2.3.2 BDDs

Reduced Ordered Binary Decision Diagrams (ROBDDs) provide a canonical representation of Boolean functions. They often are more compact that other canonical representations like Sum-Of-Products for instance. The presentation about BDD is based on Chapter 5 from the textbook by Clarke, Grunberg, and Peled [3].

Constructing BDDs

The fundamental notion underlying this section is Shannon's expansion. For a Boolean function f and a given variable x of f, it states that function f can be decomposed into two sub-functions considering the cases where either x is false or x is true. Formally, we have the following equation:

$$f = (\neg x \wedge f|_{x=0}) \vee (x \wedge f|_{x=1})$$

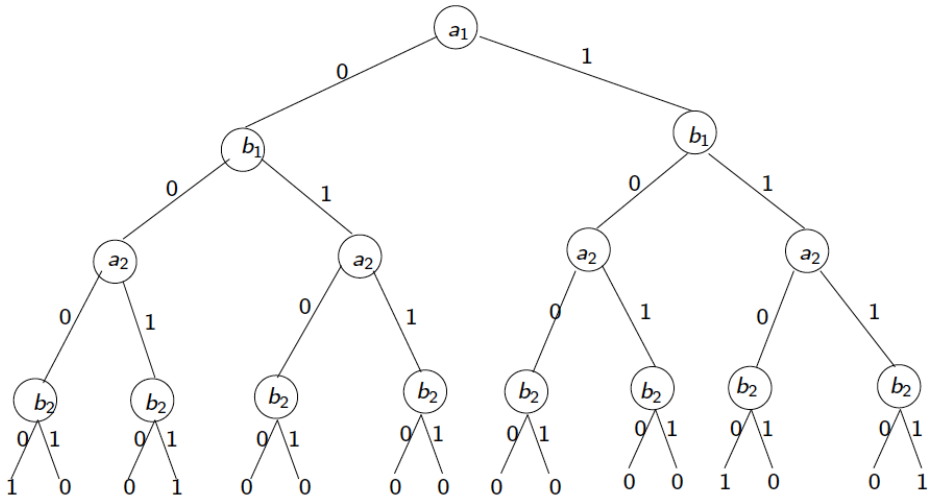

Figure 2.1: Binary decision tree for two-bit comparator.

Using Shannon's decomposition, it is possible to build a binary decision tree. Figure 2.1 shows the tree for a two-bit comparator, that is, for the following Boolean function:

$$f(a_1, a_2, b_1, b_2) = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$$

A binary tree basically is a rooted, directed tree with terminal and nonterminal vertices. Each nonterminal vertex v is labelled by a variable noted $var(v)$. Such a vertex has two successors: a successor corresponding to the case where the variable is low (false) noted $low(v)$ and a successor corresponding to the case where the variable

is high (true) noted $high(v)$. Graphically, we often identify the successor by a 0 or 1 label on the edges. Each terminal vertex v is either 0 or 1. To know the truth value of formula for a given assignment, one simple traverses the tree down to a leaf by following the low successor if a variable is false or the high successor if a variable is true. As one can see, such a tree is not a very compact representation.

Bryant [2] showed how to obtain a compact and canonical representation of Boolean functions by placing two restrictions on binary diagrams. The first restriction imposes an ordering among the variables. This means that variables must always appear in the same order along all paths from the root to terminals. Second, there should be no isomorphic sub-trees or redundant vertices in the diagram. To achieve the first restriction, a total order $<$ is imposed among the variables. We require that if any vertex u has a nonterminal successor v, then $var(u) < var(v)$. The second restriction is achieved by recursively applying the following rules:

- *Remove duplicate terminals:* Keep only one terminal vertex for each label (0 or 1) and redirect all edges to eliminated vertices to the remaining ones.
- *Remove duplicate nonterminals:* If two nonterminals vertices u and v are labelled with the same variables, that is, $var(u) = var(v)$, eliminate one of the vertices and redirect edges to the eliminated vertex to the remaining one.
- *Remove redundant tests:* If the two successors of a nonterminal vertex v are equal, that is, $low(v) = high(v)$, remove v and redirect incoming edges of v to $low(v)$.

Starting with an ordered diagram, the reduced version is obtained by applying these rules until the size can no longer be reduced. We then obtained a Reduced Ordered Binary Decision Diagram (ROBDD). Bryant [2] showed that this reduction can be done in a bottom-up manner and in time that is linear in the size of the original BDD. Figure 2.2 shows the ROBDD obtained from the decision tree in Figure 2.1.

Figure 2.2: ROBDD for two-bit comparator.

The variable ordering has a strong impact on the size of the ROBDD. For instance, choosing the ordering $a_1 < b_1 < a_2 < b_2$ the 2-bit comparator results in the ROBDD shown in Figure 2.3, which is larger than the one shown in Figure 2.2.

Figure 2.3: ROBDD for two-bit comparator.

ROBDDs are canonical given a variable ordering. This means that checking equivalence between two ROBDDs reduce to checking equality between two diagrams. Validity reduces to showing equivalence to the trivial diagram composed of one terminal labelled by 0.

To further illustrate the construction of BDDs we consider the following Boolean function:

$$f(a, b, c) = a \vee (b \wedge c)$$

We will use Shannon's expansion to build a Binary Decision Tree for this Boolean function. To apply the expansion we need to determine an ordering of the variables. Let us consider the ordering $a < b < c$.

We first decompose according to a, this gives the following equation:

$$f(a, b, c) = (\neg a \wedge f(0, b, c)) \vee (a \wedge f(1, b, c))$$

The first part of the tree from this equation is pictured in Figure 2.4.

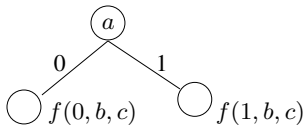

Figure 2.4: First part of the binary decision tree.

We can recursively proceed with decomposing each sub-formula. For instance, we can expand the subtree $f(0, b, c)$ and obtain:

$$f(0, b, c) = (\neg b \wedge f(0, 0, c)) \vee (b \wedge f(0, 1, c))$$

A similar expansion is obtained for $f(1, b, c)$. We then insert these terms in the tree and obtain the tree pictured in Figure 2.5.

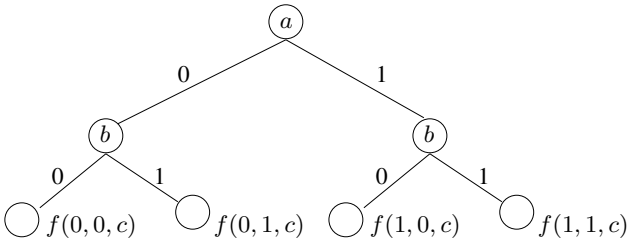

Figure 2.5: The binary decision tree continued ...

We proceed with the four leaves and then compute the values of f where all literals have been assigned a definite value. We obtain the Binary Decision Tree pictured in Figure 2.6.

We apply the rules proposed by Bryant [2] to transform this BDT into a Reduced Ordered BDD. We first apply the rule "remove redundant tests"[1], this means that we eliminate all nodes where the value at the end of the left and the right branches is the same. There are three cases where this applies. We obtain the graph in Figure 2.7.

The same rule applies once more to the true branch of a. After applying this rule, we apply the rule "remove duplicate terminals" and obtain the ROBDD in Figure 2.8.

The size of a ROBDD depends on the variable ordering. The following question illustrates this point.

[1]The order in which the rules are applied does not matter.

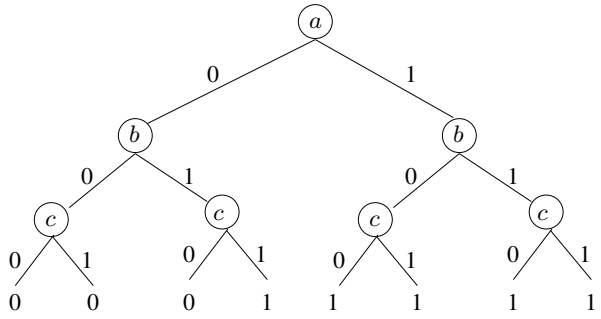

Figure 2.6: The binary decision tree

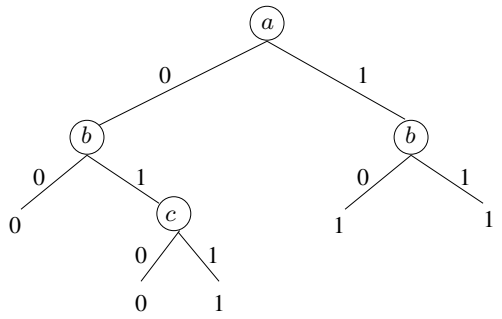

Figure 2.7: Reducing the BDT with "remove redundant tests".

Exercise 2.3.2. For which ordering(s) is the size of the ROBDD for our function f maximal? (Note: The size is counted in the number of nodes and edges)?

Finding an optimal ordering is infeasible and there exist functions with no optimal ordering, like bitwise multiplication. In such cases, explicit algorithms might perform better than symbolic ones.

Operations on BDDs

The main advantage of ROBDDs is that they can be manipulated efficiently. In particular, checking that two ROBDDs are equal requires constant time. Negation is also very easy and simply requires to negate the two leaves[2] of a ROBDD.

Other operations are performed using the Apply algorithm proposed by Bryant [2] and that can be used to compute any 2-argument logical operation. The main idea is to use Shannon's decomposition to break problems into sub-problems.

Let $\star$ be an arbitrary 2-argument logical operation. Let f and f' be two Boolean functions. Let v be the current vertex of f and v' be the current vertex of f'. The algorithm basically is the following case distinction:

[2]Note that any ROBDD has only two leaves, namely, 1 for "true" and 0 for "false".

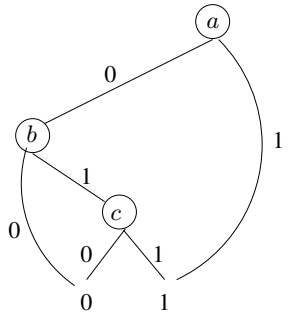

Figure 2.8: Reducing the BDT to a ROBDD.

- If v and v' are both terminals, $f \star f' = value(v) \star value(v')$.
- If $var(v) = var(v')$, apply Shannon's expansion:

  $$f \star f' = \neg var(v) \wedge (f|_{var(v)\leftarrow 0} \star f'|_{var(v)\leftarrow 0}) \vee var(v) \wedge (f|_{var(v)\leftarrow 1} \star f'|_{var(v)\leftarrow 1})$$

  to break the problem into subproblems. The resulting ROBDD will have a new node w with $var(w) = var(v)$, $low(w)$ will be the ROBDD for $(f|_{var(v)\leftarrow 0} \star f'|_{var(v)\leftarrow 0})$ and $high(w)$ will be the ROBDD for $(f|_{var(v)\leftarrow 1} \star f'|_{var(v)\leftarrow 1})$.
- If $var(v) < var(v')$, $f'|_{var(v)\leftarrow 0} = f'|_{var(v)\leftarrow 1} = f'$ since f' does not depend on $var(v)$. In this case, the Shannon's expansion reduces to:

  $$f \star f' = \neg var(v) \wedge (f|_{var(v)\leftarrow 0} \star f') \vee var(v) \wedge (f|_{var(v)\leftarrow 1} \star f')$$

- If $var(v') < var(v)$, similar to previous case.

By reducing and caching intermediate results, the algorithm is polynomial. The number of sub-problems is bounded by the product of the size of the ROBDDs for f and f'.

We illustrate the *Apply* algorithm using our small example:

$$f(a) \vee f'(b, c)$$

where $f(a) = a$ and $f'(b, c) = b \wedge c$.

We consider the ordering $a < b < c$. We assume we have the ROBDDs of each function, that is, the graphs pictured in Figure 2.9.

We now use Apply and Shannon's expansion to compute the disjunction of these two ROBDDs. Since the roots of the ROBDDs are not terminal vertices and since the variables in the two roots are not the same we need to determine which of the two variables comes first in the ordering. Since we consider the ordering where $a < b$, we apply the formula from the third case of the aforementioned algorithm:

$$f(a) \vee f'(b, c) = (\neg a \wedge (f(0) \vee f'(b, c))) \vee (a \wedge (f(1) \vee f'(b, c)))$$

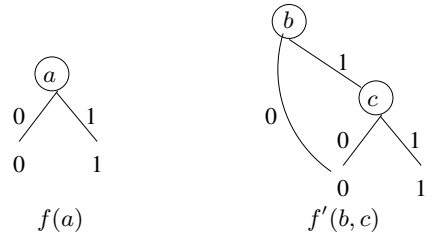

Figure 2.9: ROBDDs for f and f'.

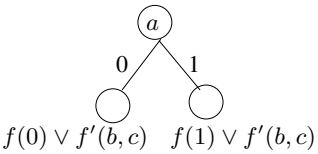

Figure 2.10: Partial ROBDDs for $f \vee f'$.

We now start to build a new ROBDD as illustrated in Figure 2.10.

Now, we have $f(0) = 0$ and $f(1) = 1$ (since $f(a) = a$). Therefore, the left branch simplifies to $f'(b, c)$ and the right branch to 1. We already have a ROBDD for $f'(b, c)$. We simply insert it as the left branch. Finally, we obtain the ROBDD pictured in Figure 2.8 for the formula:

$$f(a, b, c) = a \vee (b \wedge c)$$

2.3.3 AIGs

And Inverter Graphs (AIGs) are directed and acyclic graphs representing Boolean functions. They consist in two input nodes representing conjunction and edges with an optional label indicating negation. The size of an AIG is given by the number of AND nodes in it. The number of logic levels is the number of AND-gates on the longest path from a primary input to a primary output. If you remember your old course 2IC30 on Computer Systems, you should remind yourself about the completeness of the NAND gates. That is, any Boolean function can be represented using two-input NAND gates only. Therefore, AIGs can also represent any Boolean functions.

Figure 2.3.3 shows the AIGs representing basic gates. The only slighly more difficult one is the `xor` gate. One has to express this function using conjunction and negation only. This is given by the following expression:

$$x \text{ xor } y = \neg(x \wedge y) \wedge \neg(\neg x \wedge \neg y)$$

Figure 2.12 shows a circuit and its representation as an AIG. The AIG is constructed in a bottom-up manner starting from the Primary Outputs (POs). Note that each time all gates are expressed using `and`-gates and inverters only using equivalences like the ones shown in Figure 2.3.3. When a PI is reached, the construction stops with a leave

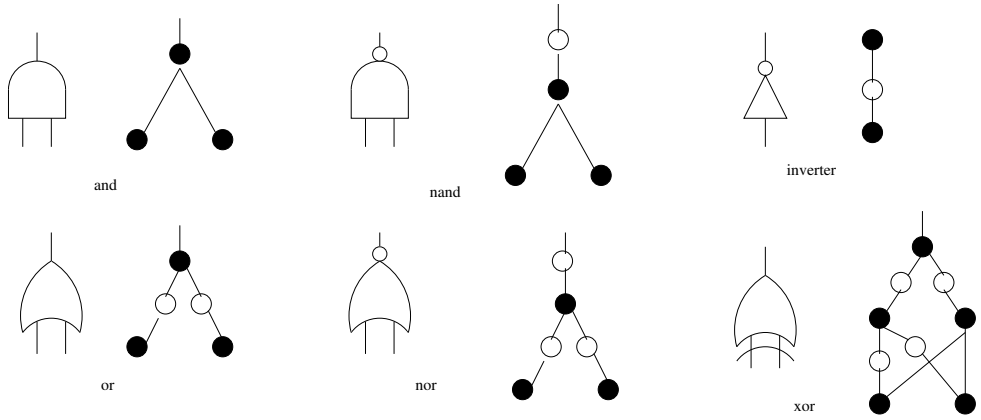

Figure 2.11: Basic gates and their representation using And Inverter Graphs

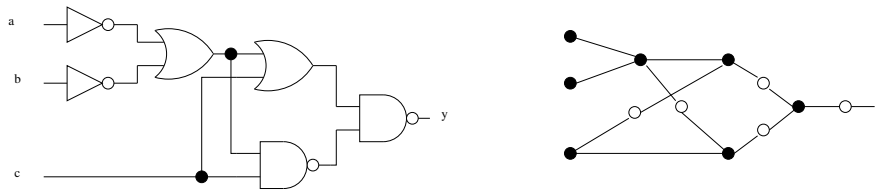

Figure 2.12: A circuit and its representation as an AIG.

node representing this input. Otherwise, the construction proceeds recursively for the fanins of the node.

The time needed to construct AIGs and their size is proportional to the size of the input, being a circuit, a SoP expression, or a BDD.

A nice feature of AIGs is that they can be smaller than Sum-Of-Products or BDDs. Figure 2.3.3 shows four cascaded `xor`-gates and their AIGs. The SoP representation of this function would be exponentially large, whereas the AIG is rather small. Also, there are BDDs that are exponential – e.g., multiplier circuits – while the size of an AIG always is proportional to the size of the input circuit, SoP, or BDD.

Also, AIGs can be smaller than the circuit they represent. Figure **??** shows such an example. The circuit has xx gates while the AIG has only ... CHEK EXAMPLE

The main drawback of AIGs is that they are not *canonical*, that is, testing for equality is not trivial. AIGs may contain syntactically distinct but functionally equivalent – that is redundant – nodes. Manipulating such AIGs is also not efficient.

To solve this issue, algorithms have been developed introduce some canonicity, in particular, Functionally Reduced AIGs (FRAIGs) [5]. The idea is to reduce AIGs such that they do not exist any pairs of functionally equivalent nodes. FRAIGs are semi-canonical in the sense that a Boolean function may have distinct FRAIGs structures. Recent work [5] proposes an algorithm that generates on-the-fly an AIG that is ensured

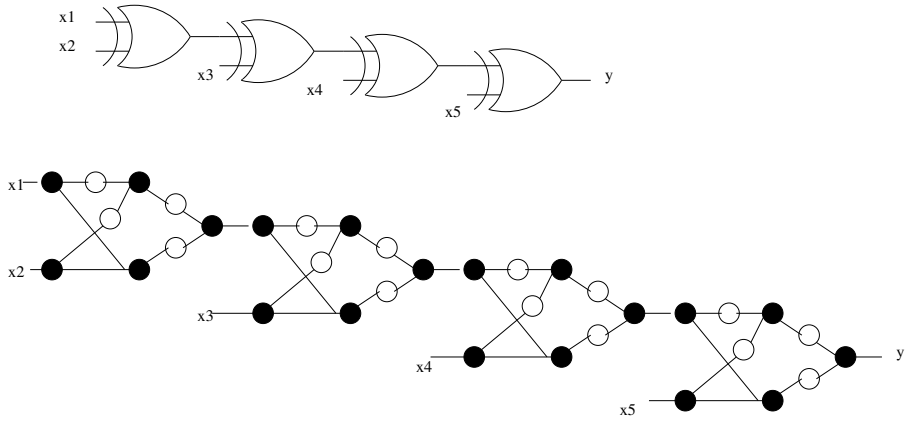

Figure 2.13: A cascade of four `xor`-gates.

to be functionally reduced.

AIGs used at Galois to verify Crypto stuff?

2.3.4 Constructing AIGs from SoP

Given a Boolean function f represented using SoP

2.4 Boolean satisfiability (SAT)

Boolean satisfiability consists in checking whether there exists an assignment of the free variables of a Boolean formula that makes this formula "true". This problem is well-known and is very hard (NP-Complete). Still, there are several SAT solvers that can effectively check satisfiability of formulas with thousands of variables. Also, recent advent in SAT technologies makes it possible to check correctness of UNSAT results [**?**]. SAT solvers are complex software and their output cannot in general be trusted.

Very short introduction about SAT solving? (Actually used also in BMC?) Be careful not to overlap with Automated Reasoning

2.5 Combinatorial Equivalence Checking (CEC)

One of the main application of formal verification techniques is Boolean equivalence checking. The idea is to check whether two Boolean functions are equivalent, that is, for all possible input vectors they produce the same outputs.

2.5.1 Using SoP

SoP can be used as a normal form. This means that two equivalent functions are syntactically equivalent. The procedure is simple. Expand the two functions into their SoP and compare the result syntactically. The issue is that the SoP expansion grows very quickly. This makes SoP not practically applicable for equivalence checking.

2.5.2 Using BDDs

ROBDDs are also a normal form. To check the equivalence between two functions, build their ROBDD and check equality of their root pointer. Building ROBDDs is more efficient than SoP. BDDs have been the main tool for equivalence checking for a long time.

2.5.3 Using AIGs

AIGs – in contrast to SoP and BDDs – do not have canonical representations. Checking for equality seems therefore more difficult. This issue is solved using a SAT solver to determine local equality between AIG nodes. When equal nodes are found, they are removed until the remaining AIGs are equal. Because SAT solvers are very efficient, the queries sent to the SAT solver are answered very quickly. Nowadays, AIGs are at the core of modern CEC checkers.

The technique [**?**] proceeds in three steps:

1. Random Simulation: First candidate equivalent nodes are found by simulating the circuits using random test vectors
2. Build AIGs: More equivalent nodes are found by constructing an AIG of the combination of the two circuits.
3. SAT Sweeping: Candidates nodes are used to guide calls to the SAT solver

We illustrate the technique on the following example:
The technique is to build the AIG of one of the two circuits.
Then, we ”merge” the AIG of the second circuit with the AIG of the first one.

2.6 References and further reading

AIGER library of Armin Biere. Manipulate and reduce /optimise AIG representations.

2.7 Exercises

Exercise 2.7.1. Draw the AIG for the circuit shown in Figure **??**.

Make exercise that should take max 2 hours together (these exercises are meant to check understanding + deepen understanding)

1. Exercise 1 - experience exponential blow up of SoP?
2. Exercise 2 - build BDD and check that the ordering matters
3. Exercise 3 - build an AIG and simplifies it to a FRAIG
4. Exercise 4 - equivalence checking using BDD (work on ROBDD rules)
5. Exercise 5 - equivalence checking using AIGs

Chapter 3

Sequential equivalence and reachability analysis

3.1 Moore and Mealy machines

3.2 Reachability analysis

3.2.1 Symbolic method using BDDs

Transition functions as BDDs

We show here how to represent the transition relation of a Kripke structure using ROBDDs.

Reachability with BDDs

3.2.2 Inductive proofs

3.3 Sequential equivalence checking

Basically go for reachability analysis using a miter.

3.4 Further reading

3.5 Exercises

Chapter 4

Temporal Logics

4.1 Semantics

Definition 4.1.1. A Kripke structure is a tuple M = (S, I, T, L) with a finite set of state S, a finite set of initial states $I \subseteq S$, a finite transition relation between states $T \subseteq S \times S$, and a labelling function $L : S \to \mathcal{P}(A)$ with atomic propositions A.

Definition 4.1.2. LTL semantics

Chapter 5

Model checking: with or without BDDs

This chapter presents two techniques to verify temporal properties. The first technique is based on BDDs. The second technique is based on SAT. Historically, BDD-based model checking was introduced first. The idea was to use the symbolic representation provided by BDDs to explore larger state spaces. In many cases, this approach turned out to be indeed more efficient than explicit techniques, where all states are explicitly represented in the memory. Still, there are cases where explicit state model checking out-performs BDD techniques. Later, SAT solvers became very good at solving large problems. The idea of SAT-based techniques was to leverage this new technology to verification. Today, SAT-based techniques are very often out-performing other ones. Still, there are cases where BDD-based solvers work better.

5.1 Symbolic model checking with BDD

5.1.1 Expression

5.2 Bounded model checking with SAT

The presentation is largely based on the original paper by Biere et al. [1] with some small modifications. The presentation of the completeness threshold differs more significantly from the original paper.

5.2.1 Principle

The main idea behind Bounded Model Checking (BMC) is to prove a property by showing that there exists no finite counter-examples to it. There actually two ideas here: (1) prove that no counter-example exists and (2) restrict to finite paths.

Assume one wants to prove an LTL property of the form $\mathbf{G}p$. The semantics of LTL are such that this property holds if and only it holds on all paths. The negation of