodes that have an arbitrary number of successors. In the case of lists (or sets) of BDDs for representing sets of Boolean functions the sets (non determinism) are encoded in the root level (Multi-rooted BDDs). We extend this concept such that we allow sets of SBDDs in every level of the graph. This is done by introducing new nodes that have an arbitrary number of successors representing the union of all its successors.

There exist two semantic interpretations of the SBDD structure. One is similar to BDDs in that the denotation works directly on the graph. The other interprets a given SBDD as a binary tree automaton. Both of them give us the opportunity to apply well known algorithms for performing operations on the SBDD structure.

In contrast to BDDs, SBDDs are neither canonical nor minimal by itself. We can get rid of this lack by interpreting the SBDD as a binary tree automaton that has unique minimal form because there exist a well known method for building a unique minimal tree automaton.

2.1 Syntax of SBDDs

Now we talk about the basic idea of a SBDD, namely: we combine the idea of a BDD consisting of decision nodes with the idea of structural nodes that bring the opportunity

to have the set encoding of sets of Boolean function at the lowest possible level in the graph. The decision nodes have two successors which give the choices x or $\neg x$ for a variable x in a Boolean formula. However, the structural nodes have an arbitrary number of successor nodes meaning the union of the sets of functions encoded by the subtrees.

First, the following definition gives the syntax of a SBDD:

Definition 1 SBDD

A Structural Binary Decision Diagram (SBDD) is a directed acyclic graph $(\mathcal{V}, \mathcal{S}, \mathcal{D}, \phi, low, high, succs)$ s.t.

- \mathcal{V} set of variables
- \mathcal{D} set of decision nodes
- \mathcal{S} set of structural nodes
- a root node $\phi \in \mathcal{S}$
- two terminal nodes $0, 1$
- $var : \mathcal{D} \rightarrow \mathcal{V}$
- $low : \mathcal{D} \rightarrow \mathcal{S}$
- $high : \mathcal{D} \rightarrow \mathcal{S}$
- $succs : \mathcal{S} \rightarrow 2^{\mathcal{D}} \cup \{0, 1\}$

The function var assigns a variable $x \in \mathcal{V}$ to every decision node. The functions low and $high$ assigns to every decision node its high and low successor respectively. The function $succs$ gives to every structural node the set of its successor nodes which are decision nodes. From this definition, decision nodes and structural nodes appear alternately on a path in the graph. Further, a SBDD is an acyclic, undirected and connected graph. Hence, it is a tree structure. A SBDD is represented by its root node ϕ .

Figure 2.1 (a) illustrates a SBDD that represents the set $\{x \wedge y, x \wedge z, x \wedge true\}$. The ovals with a variable inside gives the decision nodes. The circles are the structural nodes. Solid lines display $high$ -edges and dotted lines low -edges. A node without an incoming edge is the root node. The variables inside of the decision nodes gives you the label of this node.

A formal definition of SBDDs has been introduced in this section such that we can define the semantical meaning in the next section.

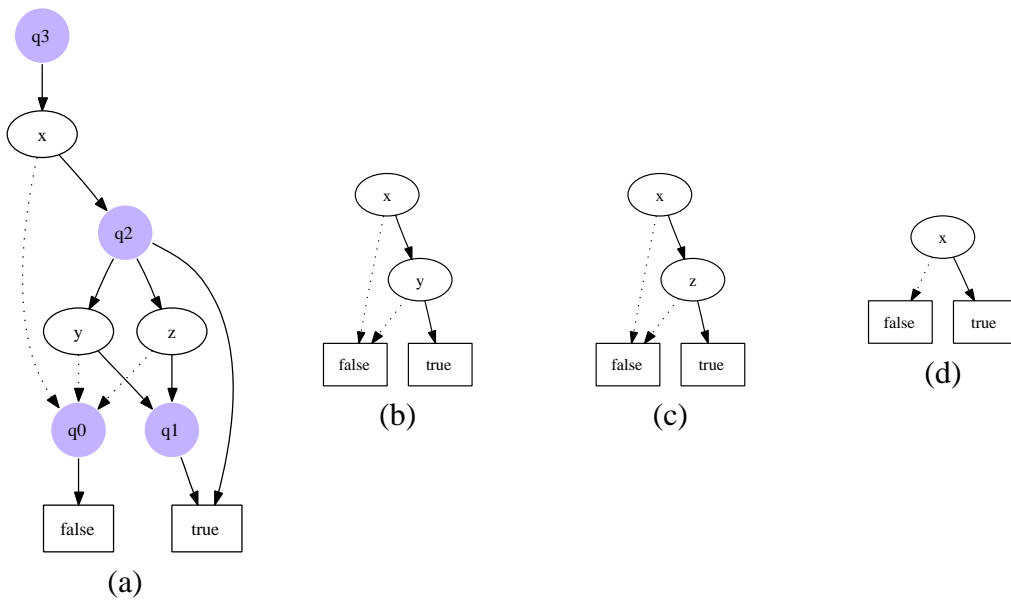


Figure 2.1: Example SBDD: (a) gives a SBDD whose elements are the BDDs (b), (c) and (d)

2.2 Semantics of SBDDs

In this section we define the interpretation function that yields for a given SBDD s the set of Boolean functions represented by s .

Definition 2 An evaluation function is a function $id : \mathcal{V} \rightarrow \{true, false\}$ mapping Boolean values to variables.

Definition 3 Semantics of SBDD

Given a SBDD $S = (\mathcal{V}, \mathcal{S}, \mathcal{D}, \phi, low, high, succs)$ we define $F(S) = F(\phi)$ as follows:

- for terminal nodes 0, 1

$$F(0) = \{false\} \qquad F(1) = \{true\}$$

- for decision node $d \in \mathcal{D}$:

$$F(d) = \left\{ \begin{array}{l} \{(var(d) \wedge f_0) \vee (\neg var(d) \wedge f_1) \mid \\ f_0 \in F(high(d)) \text{ and } f_1 \in F(low(d))\} \end{array} \right.$$

- for structural node $s \in \mathcal{S}$

$$F(s) = \bigcup_{d \in \text{succs}(s)} F(d)$$

Definition 4 *Evaluation of SBDD*

Given a SBDD $S = (\mathcal{V}, \mathcal{S}, \mathcal{D}, \phi, \text{low}, \text{high}, \text{succs})$ and an evaluation function id we define the evaluation of the SBDD S i.e. $F_{id}(S) = F_{id}(\phi)$ as follows:

- for terminal nodes 0, 1

$$F_{id}(0) = \{false\}$$

$$F_{id}(1) = \{true\}$$

- for decision node $d \in \mathcal{D}$:

$$F_{id}(d) = \begin{cases} \text{if } id(\text{var}(d)) \text{ then } F_{id}(\text{low}(d)) \text{ else } F_{id}(\text{high}(d)) \end{cases}$$

- for structural node $s \in \mathcal{S}$

$$F_{id}(s) = \bigcup_{d \in \text{succs}(s)} F(d)$$

Definition 5 *An SBDD is ordered (OSBDD) if on all paths through the graph all labellings with variables respect a linear order $v_1 < v_2 < \dots < v_n$. An OSBDD is reduced if all decision nodes are different, there are no redundant test and the sets of successor nodes of the structural nodes are pairwise disjoint.*

Note: In contrast to the case of *BDDs*, reduced and ordered SBDDs are not a canonical representation by itself.

2.3 SBDD: Data Structure or Tree-Automaton?

A SBDD S , as it has been defined before, is equivalent to a *Tree-Automaton* A such that the language accepted by A is exactly the set of Boolean formulas represented by S . We will use this fact for building a canonical form of S since it is well known how this is done for *Tree-automaton*. So, this section will first give a brief introduction into *Tree-Automaton* and will secondly show the isomorphism between SBDDs and tree-automaton.

Definition 6 A finite **Binary Tree Automaton** is a tuple $A = (Q, N, R, N_{fin})$, where Q is an alphabet with a set of constant symbols Q_c and a set of binary symbols Q_b , a set of nodes N a set of rules R and a set of final nodes N_{fin} . A rule R is defined

$$r ::= \begin{array}{l} (Q_c, N) \quad \text{rule with constant symbol} \\ | \quad (Q_b, N, N, N) \quad \text{rule with binary symbol} \end{array}$$

The language of a binary tree automaton is a binary tree that is defined as follows:

Definition 7 A **Binary Tree** B over alphabet Q is

$$B ::= \begin{array}{l} Q_c \quad \text{leaf} \\ | \quad (Q_b, B * B) \quad \text{composition} \end{array}$$

Example 1 Consider alphabet $Q = Q_c \cup Q_b$ with $Q_c = \{nil, a\}$ and $Q_b = \{cons\}$. Here the symbols nil and a are constants and $cons$ is a binary symbol. The expression $cons(a, cons(a, nil))$ is a binary tree.

A run on an automaton starts at the leaves of input, which is a binary tree and moves upwards, associating each subtree with a state.

Definition 8 A **run tree** T is a binary tree labeled with nodes of the automaton.

$$T ::= \begin{array}{l} (N, Q_c) \quad \text{leaf} \\ | \quad (N, (Q_b, T * T)) \quad \text{composition} \end{array}$$

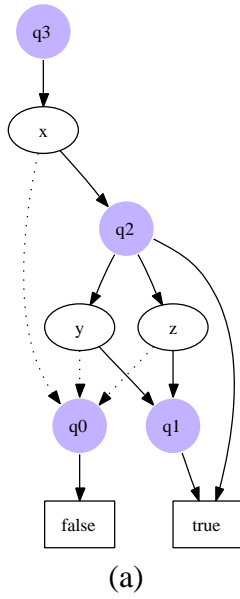
Definition 9 Given a binary tree B , a run Tree T is a **Run** of B in a finite binary tree automaton $A = (Q, N, R, N_{fin})$, if one of the following holds:

$$\begin{array}{l} T = (m, l) \quad \text{and } B = l \quad \text{and } (l, m) \in R, \text{ or} \\ T = (m, (l, (\delta_1, \delta_2))) \quad \text{and } B = (l, B_1 * B_2) \quad \text{and } (l, \delta_1, \delta_2, m) \in R \\ \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{and } (\delta_1, L_1) \text{ is a run of } B_1 \text{ in } A \\ \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{and } (\delta_2, L_2) \text{ is a run of } B_2 \text{ in } A \end{array}$$

Definition 10 *Language*

A binary tree B is a model of a binary tree automaton A if there is an accepting run of B in A . The set of all models of A is called the language of A i.e. $\mathcal{L}(A)$

For building a tree-automaton from an SBDD the following definition gives an isomorphism that translates an SBDD into an equivalent *Binary Tree automaton*.



- $R = \{ \text{true} \rightarrow q_2, \text{true} \rightarrow q_1, \text{false} \rightarrow q_0, y(q_0, q_1) \rightarrow q_2, z(q_0, q_1) \rightarrow q_2, y(q_0, q_2) \rightarrow q_3 \}$

- $N = \{q_0, q_1, q_2, q_3\}$
- $N_{fin} = \{q_3\}$
- $Q_c = \{\text{true}, \text{false}\}$
- $Q_b = \{x, y, z\}$

(b)

Figure 2.2: A SBDD (a) and its equivalent binary tree automaton (b): structural nodes (cycles) in (a) are equivalent to the states of the automaton, decision nodes and terminal nodes correspond to Q_b and Q_c respectively and the arrows give the set of rules R

Definition 11 For a SBDD $D = (\mathcal{V}, \mathcal{S}, \mathcal{D}, \phi, low, high, succs)$ the associated Binary Tree Automaton $A(D) = (Q, N, R, N_{fin})$ is defined as

- $Q_c = \{0, 1\}$
- $Q_b = \mathcal{V}$
- $N = \{m_s | s \in \mathcal{S}\}$
- $R = \{(var(n), m_{low(n)}, m_{high(n)}), m_s | n \in succ(s) \cap \mathcal{D}, s \in \mathcal{S} \cup \{(0, m_s) | 0 \in succ(s), s \in \mathcal{S}\} \cup \{(1, m_s) | 1 \in succ(s), s \in \mathcal{S}\}\}$
- $N_{fin} = \{m_s | s \in \Phi\}$

The set of $Q_c = \{0, 1\}$ of constant symbols consists of the two terminal symbols of the SBDD, D (We use 0 for *false* and 1 for *true*). The set of binary symbols is the set of variables \mathcal{V} in the SBDD. Structural nodes are mapped to states of the *Tree-Automaton* and the decision nodes become rules. Intuitively, we have in an SBDD a decision node d with $d \in succs(s)$, $high(d) = q$ and $low(d) = q'$. The way of representing this in the automaton is: if the automaton is in the state q and q' and it reads the binary

symbol $var(d)$ then we go to the state s . Analogously for the terminal nodes, if it reads a constant symbol c then it goes to the associated state which is the predecessor of c .

The next definition shows the inverse: How we construct a SBDD from a given *Binary Tree Automaton*.

Definition 12 For each Binary Tree Automaton $A = (Q_b \cup \{0, 1\}, N, R, N_{fin})$ the associated SBDD $D(A) = (\mathcal{V}, \mathcal{S}, \mathcal{D}, \phi, low, high, succs)$ is defined as follows:

- $\mathcal{V} = Q_b$
- $\mathcal{D} = \{d_r | r = (l, \delta_1, \delta_2, m) \in R\}$
- $\mathcal{S} = \{s_m | m \in N\}$
- $var(d_{(l, \delta_1, \delta_2, m)}) = l$
- $low(d_{(l, \delta_1, \delta_2, m)}) = s_{\delta_1}$
- $high(d_{(l, \delta_1, \delta_2, m)}) = s_{\delta_2}$
- $succ(s_m) = \{d_r | (l, \delta_1, \delta_2, m) \in R, \delta_1 \in N, \delta_2 \in N\} \cup \{t \in \{0, 1\} | (t, m) \in R\}$
- $\Phi = \{s_m | m \in N_{fin}\}$

The set of variables \mathcal{V} is exactly the set of binary symbols and the set of structural nodes \mathcal{S} contains for each state m of the automaton exactly one corresponding node s_m . For each rule in R we have exactly one decision node in \mathcal{D} . The functions $high$ and low are defined as expected, mapping each decision node to a structural node corresponding to the left or right state in a binary rule, respectively. The function $succs$ assigns to each structural node s_m a set of decision nodes that correspond to the rules which have m as a destination state.

Now we can translate a SBDD into an equivalent canonical form because in [CDG⁺97] it is shown how a *Tree-automaton* can be determinized and how redundant tests could be removed; see also 3.3.3. Thus we can now use the *Myhill-Nerode Theorem* [Koz92] and [CDG⁺97] to minimize a deterministic *Tree-automaton* without any redundant tests. The Theorem also says that the minimized automaton is the smallest one with exactly these properties; this implies that it yields a canonical representation of a SBDD.

We have seen that SBDDs and *Binary Tree Automaton* are indeed isomorphic to each other. Hence, we can use the *Myhill-Nerode Theorem* so that we get a canonical representative of an SBDD.

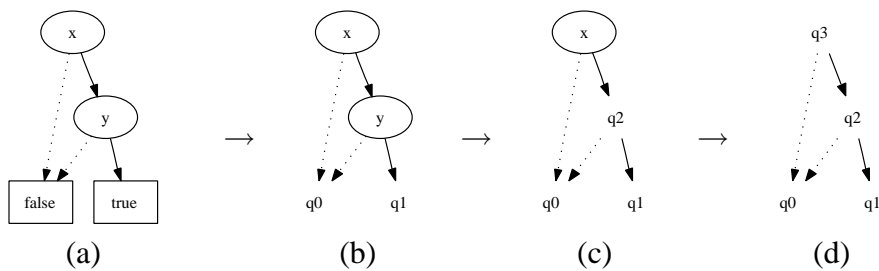


Figure 2.3: The run of the automaton in 2.2 (b) on a BDD that is in its language. (a) Considering the transition relation of 2.2(b) $\text{true} \rightarrow q1$ and $\text{false} \rightarrow q0$ (b) $y(q0, q1) \rightarrow q2$ (c) $x(q0, q2) \rightarrow q3$. $q3$ is the accepting state. Hence, (a) is accepted by 2.2

2.4 Summary

In this chapter we have seen the definition of SBDDs as well as their interpretation. Furthermore, we have shown the equivalence between SBDDs and *Binary Tree Automaton*. We are now ready to build algorithms performing Boolean operations and set operations based on SBDDs as well as building a unique and minimal representation. Now, we can tap the full potential of the SBDD structure.

Chapter 3

Operations on SBDDs

In this chapter we will introduce the algorithms of SBDDs implementing the operations needed on sets of Boolean functions; this includes set operations, Boolean operations and operations for computing a unique representation of the SBDD data structure.

The first section deals with the basic set operations: UNION and INTERSECTION.

In the next section we will show how one can implement the Boolean operations OR and NOT. Because of the law of *De Morgan* it is sufficient to consider only these two operations. Those operations are for the decision nodes similar to them for ordinary BDDs. However, the structural nodes have to be treated differently in order to respect the non-determinism.

Since SBDDs are not a canonical representation of a set of Boolean functions by itself, we have to investigate a method for building a canonical form. We will use the *Myhill-Nerode-Theorem* [Koz92] on the isomorphic *Tree-Automaton* for this purpose. (see section 2.3).

All the operations presented in this chapter assume that there exist the functions `MAKEDNODE` and `MAKESNODE` which build the interface between the memory management and the operations on SBDDs. These operations create a new decision node (`MAKEDNODE`) and a new structural node (`MAKESNODE`) respectively. They ensure that the memory requirements are met (for example, that every node is unique). We introduce them in chapter 4.

3.1 Set Operations

This section presents the implementation of the basic set operations UNION and INTERSECTION.

3.1.1 UNION

For two sets of Boolean functions Φ_1 and Φ_2 represented as SBDDs, the union of these sets $\Phi_1 \cup \Phi_2$ can be built easily; we only need to build the union of the successors of the root nodes Φ_1 and Φ_2 , respectively. Let ϕ_1 be the root node of Φ_1 and ϕ_2 the root node of Φ_2 . Then $\Phi_1 \cup \Phi_2$ is $\text{succ}(\phi_1) \cup \text{succ}(\phi_2)$.

3.1.2 INTERSECTION

The Intersection procedure follows a cross-product construction for reduced *Tree Automaton*, in particular they don't contain any redundant tests. So, we assume that Φ_1 and Φ_2 are reduced. In 2.2 we have seen that for every SBDD S there exists an equivalent *Binary Tree Automaton* A . Thus, we can use the cross-product construction that builds the intersection of the language of two reduced *Tree Automaton*. That is exactly what we want to have. The words of the language of the *Binary Tree Automaton* A equivalent to the SBDD S are the elements of the set of Boolean functions represented by S . Then Figure 3.1.2 gives the algorithm that reveals the intersection of the two sets Φ_1 and Φ_2 .

```

SET-INTERSECTION(  $\phi_1, \phi_2$  )
1:  $S = \text{succ}(\phi_1) \cap \text{succ}(\phi_2)$ 
2: for all  $p \in \text{succ}(\phi_1), q \in \text{succ}(\phi_2)$ , with  $\text{var}(p) = \text{var}(q), p \neq q$  do
3:    $l = \text{SET-INTERSECTION}(\text{low}(p), \text{low}(q))$ 
4:    $h = \text{SET-INTERSECTION}(\text{high}(p), \text{high}(q))$ 
5:   if  $\text{succ}(l) \neq \emptyset \wedge \text{succ}(h) \neq \emptyset$  then
6:      $d = \text{MAKEDNODE}(\text{var}(p), l, h)$ 
7:      $S = S \cup \{d\}$ 
8:   end if
9: end for
10: return  $\text{MAKESNODE}(S)$ 

```

Figure 3.1: intersection of SBDDs

Now we have seen how the basic set operations (union and intersection) can be implemented. In the case of intersection we used the fact that there exists the isomorphism between SBDD and *Tree Automaton*. (2.2)

3.2 Boolean Operations

In this section we will show the implementation of the basic *Boolean operations* OR and NOT. All the other Boolean operations can be expressed with the help of these (De

Morgan).

In [Fin] it is shown that all *Boolean Operations* can be implemented with the **If-then-else** operator. Given three sets of Boolean functions F, G, H **if-then-else** is defined as

$$\text{if } F \text{ then } G \text{ else } H := \{ \text{if } f \text{ then } g \text{ else } h \mid f \in F, g \in G, h \in H \}$$

Since OR is a special case of the **if-then-else** procedure and we do not need to implement the whole **if-then-else** operation; we only implement OR.

Negating an SBDD can be done in linear time by going down the graph structure of the SBDD, swapping the terminal nodes and rebuilding the new graph bottom-up.

These operations are sufficient for performing all other Boolean operations.

3.2.1 OR

As mentioned before, the implementation of the OR procedure is actually a special case of the **if-then-else** procedure, namely "OR(A, B) = if A then B else $\{false\}$ ". Thus we simply modify the **if-then-else** procedure in order to obtain the functionality of OR. The idea is to decompose the structure of the SBDD into a BDD and then use the well-known procedure for BDDs. While processing the OR procedure on two SBDDs we only have to decompose a subgraph if it comes into the scope of a subgraph of the other SBDD which has a smaller variable at the root node. In [Fin] is an algorithm that, in this situation, explicitly flattens a SBDD into a set of SBDD that each represent a singleton set. We actually do not need to explicitly flatten the graph because this is done already by recursion. The recursion itself splits a set of Boolean functions into singleton sets of Boolean functions whenever it needs to go down into recursion.

As a result of the fact that we decompose the compressed graph structure, OR will not produce a minimized SBDD even if the input is in minimized form.

```

OR( $m, n$ )
1:  $S = \emptyset$ 
2: for all  $p \in succ(m), q \in succ(n)$  do
3:    $S = \text{UNION}(S, \text{ORDNODE}(p, q))$ 
4: end for
5: return  $\text{MAKESNODE}(S)$ 

```

Everytime a subgraph comes into scope of a smaller variable OR decomposes a given graph into a set of graphs representing singleton sets (line 3). It recursively calls ORDNODE which is almost the procedure used for BDDs.

ORDNODE (m, n)

```

1: if  $m = 0$  ||  $n = 0$  then
2:   return false
3: else if  $m = 1$  then
4:   return  $n$ 
5: else if  $n = 1$  then
6:   return  $m$ 
7: else if  $m = n$  then
8:   return  $m$ 
9: end if
10: if  $var(m) = var(n)$  then
11:    $s = \text{MAKEDNODE}(var(m), \text{OR}(high(m), high(n)), \text{OR}(low(m), low(n)))$ 
12:   return  $\text{MAKESNODE}(\{s\})$ 
13: else if  $var(m) < var(n)$  then
14:    $s = \text{MAKEDNODE}(var(m), \text{OR}(high(m), n), \text{OR}(low(m), n))$ 
15:   return  $\text{MAKESNODE}(\{s\})$ 
16: else if  $var(m) > var(n)$  then
17:    $s = \text{MAKEDNODE}(var(m), \text{OR}(m, high(n)), \text{OR}(m, low(n)))$ 
18:   return  $\text{MAKESNODE}(\{s\})$ 
19: end if

```

ORDNODE has to respect the strict variable order and gives the result of performing the OR procedure on two decision nodes. For practical reasons, the result is a structural node (SBDD) that has only one successor - namely the expected decision node.

3.2.2 NOT

For negating a SBDD we simply have to go down the graph to the terminal nodes, invert them and rebuild the graph. By recursion we have the structure of the initial graph on the function stack such that we can easily build the negated SBDD. Therefore, NOT is in linear time.

3.2.3 RESTRICT

The RESTRICT operation returns for a given SBDD that SBDD which has one variable replaced by a constant (*true* or *false*).

RESTRICTDNODE(d, v, b)

```

if  $var(d) = v$  then
  if  $b = 0$  then
    return  $low(d)$ 
  end if
  if  $b = 1$  then
    return  $high(d)$ 
  end if
else
   $l = \text{RESTRICT}(low(d), v, b)$ 
   $h = \text{RESTRICT}(high(d), v, b)$ 
  return  $\text{MAKEDNODE}(var(d), l, h)$ 
end if

```

RESTRICT(m, v, b)

```

return  $\text{MAKESNODE}(\{\text{RESTRICTDNODE}(p, v, b) \mid p \in succ(m)\})$ 

```

We can use the RESTRICT operator in order to define other operations:

- $\text{EXISTS}(s, v) = \text{OR}(\text{RESTRICT}(s, v, 0), \text{RESTRICT}(s, v, 1))$
- $\text{FORALL}(s, v) = \text{AND}(\text{RESTRICT}(s, v, 0), \text{RESTRICT}(s, v, 1))$
- $\text{COMPOSITION}(f, g, v) =$
 $\text{OR}(\text{AND}(g, \text{RESTRICT}(s, v, 1)), \text{AND}(\text{NOT}(g), \text{RESTRICT}(s, v, 0)))$
 which defines the function $f|_{v=g}$.
- SUBST is a special case of COMPOSITION where g is the identity function.

3.2.4 FILTER

The algorithm FILTER computes for a set of Boolean functions A , given as a SBDD, and a Boolean function b , given as singleton SBDD, the set of Boolean functions such that for all $a \in A$ it holds that $a \Rightarrow b$, i.e. $\{a \in A \mid a \Rightarrow b\}$

FILTER(A, b) checks for every successor d of A if it implies b . This is exactly the case when b is structurally contained in d . Thus, FILTERDNODE(n, b) checks whether b is structurally contained in d . Since d could be representing a set, the algorithm returns

$\text{FILTER}(A, b)$

Require: A SBDD A and a singleton SBDD (BDD) b .

Ensure: $\text{FILTER}(A, b) = \{a \in A \mid a \Rightarrow b\}$.

```

1: for all  $d$  such that  $d \in \text{succs}(A)$  do
2:    $s = \text{FILTERDNODE}(d, b)$ 
3:   if  $s \neq \emptyset$  then
4:      $S = S \cup \{s\}$ 
5:   end if
6: end for
7: return  $\text{MAKESNODE}(S)$ 

```

only those elements of d that imply b . This is done by implicitly decomposing the graph like in the OR procedure. We have a strict order on the variables. From there, we also have an order on the decision nodes. Hence, we have to traverse the graphs of d and b simultaneously for checking the implication property. Due to this order we know at each position in the graphs how to proceed. Thus, the running time is linear in the number of nodes and edges. $\mathcal{O}(|\mathcal{D}| + |\mathcal{S}| + \#edges)$

$\text{FILTERDNODE}(d, b)$

Require: DNodes d and b .

```

1: if  $(b = 0 \vee b = 1 \vee d = 0 \vee d = 1)$  then
2:   if  $(b = d)$  then
3:     return  $d$ 
4:   else if  $(d \neq 0 \vee d \neq 1)$  then
5:     return  $d$ 
6:   else
7:     return  $\emptyset$ 
8:   end if
9: else if  $(\text{var}(d) \neq \text{var}(b))$  then
10:   $\text{FILTERVARNOTEQUAL}(b, d)$ 
11: else
12:   $\text{FILTERVAREQUAL}(b, d)$ 
13: end if

```

$\text{FILTERDNODE}(d, b)$ returns for Decision nodes d and b the decision node d' such that $d' \Rightarrow b$. d may contain structural nodes with more than one successor, i.e. d is representing a set. So, d' is the subset of d such that b follows from every element in d' . In the lines 1-8 we check the trivial cases, i.e. d or b is a terminal node. If the variables represented by d or b respectively are not equal, FILTERVARNOTEQUAL will be called. Otherwise, if the variables are equal we call FILTERVAREQUAL , which deals with the situation in which the variables assigned to b and d are equal.

```

FILTERVARNOTEQUAL( $b, d$ )
1: if ( $var(d) < var(b)$ ) then
2:   if  $low(d) = 0$  then
3:      $h = \text{FILTER}(high(d), b)$ 
4:     if  $h = \emptyset$  then
5:       return  $\emptyset$ 
6:     else
7:       return  $\text{MAKEDNODE}(var(d), h, 0)$ 
8:     end if
9:   else if  $high(d) = 0$  then
10:     $l = \text{FILTER}(low(d), b)$ 
11:    if  $l = \emptyset$  then
12:      return  $\emptyset$ 
13:    else
14:      return  $\text{MAKEDNODE}(var(d), 0, l)$ 
15:    end if
16:   else
17:      $l = \text{FILTER}(low(d), b)$ 
18:      $h = \text{FILTER}(high(d), b)$ 
19:     if ( $l = \emptyset \vee h = \emptyset$ ) then
20:       return  $\emptyset$ 
21:     else
22:       return  $\text{MAKEDNODE}(var(d), h, l)$ 
23:     end if
24:   end if
25: else
26:   if  $low(b) = 1$  then
27:     return  $\text{FILTERDNODE}(d, high(b))$ 
28:   else if  $high(b) = 1$  then
29:     return  $\text{FILTERDNODE}(d, low(b))$ 
30:   else
31:      $l = \text{FILTERDNODE}(d, low(b))$ 
32:      $h = \text{FILTERDNODE}(d, high(b))$ 
33:     if  $l = \emptyset \vee h = \emptyset$  then
34:       return  $\emptyset$ 
35:     else
36:       return  $\text{MAKEDNODE}(var(d), h, l)$ 
37:     end if
38:   end if
39: end if

```

For $\text{FILTERVARNOTEQUAL}(b, d)$ assume $var(b) \neq var(d)$. If $var(d) < var(b)$ there are three possible situations in which b follows from a subset of d . The first possibility

is $low(d) = 0$ i.e. $low(d)$ is false. In this case it holds: if $d \Rightarrow b$ then $high(d) \Rightarrow b$ ($false \Rightarrow b$ is trivially true). Hence, we have to check whether $high(d) \Rightarrow b$ recursively. The case $high(d) = 0$ is treated analogously. For the remaining case $low(d) \neq 0 \wedge high(d) \neq 0$ we have the situation that d represents the logical operation \vee . $d \Rightarrow b$ if and only if $low(d) \Rightarrow b \wedge high(d) \Rightarrow b$.

If $var(d) > var(b)$, meaning d does not mention $var(b)$, we also will have three cases. If $low(b) = 1$ and $d \Rightarrow b$ then either $var(b)$ is *false*, considering a particular evaluation function, or $high(b)$ follows from d . The case of $high(b) = 1$ is analogous to the case of $low(b) = 1$. In the remaining case we have to check whether the successors of b both follow from d because we do not know whether the evaluation function evaluates $var(b)$ to *true* or to *false*. However, if both successors follow from d we know regardless of which value $var(b)$ is assigned to: if d holds then b also holds

FILTERVAREQUAL(d, b)

```

1: if  $low(d) = 0$  then
2:    $h = \text{FILTER}(high(d), high(b))$ 
3:   if  $h = \emptyset$  then
4:     return  $\emptyset$ 
5:   end if
6:   return  $\text{MAKEDNODE}(var(d), h, 0)$ 
7: else if  $high(d) = 0$  then
8:    $l = \text{FILTER}(low(d), low(b))$ 
9:   if  $l = \emptyset$  then
10:    return  $\emptyset$ 
11:  end if
12:  return  $\text{MAKEDNODE}(var(d), 0, l)$ 
13: else
14:    $h = \text{FILTER}(high(d), high(b))$ 
15:    $l = \text{FILTER}(low(d), low(b))$ 
16:   if  $h = \emptyset \vee l = \emptyset$  then
17:     return  $\emptyset$ 
18:   end if
19:   return  $\text{MAKEDNODE}(var(d), h, l)$ 
20: end if

```

FILTERVAREQUAL(b, d) assumes $var(b) = var(d)$. From $low(d) = 0$ it follows that if d is *true* with respect to the variable assignment than $var(d)$ is assigned to *true*. Thus, we will consider the high branch of d . So, we have to check if $high(d)$ implies $high(b)$. If $high(d) = 0$ a similar argumentation holds. If neither holds, $high(d) \Rightarrow high(b)$ and $low(d) \Rightarrow low(b)$ must be valid. If this is satisfied we have $d \Rightarrow b$.

3.3 Building canonical form

A SBDD respecting the variable order is not in canonical form by itself. This means there are many representation for a particular set of Boolean functions. Fortunately, SBDDs can be interpreted as *Binary Tree Automata* that have a normal form.

All the operations on SBDD respect the variable order. However, some of them do not return an SBDD in normal form even if the input is. For getting a high compression rate it is desirable to transform a SBDD in its unique minimal form.

First of all, we can construct from every *Tree Automaton* an equivalent *Deterministic Tree Automaton*. Secondly, we can remove redundant tests. These are nodes whose *low* and *high* pointers both show the same successor note. Thus the note (or state) can be removed because it is unnecessary.

Then the *Myhill-Nerode-Theorem* defines a minimal representation of a *Deterministic Reduced Tree Automaton*. Thus we can obtain a canonical representation of a SBDD by processing the following procedure

NORMALIZE(s)

- 1: $s = \text{determinize}(s)$
- 2: $s = \text{remove} - \text{redundant} - \text{tests}(s)$
- 3: $s = \text{determinize}(s)$
- 4: $s = \text{Minimize}(s)$
- 5: **return** s

We have to call DETERMINIZE twice because the removal of redundant tests may produce a *Nondeterministic Tree Automaton*.

3.3.1 DETERMINIZE

DETERMINIZE yields for a SBDD an equivalent deterministic SBDD. The procedure uses the standard subset construction for tree automaton. The structural nodes (states in the automaton) of the resulting SBDD S' are sets of structural nodes of the old SBDD S . The function $\gamma : S' \rightarrow 2^S$ gives this relation.

DETERMINIZE(s)

Require: SBDD s

```

1:  $n'_0 = \text{MAKESNODE}(\{0\})$ 
2:  $n'_1 = \text{MAKESNODE}(\{1\})$ 
3:  $N = \{n'_0, n'_1\}$ 
4:  $\gamma(n'_0) = \{m \in S \mid 0 \in \text{succ}(m)\}$ 
5:  $\gamma(n'_1) = \{m \in S \mid 1 \in \text{succ}(m)\}$ 
6: for all  $v \in V$  in decreasing order do
7:    $N' = N$ 
8:   for all  $p, q \in N'$  do
9:      $S = \{\text{MAKESNODE}(v, p, q) \mid \exists r \in D. \text{var}(r) = v, \text{low}(r) \in \gamma(p),$ 
        $\text{high}(q) \in \gamma(q)\}$ 
10:     $G = \{s \in S' \mid \exists r \in \text{succ}(s). \text{var}(r) = v, \text{low}(r) \in \gamma(p), \text{high}(r) \in$ 
        $\gamma(q)\}$ 
11:    if  $S \neq \emptyset$  then
12:       $n = \text{MAKESNODE}(S)$ 
13:       $N = N \cup \{n\}$ 
14:       $\gamma(n) = G$ 
15:    end if
16:  end for
17: end for
18:  $s' = \text{MAKESNODE}(\{s' \in S' \mid s \in \gamma(s')\})$ 
19: return  $s'$ 

```

3.3.2 ELIMINATE-REDUNDANT-TESTS

The algorithm for removing redundant tests in a SBDD works on the graph structure and eliminates nodes that are obvious unnecessary, i.e. they can be eliminated without changing the represented set of Boolean functions.

If we have a decision node d in a SBDD such that $\text{low}(d) = \text{high}(d)$ then it is not sufficient to simply remove d because $\text{high}(d)$ and $\text{low}(d)$ may present more than one function. We solve this problem by using the auxiliary function FLAT every time we have the situation that $\text{low}(d) = \text{high}(d)$. FLAT itself returns for a set S of Boolean functions a set of singleton sets F such that $s \in S \Leftrightarrow \{s\} \in F$.

REMOVE-REDUNDANT-TESTS-DNODE

Require: Decision Node d

```

1: if  $d \in \{0, 1\}$  then
2:   return  $\{d\}$ 
3: else
4:   if  $low(d) = high(d)$  then
5:     for all  $n \in succ(low(d))$  do
6:        $F = FLAT(n)$ 
7:        $S = S \cup F$ 
8:     for all  $l, h \in F, l \neq h$  do
9:        $l' = MAKESNODE(\{l\})$ 
10:       $h' = MAKESNODE(\{h\})$ 
11:       $S = S \cup \{MAKEDNODE(var(d), l', h')\}$ 
12:    end for
13:    return  $S$ 
14:  end for
15:  else
16:     $l = ELIMINATE-REDUNDANT-TEST(LOW(D))$ 
17:     $h = ELIMINATE-REDUNDANT-TEST(HIGH(D))$ 
18:    return  $MAKEDNODE(var(d), l, h)$ 
19:  end if
20: end if

```

REMOVE-REDUNDANT-TESTS

Require: SBDD s

```

1:  $S = \emptyset$ 
2: for all  $n \in succs(s)$  do
3:    $S = S \cup ELIMINATE-REDUNDANT-TEST-DNODE(n)$ 
4: end for

```

3.3.3 MINIMIZE

Deterministic Tree Automata can be minimized using the *Myhill-Nerode* Theorem presented in [CDG⁺97] and [Koz92]. Since SBDDs can be interpreted as *Tree Automata* and can be transformed into deterministic form, we can use this theorem for building such a procedure. The procedure builds an equivalence relation on the states and rebuilds the automaton on the basis of these equivalence classes. Hence, we get a minimal automaton that is equivalent to the original one. The *Myhill-Nerode* Theorem defines what it means that two states (nodes) are equivalent.

Require: SBDD $s = (\mathcal{V}, \mathcal{D}, \mathcal{S}, \Phi, low, high, succ)$

```

1:  $\approx \leftarrow s, \mathcal{S}$ 
2: repeat
3:   for all  $p, q \in \mathcal{S}$  s.t.  $p \approx q$  do
4:     if  $MinCheck(p,q) \wedge MinCheck(q,p)$  then
5:       return  $p \approx q$ 
6:     end if
7:   end for
8: until  $\approx = \approx'$ 
9: return  $[s]_{\approx}$ 

```

All nodes, but the root node, are considered equivalent in the first round of the **for loop**. Hence, we start with two equivalence classes: the root node goes into the first and all the others go into the second one. Then we refine this equivalence relation until we reach a fixpoint. During the rounds the auxiliary procedure MINCHECK checks for two structural nodes p and q whether they can be confused i.e we check if they are equivalent. This procedure checks, corresponding to the *Myhill-Nerode* theorem, what it means that p and q are equivalent. Since \approx is supposed to be an equivalence relation we have to make sure that symmetry is kept which means we have to execute for each p and q $MINCHECK(p, q)$ as well as $MINCHECK(q, p)$.

Therefore, MINCHECK checks for p and q with $p \approx q$ whether for all $s \in \mathcal{S}$ with $d \in succs(s)$ and $p = high(d)$ there exists $s' \in \mathcal{S}$ which is equivalent to s ($s \approx s'$) and $d \in succs(s')$ with $var(d) = var(d')$ and $q = high(d')$ and $low(d) = low(d')$. The case of $p = low(d)$ is similiar. Then we make $p \approx q$ because we know that p and q can be confused to one state by keeping the recognizable language unchanged by the *Myhill-Nerode Theorem*.

MINCHECK(p, q)

$$\forall s \in \mathcal{S}, d \in succ(s) \text{ with } p = high(d)$$

$$\exists s' \in \mathcal{S}, d' \in succ(s').$$

$$s' \approx s, low(d) = low(d'), var(d) = var(d'), q = high(d') \wedge$$

$$\forall s \in \mathcal{S}, d \in succ(s) \text{ with } p = low(d)$$

$$\exists s' \in \mathcal{S}, d' \in succ(s').$$

$$s' \approx s, high(d) = high(d'), var(d) = var(d'), q = low(d')$$

Definition 13 The resulting SBDD $[s]_{\approx} = (\mathcal{V}, \mathcal{D}', \mathcal{S}', \Phi', low', high', succ')$ is defined as follows:

- $\mathcal{S}' = \{[s]_{\approx} \mid s \in \mathcal{S}\}$ where $[s]_{\approx} = \{s' \in \mathcal{S} \mid s' \approx s\}$
- $\mathcal{D}' = \{(var(d), [low(d)]_{\approx}, [high(d)]_{\approx}) \mid d \in \mathcal{D}\}$
- $low((var(d), l, h)) = l$

- $high((var(d), l, h)) = h$
- $succ'([s]_{\approx}) = \{(var(d), [low(d)]_{\approx}, [high(d)]_{\approx}) \mid d \in succ(s)\}$

With the help of MINIMIZE we confuse all states that do not change the expressiveness of the SBDD or the equivalent automaton. So, the computation yields the minimal automaton or equivalently the minimal SBDD.

3.4 Equality check

We have developed a canonical form of an SBDD that allows us to easily check whether two SBDDs are equal. Unfortunately, it is expensive to construct the normal form of an SBDD. If we only want to check whether two SBDDs represent the same set, we could determine whether they are bisimilar; this means their root nodes are equivalent. Thus, we will define an equivalence relation satisfying the required properties.

Considering the structure of SBDDs we can define the following equivalence relation recursively:

Definition 14 *Two SBDDs $(\mathcal{V}_1, \mathcal{S}_1, \mathcal{D}_1, \phi_1, low_1, high_1, succs_1)$ and $(\mathcal{V}_2, \mathcal{S}_2, \mathcal{D}_2, \phi_2, low_2, high_2, succs_2)$ are equivalent, i.e. they represent the same set of Boolean functions if $\phi_1 \approx \phi_2$*

- *Terminal nodes are equivalent if they are equal*
- *Two structural nodes $s_1 \in \mathcal{S}_1$ and $s_2 \in \mathcal{S}_2$ are equivalent if*

$$s_1 = s_2 \vee$$

$$(\forall d_1 \in succs_1(s_1). \exists d_2 \in succs_2(s_2). d_1 \approx d_2 \wedge$$

$$\forall d_2 \in succs_2(s_2). \exists d_1 \in succs_1(s_1). d_2 \approx d_1)$$
- *Two decision nodes $d_1 \in \mathcal{D}_1$ and $d_2 \in \mathcal{D}_2$ are equivalent if*

$$d_1 = d_2 \vee$$

$$(high_1(d_1) \approx high_2(d_2) \wedge$$

$$low_1(d_1) \approx low_2(d_2))$$

This relation is reflexive, symmetric and transitive. Hence, it is indeed an equivalence relation.

Two nodes are in the same equivalence class exactly if they represent the same set of Boolean functions.

3.5 Summary

In this section we have presented the basic operations on sets of Boolean functions such as UNION, INTERSECTION, OR, NOT, RESTRICT and FILTER. Those operations except of INTERSECTION could be implemented close to their semantic meaning on the graph structure like in the BDD case. In actuality, we handle the *Decision Nodes* in the implementation of OR with almost the same procedure that we use for a BDD implementation.

However, when building the canonical representation, we have to treat the SBDD as a *Binary Tree Automaton* in order to apply the algorithm resulting from the *Myhill Nerode Theorem* for constructing minimal *Tree Automata*.

In the last section we define what it means for two SBDDs to be equal even if they are not syntactically equal.

Chapter 4

Implementing SBDDs

In this chapter we will see a pointerless implementation of an SBDD package using a *clean-sweep garbage collector*. There are three kind of concepts that are needed for implementing the pointerless package: the *node arrays*, the *unique tables* and the *garbage collector*. We built a C++ libray using these concepts for providing the functionality of SBDDs.

There are two *node arrays*: one for storing the decision nodes and one for storing the structural nodes. This implies that the nodes have a strict order. Thus every node can be identified by its position in the array.

The package keeps two *unique tables*: one for the decision nodes and one for the structural nodes. The *unique table* in the case of decision nodes is a map that maps a tuple (v, h, l) where v is a variable $h = high(v)$ and $l = low(v)$ to the position in the *node array* where the unique node is stored if there exists already such a node. In the case of structural nodes it is essentially the same. You map a set of decision nodes D to the unique representative s in the *node array* of structural nodes such that $D = succs(s)$. These *unique tables* make it possible to reuse parts of our data structure and locate them in amortized constant time by using hash tables.

Since the operations are expensive and the operations on the same input are performed frequently, we maintain a cache: the *computed table*. For every operation, we first check whether the operation was performed before. If so, then we get the result in constant time; otherwise, we have to compute the result and then add it to the table.

The last important concept is that of the *mark-sweep-update-sweep garbage collector*. First, the garbage collector marks all active nodes. Then it cleans the *node arrays* by filling in the gaps of dead nodes with active nodes. After that, it updates the *unique table* and the references in the nodes because they were likely moved during the cleaning process. Finally, it goes on to destroy the *computed table*.

4.1 Storing Nodes

The main idea is to use an array structure instead of pointers, as introduced for BDD packages by [Jan01]. This enforces that we have a strict order for the SBDD nodes. The advantage of this representation is that we do not need a reference counter for garbage collection. Concerning the theoretical structure of the SBDD there are two different data structures: one for the *decision* nodes and the other for the *structural* nodes. Hence, we have two respective arrays. A node is identified by its array position. As we can distinguish the types of the nodes every node is uniquely identified.

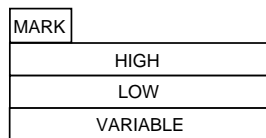


Figure 4.1: Memory Block of Decision Node

A *Decision Node* is similar to a node of a normal BDD. It consists of the integer fields, *High* and *Low*, that stores the array position of its children. These are in this case *structural* nodes. Furthermore, a *decision* node contains a memory block for saving the variable that it represents. This block is an integer value as well. Additionally, we require that a mark flag be used for internal operations. So, the overall memory that is needed for storing a decision node is $3 * \text{sizeof}(\text{int}) + \text{sizeof}(\text{bool})$. The advantage of this representation is that we do not need a reference counting memory block. Figure 4.1 shows the layout of such a memory block.

A *structural* node consists of a set of successor nodes because they can have an arbitrary number of successors. This is realized in the implementation by having a pointer to an array where the positions of the successor nodes in the node array are stored. Because of this, the size of a *structural* node is $n * \text{sizeof}(\text{int}) + \text{sizeof}(\text{bool})$ where n is the number of successors. Again, we have a bit for marking the node.

A node is always inserted at the first free position of the node array. This makes sure that nodes that are later added have higher array positions. As a result the nodes have a strict order.

4.2 Making nodes unique

If a new node has to be allocated we can check with the help of the *Unique Tables* whether there exists already a node with the same properties.

Figure 4.2 presents the algorithm for creating a new decision node (v, h, l) with v as a variable and h and l as successors that are assumed to be unique. We check the *unique*

table for decision nodes *uniqueDtable* whether such a node already exists. If it exists, the unique table will give the position of this node in the node array. This implies that all nodes in the node arrays are different, i.e., they are unique.

MAKEDNODE(v, h, l)

Require: variable v , structural nodes h, l

if if there exists node d with $var(d) = v$, $high(d) = h$ and $low(d) = l$ in *uniqueDtable* **then**

return *uniqueDtable*(d)

else

 create new decision node d s.t. $var(d) = v$, $high(d) = h$ and $low(d) = l$

 add d to *uniqueDtable*

return *uniqueDtable*(d)

end if

Figure 4.2: makeDNode

Figure 4.2 is the algorithm for creating the *structural* node with successor set S . First the algorithm checks again whether there exists a node s such that $succ(s) = S$. If it exists it will be returned; if not, such a node will be created.

MAKESNODE(S)

Require: set S of decision nodes - successor nodes

if if there exists node s with $succ(s) = S$ in *uniqueStable* **then**

return *uniqueStable*(s)

else

 create new structural node s s.t. $succ(s) = S$

 add s to *uniqueStable*

return *uniqueStable*(s)

end if

Figure 4.3: makeSNode

4.3 Computed table

In this section we will see how the *computed table* can be used for saving costly computations that have already been computed and how we can implement it efficiently.

The main idea is that previously computed results do not need to be re-computed. This means we can simply maintain a hash table that maps triples of the form (op, s_1, s_2) to a result node s where $op \in \{\text{OR}, \text{UNION}, \text{INTERSECTION}, \text{RESTRICT}\}$ and s_1, s_2, s are SBDDs. Each time we want to process an operation we first check whether we have already computed the result or whether we must still compute it. The computed result can then be placed into the *computed table*.

Hash tables have amortized constant running time such that the additional effort spent on maintaining the table is negligible, but tests show that there are many results that are frequently re-used and hence have to be re-computed. This additional work can be saved and hence makes the overall computation faster.

After the garbage collection, the *computed table* is no longer valid because the nodes representing a certain SBDD may have been moved or deleted in such a way to make the table invalid.

4.4 Collect garbage

The last important concept of the implementation is the *mark-sweep-update-sweep* garbage collector. Since, during operations, some parts of the node array are no longer used we can remove them in order to create room for new nodes. The package manages a table of nodes that can be accessed by the user. If the garbage collector is started it checks this table to determine which nodes are accessible directly by the user and which nodes are reachable through this nodes and marks them; we call these nodes *active*.

We have seen in section 4.1 that there is a strict order on the nodes. Thus, we move the active nodes to a lower position in the array where another node has become inactive. The gaps of inactive nodes are filled whereas the order of the active nodes is kept. This procedure is called the *sweep phase*. Figure 4.4 illustrates this procedure.

During the sweep phase the nodes are moved in the array; this is why the pointers stored in the nodes have to be updated. This is done in the *update phase*. During the sweep phase the algorithm manages *forward tables* which gives, for every node, its new array position. In the update phase the garbage collector runs through the array and updates the successors of the nodes by using the forward table.

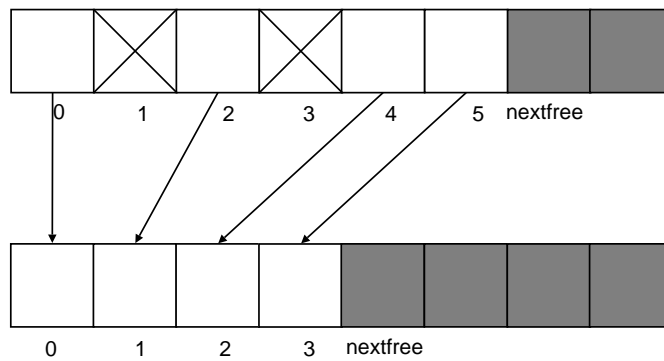


Figure 4.4: sweep phase during garbage collection

Figure 4.5 shows the *mark-sweep-update-sweep* approach of the garbage collector of the SBDD package. The function `next_insert_pos` returns the smallest position of an inactive node in the storage array. *SFORWARD* and *DFORWARD* are the forward tables for the relating storage array. *dnextfree* and *snextfree* show to the next free position that did not contain a node. `update_unique_table` updates the positions of the nodes in the *unique table*.

4.5 Design

For the implementation, it was decided to put the main focus on saving memory. Therefore, we have the disadvantage of slower algorithms. If more memory was spent on additional data structures for the algorithms, we would have better running time.

Consider `MINIMIZE`, for example. We could spend additional memory for maintaining a hash table in a way that enables us to find nodes we are looking for faster. The maintenance of this hash table reduces the effort spent during the `exist` (\exists) operation in the `MINCHECK` routine. This presents a trade-off because it is memory costly to organize all the nodes of the given SBDD.

We analyzed the profiling data received from running instances of the example application which we introduce in chapter 5. These results verify the fact that the `MINIMIZE` procedure is actually the most expensive one. A lot of the running time of the whole *language containment checking* procedure was spent in the procedure that implements the `exists` operation in the `MINCHECK` routine. This means that a lot of time is spent on searching through the nodes. We implemented `MINCHECK` such that we go through all structural nodes and check them to see whether the condition holds. Therefore, we

have twice a quadratic running time for the MINCHECK routine. Additionally, MINIMIZE goes through all the $p, q \in \mathcal{S}$. This is repeated until we reach a fixpoint. Hence, it is expensive to compute a minimized SBDD.

```

1: markphase();
2: for  $i < dnextfree$  do
3:     if ismarked( $i$ ) then
4:          $n = next\_insert\_pos$ ;
5:          $mem[n] = mem[i]$ ;
6:          $DFORWARD(i) = n$ ;
7:     end if
8:      $i++$ 
9: end for
10: for  $i < snextfree$  do
11:     if ismarked( $i$ ) then
12:          $n = next\_insert\_pos$ ;
13:          $mem[n] = mem[i]$ ;
14:          $SFORWARD(i) = n$ 
15:     end if
16:      $i++$ 
17: end for
18: for  $i < dnextfree$  do
19:      $high(i) = SFORWARD(high(i))$ 
20:      $low(i) = SFORWARD(low(i))$ 
21:      $i++$ 
22: end for
23: for  $i < dnextfree$  do
24:     for all  $s \in succ(i)$  do
25:          $S = DFORWARD(s)$ 
26:     end for
27:      $i++$ 
28: end for
29: update_unique_table
30: update(dnextfree)
31: update(snextfree)
32: unmark()

```

Figure 4.5: mark-sweep-update-sweep garbage collector

Chapter 5

Language containment checking

In this chapter we present an application of the SBDD functionality that was previously introduced. We will build a *language inclusion checker* for *Symbolic Labeled Transition Systems*.

We introduce an algorithm for checking whether, for two given *Symbolic Labeled Transition Systems* A and B , the language of A is a subset of the language of B . We use the implementation of the SBDD library that was introduced in chapter 3 and chapter 4. Using this library we can easily implement the *language inclusion checker*.

5.1 Symbolic Labeled Transition System

In this section we will give a definition of *Symbolic Labeled Transition System* which we will use in the next section in order to describe *Transitions Systems* symbolically. The symbolic representation of models was introduced in [McM93].

Definition 15 *We can represent a set of states in a transition system symbolically by encoding the set as a Boolean function over a set of variables \mathcal{V} . This Boolean function can be represented as a BDD or in our case as a SBDD containing exactly this Boolean function.*

Definition 16 *A set of transitions t over a set of states can be represented symbolically by encoding the set of transitions as a Boolean function f over the variables \mathcal{V} and their primed version \mathcal{V}' . \mathcal{V} is the set of variables that encode the source states of a transition and \mathcal{V}' gives the respective destination states of the transition. f is again encoded as BDD or singleton SBDD.*

Definition 17 A Symbolic Labeled Transition is a tuple (a, t)

- $a \in \Sigma$ and Σ is an alphabet
- t is a symbolically represented set of transitions

t is the set of transitions with label a , i.e. if any transition from t is taken one can observe a

Definition 18 A Symbolical Labeled Transition System is a tuple (I, T, \mathcal{V})

- \mathcal{V} set of variables
- symbolic represented set of initial states I with variables in \mathcal{V}
- a set of Symbolic Labeled Transitions T with variables in \mathcal{V}

5.2 Language containment checking

A common problem in software-aided verification is that one has a concrete implementation A of a System and an abstract System B of that system that specifies the behavior of the concrete system. This means that both systems are able to observe the same external events. Thus, if you have two Systems A and B given as *Symbolic Labeled Transition Systems*, you want to verify that the language of A is also accepted by B , i.e. $\mathcal{L}(A) \subset \mathcal{L}(B)$.

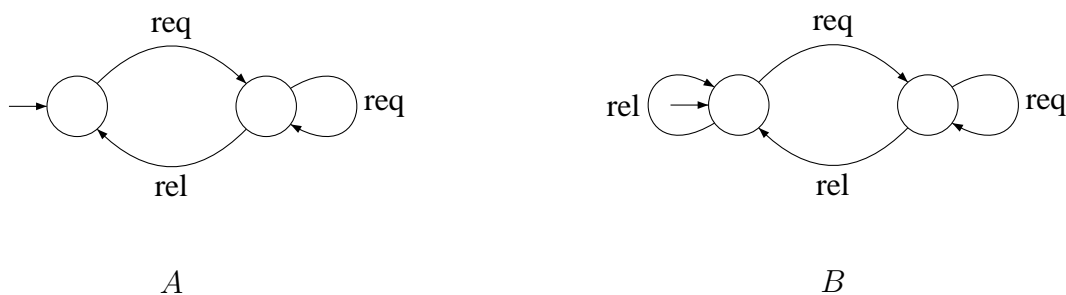


Figure 5.1: example of two systems

Figure 5.1 gives an example of two systems. System A accepts the language $\mathcal{L}(A) = (req^+.rel)^\omega$ whereas B accepts the language $\mathcal{L}(B) = (req|rel)^\omega$. If you consider A as your concrete system and B as your abstract system you can see that the language inclusion holds, i.e. $\mathcal{L}(A) \subset \mathcal{L}(B)$.

Figure 5.2 gives the algorithm performing a language inclusion check on two given *Symbolical Labeled Transitions Systems* $S_1 = (I_1, T_1, \mathcal{V}_1)$ and $S_2 = (I_2, T_2, \mathcal{V}_2)$ over alphabet Σ , $\mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset$ with

- $T_1 = \{(a_1, \alpha_1), (a_2, \alpha_2), \dots, (a_n, \alpha_n)\}$
- $T_2 = \{(a_1, \beta_1), (a_2, \beta_2), \dots, (a_n, \beta_n)\}$
- $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$
- \mathcal{V}' primed version of variables in \mathcal{V}

with $a_i \in \Sigma$, α_i and β_i *symbolically represented set of transitions*, $i \in \{1, \dots, n\}$. To each observable event a_i , there is a corresponding pair of transitions (α_i, β_i)

The algorithm in figure 5.2 consists of two steps. The first step (lines 1-15) computes the set of all simultaneously reachable configurations. The second step checks if every transition that is possible in the concrete system is also possible in the abstract system. If this holds then *language inclusion* holds.

- in lines 1-15 the algorithm computes the set of all simultaneously reachable configurations of both systems. It starts with the initial configuration $(I_1 \wedge I_2)$. Then it computes all the configurations that can be reached from the initial state if both systems take transitions simultaneously. If the computation reaches a fixpoint S then S is the symbolical representation of the set of all simultaneously reachable states.
- in lines 17-21 it computes the set of configurations where α_i is enabled
- in lines 22-25 the predicate *disabled* is computed; this is a SBDD representing a set that contains exactly one Boolean function.
- in line 26 the algorithm computes the set of those configuration from which all transitions in β_i are *disabled*, i.e. the set of configurations such that *disabled* follows from them.
- in lines 27-31 we check whether the set B is the empty set. If $B \neq \emptyset$ language inclusion does not hold because there is a reachable configuration where a transition in α_i can be taken but there is no corresponding transition in β_i that can be taken.

It is easy to implement such a decision procedure efficiently with the help of the *SBDD-Library* because the library provides an efficient data structure for representing sets of Boolean functions and all the required operations.

5.3 Design

The strategy we have implemented for checking the language inclusion property is: we compute the fixpoint from the *initial state* and the *set of transitions* without computing the minimal representative of the intermediate results. Then we build the minimal representative of the fixpoint. After that, we run the second step in the *language inclusion checking* algorithm. Namely, we check, in each transition whether one can take this transition in the *concrete system* but not in the *abstract system*.

```

1:  $S = \text{AND}(I_1, I_2)$  {initialize  $S$ }
2:  $H = \emptyset$ 
3: repeat
4:    $S = S'$ 
5:   for  $i = 0$  to  $n$  do
6:      $H = \text{AND}(S, \text{AND}(\alpha_i, \beta_i))$ 
7:     for  $v \in \mathcal{V}$  do
8:        $H = \text{EXISTS}(H, v)$   $\{\exists v.H\}$ 
9:     end for
10:     $S' = \text{UNION}(S', H)$ 
11:   end for
12:   for  $x' \in \mathcal{V}'$  do
13:      $S' = \text{SUBST}(S', x, x')$   $\{S'[x' := x]\}$ 
14:   end for
15: until  $S = S'$ 
16: for  $i = 1..n$  do
17:    $A = \text{AND}(S, \alpha_i)$ 
18:   for  $v \in \mathcal{V}'_1$  do
19:      $A = \text{EXISTS}(A, v)$ 
20:   end for
21:    $A = \text{REM\_FALSE}(A)$   $\{A - \{false\}\}$ 
22:    $disabled = \text{NOT}(\beta_i)$ 
23:   for  $v \in \mathcal{V}'_2$  do
24:      $disabled = \text{FORALL}(disabled, v)$   $\{\forall v. \neg \beta_i\}$ 
25:   end for
26:    $B = \text{FILTER}(A, disabled)$ 
27:   if  $\text{ISEMPTY}(B)$  then
28:     return language inclusion holds
29:   else
30:     return language inclusion does not hold
31:   end if
32: end for

```

Figure 5.2: algorithm that checks the language inclusion property for two given systems

Chapter 6

Experimental Results

In this chapter we present the results obtained by running the *language containment checker* on certain instances of the *mutual exclusion problem*. It will turn out that the compression rate of sets of Boolean function represented as SBDD is exponentially better than the conventional approach of representing a set of Boolean functions as a set of BDDs. However, another result shows that it is expensive to compute the minimal representative.

We ran the *language inclusion checker* introduced in chapter 5 on certain mutual exclusion problems. For our experiments we had a AMD Athlon XP 2600+ machine with 2GB of RAM installed. This machine was running Debian Linux kernel version 2.6.15-1-k7.

6.1 Dining Philosophers

To analyze the behavior of our implemented SBDD data structure, we run the *language inclusion checker* on an instance of the four dining philosophers. During the fixpoint computation we contrast the number of nodes of the SBDD of the intermediate result to the number of nodes of the equivalent set of BDDs.

Since there were previously no benchmarks for testing implementation of sets of Boolean functions, we present here a common concurrency problem, namely, the dining philosophers problem described for example in [Wik06]. In this problem there exists a round table with four forks and four seats. Each philosopher has exactly one seat and two out of the four forks are assigned to this seat. A philosopher is able to eat only if he has two forks. However, he has to share his forks with his neighbors. So, the question is now how to arrange the philosophers so that every philosopher who wants to eat is eventually able to eat. During the process, it may be the case that the system becomes

stuck. The problem of the dining philosophers is equivalent to a common problem in computer-aided verification, namely to the problem of how to control shared resources in a system. Figure 6.1 shows the implementation of one philosopher. The whole system consists then of four parallel instances of this system, four semaphores representing the forks and a room semaphore that controls the access to the dining room. The additional room semaphore is needed to avoid a situation where the system reaches a deadlock, i.e. that four philosopher are sitting at the table and each of them has allocated only one fork.

We can formulate properties which we are interested in as an *abstract system* and check the property against our implementation of the philosopher problem (*concrete system*) with the help of the *language inclusion checker*. We showed in chapter 5: the property holds for the *concrete system* exactly if the language of the *concrete system* is a subset of the language of the *abstract system* over the same alphabet. In this case we used an *abstract system* which observes the *mutual exclusion* property.

Since we are interested in how much memory we can save by using the SBDD data structure instead of a set of BDD we keep track of the decision nodes that are used in the intermediate results during the computation of the fixpoint S and compares them to the number of nodes in the equivalent set of BDDs.

Figure 6.2 depicts the graph that compares the number of decision nodes in the SBDD (solid line) of the intermediate result to numbers of nodes of the equivalent set of BDDs (dotted line). One can observe: in the case of SBDDs the curve rises slowly whereas in the case of BDDs the curve rises exponentially.

```

1: loop
2:   Think
3:   Wait_Room
4:   Wait_left_fork_i
5:   Wait_Right_fork_i
6:   Eat_spaghetti
7:   Release_left_fork_i
8:   Release_right_fork_i
9:   Release_Room
10: end loop

```

$P[i]::$

Figure 6.1: philosopher

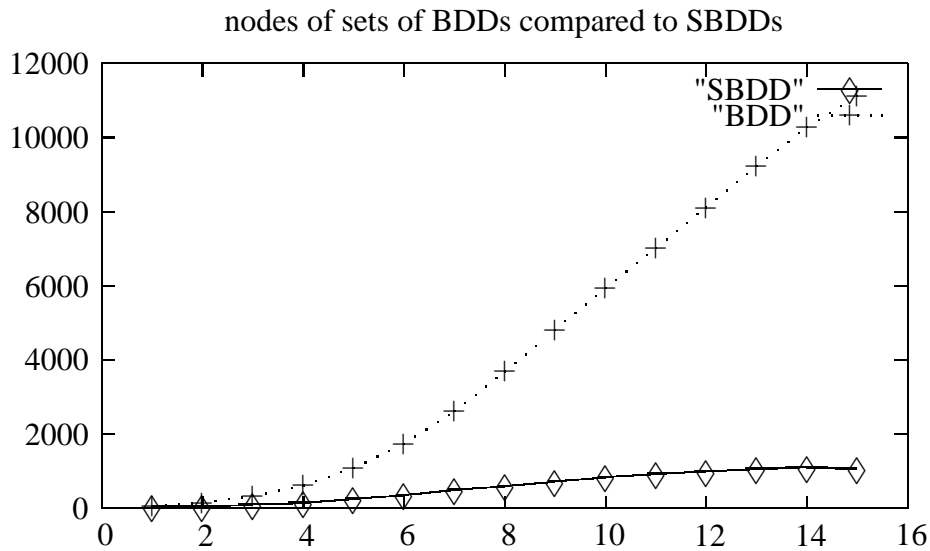


Figure 6.2: decision nodes of SBDDs compared to equivalent set of BDDs

6.2 Mutual exclusion with manager

In this section we present the results obtained by running certain instances of the mutual exclusion problem, *mutex_man_n*, consisting of a manager and n processes. We observe again that the number of decision nodes of the fixpoint represented as minimal SBDD is exponentially smaller than the number of nodes of the equivalent set of BDDs. In this case the result also shows that it is expensive to compute the minimal form from sets of Boolean functions.

An implementation of the *mutual exclusion* problem using a central manager is introduced in [MP95]. There are n processes and a central manager running in parallel. Each of the processes may want to access a memory that they share. It first sends a request to the manager, and the manager responds then which process is allowed to access the variable next. The algorithm is shown in Figure 6.3

We run certain instances of the *mutex_man_n* problem in order to compare the size of the fixpoints in compressed form to sets of BDDs. The *abstract system*, we chose, observes the mutual exclusion behavior in order to check whether the *mutex_man_n* satisfies this property. The language of the abstract system is the language that contains the words such that if one process enters the critical section it must leave this section before another process can enter it.

Table 6.1 shows the results of running instances of *mutex_man_n* of 1, 2, 3, 4 and 5 clients running in parallel. The column are as follows: *SNodes* gives the number of

$A ::$	<pre> loop while $y = 0$ do skip end while $x := y$ while $x \neq 0$ do skip end while end loop </pre>	$\parallel_{j=1}^n P[j] ::$	<pre> loop noncritical while $x \neq j$ do $y := j$ end while critical $x := 0$ end loop </pre>
--------	---	-----------------------------	--

Figure 6.3: `mux_man_n`: The processes $P[j]$ request the central manager A for entering the critical section. The central manager A receives requests in variable y and responds in variable x which process is allowed to enter the critical section.

structural nodes, $Dnodes$ the number of decision nodes, $labels$ gives the size of the alphabet, fp is the time (in seconds) spent on computing the fixpoint, fp_{min} is the time (in seconds) needed to compute the minimal representation of the fixpoint and lc gives the time (in seconds) spent on performing the language inclusion check. The *flat* row below each instance gives the number of nodes of the fixpoint represented as set of BDDs.

Considering Table 6.1 and column fp_{min} we can see that it is indeed expensive to compute the minimal fixpoint. The time, spent on computation, is increasing exponentially with the number of nodes. If we compare the number of decision nodes (DNodes) in each case with the number of nodes of the flattened fixpoint (*flat* row below each instance) we see that there is an increasing distance between them. Figure 6.5 contrasts these two values. As we can see, the number of decision nodes, in the SBDD case, increases almost linearly with the number of client processes running in parallel. On the contrary, in the case of sets of BDDs the number of nodes increases exponentially with the number of processes running. The result is: we can save exponential many nodes.

In this section we have seen that it is expensive to compute the minimal representation of a set of Boolean functions. But we have also seen that the number of nodes saved is exponential.

6.3 Semaphores

In this section we present the results of running a series of the *mutual exclusion* problem using semaphore variables introduced in [MP95]. This is actually the simplest method in order to achieve mutual exclusion.

```

                                loop
                                noncritical
                                request y[i]
                                critical
                                release y[i + M - 1]
                                end loop
||_{j=1}^M P[j] ::

```

Figure 6.4: `sem_n`: Achieves mutual exclusion by using M semaphore variables.

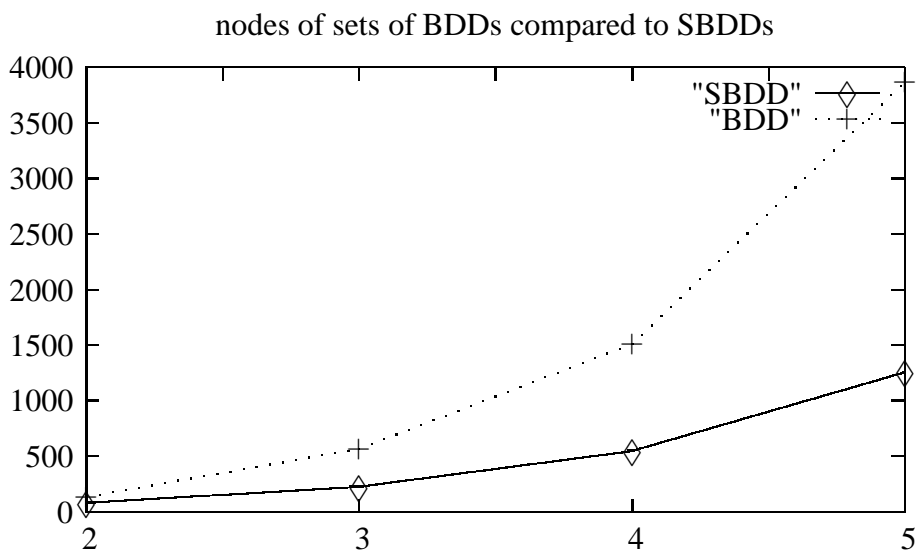
Table 6.2 shows the results of running instances of `sem_n` of 1, 2, 3, 4, 5 and 6 clients running in parallel. The columns are as follows: *SNodes* gives the number of structural nodes, *Dnodes* the number of decision nodes, *labels* gives the size of the alphabet, *fp* is the time (in seconds) spent on computing the fixpoint, *fp_{min}* is the time (in seconds) needed to compute the minimal representation of the fixpoint and *lc* gives the time (in seconds) spent on performing the language inclusion check. The *flat* row below each instance gives the number of nodes of the fixpoint represented as set of BDDs.

While running this series of experiments we can recognize, like in the case of *mutual exclusion with manager*, that it is expensive to compute the minimal representative. Consider Figure 6.6 which contrasts the decision nodes of the minimized SBDD to the nodes of equivalent set of BDDs. We observe a very slow, linear rising curve in the SBDD case. In contrast, in the BDD case, we have again a very fast exponential rising curve.

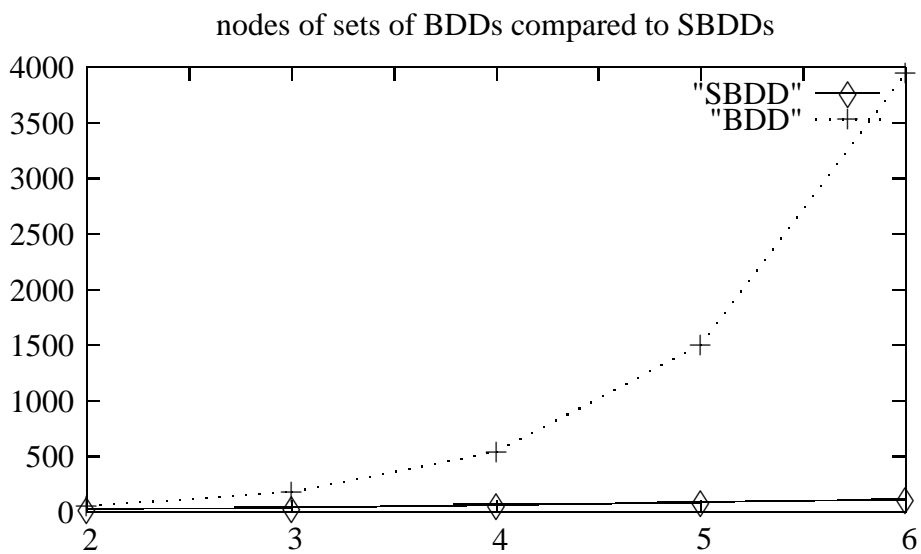
6.4 Summary

In all of the different series of experiments performed we recognized the same behavior. It is expensive to compute the minimal representative of the fixpoint. We also observed that the number of nodes we can save by using this approach instead of sets of BDDs saves exponentially many nodes.

	<i>SNodes</i>	<i>DNodes</i>	<i>labels</i>	<i>vars</i>	<i>fp</i>	<i>fp_{min}</i>	<i>lc</i>
mux_man_2	66	87	9	22	0.47	1.85	3.66
flat	113	138					
mux_man_3	170	233	13	34	7.09	22.32	35.78
flat	479	568					
mux_man_4	390	554	17	36	29.36	552.75	703.24
flat	1244	1513					
mux_man_5	864	1260	21	40	159.98	> 3600	> 3600
flat	3117	3866					

Table 6.1: results of running certain instances of *mux_man_n*Figure 6.5: decision nodes of SBDD compared to those of equivalent set of BDDs at certain instances of the *mux_man_n*

	<i>SNodes</i>	<i>DNodes</i>	<i>labels</i>	<i>vars</i>	<i>fp</i>	<i>fp_{min}</i>	<i>lc</i>
sem_2	21	26	6	16	0.16	0.06	1.39
flat	46	55					
sem_3	33	44	9	22	1.47	0.73	3.07
flat	151	184					
sem_4	51	69	12	30	10.56	20.45	44.81
flat	456	549					
sem_5	67	93	15	36	54.19	499.03	1437.28
flat	1269	1506					
sem_6	85	120	18	42	337.93	> 3600	> 3600
flat	3378	3951					

Table 6.2: results of running certain instances of *sem_n*Figure 6.6: decision nodes of SBDDs compared to those of equivalent set of BDDs at certain instances of *sem_n*

Chapter 7

Summary

In this thesis we presented the SBDD data structure which gives an exponential compression rate for sets of Boolean functions compared to sets of standard BDDs. Since we have to deal with large sets of Boolean functions in the field of computer aided-verification this is very useful in practice.

In the second chapter we gave a formal definition of what a *SBDD* actually is. We gave the syntactical representation and defined the semantical meaning of that. Furthermore, we showed the equivalence between the SBDD data structure and *Binary Tree Automaton*. Hence, every SBDD can be transformed into an equivalent *Binary Tree Automaton* with language that is exactly the set of those BDDs that is equivalent to the SBDD. The other direction holds as well, namely every *Binary Tree Automaton* can be transformed into an equivalent SBDD. This gives the opportunity to use algorithms that work on either SBDD or *Binary Tree Automaton*.

The third chapter is used to present the algorithms which build and manipulate SBDDs. There are *Boolean operations*, *set operations*, and operations for building the *canonical representative*. Unlike BDDs, SBDDs are not necessarily in canonical representation. The *Boolean operations* work directly on the graph structure. The *set operations* and the operations for building the *canonical representative* work on the automaton.

In the fourth chapter we have shown some key concepts for creating a *C++ library* providing the SBDD data structure and all its operations. Therefore, a pointerless approach for implementing this functionality was presented.

Chapter 5 showed an implementation of a *language inclusion checker* for two reactive systems using the algorithms presented in chapter three. Having a functionality, which allows us to build and manipulate sets of Boolean functions, we have seen that it is rather easy to implement the *language inclusion checker*.

The sixth chapter shows some results obtained by checking certain instances of the *mutual exclusion* problem to see if they satisfy the mutual exclusion property. We observed that it is rather expensive to compute the minimal representative of an SBDD. The most important result was that the number of decision nodes of an SBDD was exponentially smaller compared to the nodes of the equivalent set of BDDs.

7.1 Future work

We have seen that on one side, it is expensive to create the compact representation. However, on the other side we can save exponential many nodes. Some of the operations presented in chapter 3 destroy the minimal structure of an SBDD such that it has to be recomputed afterwards. We could change those operations such that they directly minimize the decomposed subtrees while performing the operations; this could possibly eliminate the expensive recomputation of the minimal representative and would also keep the minimal representation.

We also could change the design aspect of saving memory during the computation to a faster approach which would increase the speed of computing the minimal representative.

Bibliography

- [AMS02] F. Aloul, M. Mneimneh, , and K. Sakallah. Zbdd-based backtrack search sat solver. In *International Workshop on Logic Synthesis (IWLS)*, New Orleans, Louisiana, pages 131–136, 2002.
- [And97] H. Reif Andersen. An introduction to binary decision diagrams. Web: <http://www.it.dtu.dk/~hra>, 1997.
- [CDG⁺97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: <http://www.grappa.univ-lille3.fr/tata>, 1997. release October, 1rst 2002.
- [Fin] B. Finkbeiner. SBDDs. Projektvorschlag.
- [Fin01] B. Finkbeiner. Language containment checking with nondeterministic bdds. In Tiziana Margaria and Wang Yi, editors, *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of Lecture Notes in Computer Science, pages 24–38. Springer-Verlag, 2001.
- [Jan01] G. Janssen. Design of a pointerless bdd package, 2001. Submitted to 10th Intl. Workshop on Logic & Synthesis.
- [Koz92] D. Kozen. On the myhill-nerode theorem for trees. In *Bulletin of the European Assosiation for Theoretical Computer Science*. 1992.
- [LSW95] M. Lobbing, O. Schroer, and I. Wegner. The theory of zero-suppressed BDDs and the number of knight’s tours. In *Workshop on Apps. of the RM Expansion in Circuit Design*, 1995.
- [McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.
- [Min93] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *in Proc. of the Design Automation Conference*, pages 272–277, 1993.

- [MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.
- [Som] F. Somenzi. CUDD: CU Decision Diagram Package. <ftp://vlsi.colorado.edu/pub/>.
- [Wik06] Wikipedia. Dining philosophers problem — wikipedia, the free encyclopedia, 2006. [Online; accessed 23-May-2006].