15-453 Formal Languages, Automata, and Computation

Binary Decision Diagrams and Model Checking

Mihai Budiu

Lecture 4
January 26, 2000

This lecture is a continuation of Lecture 3. We will see two applications which are very important for designing digital computers.

1 (Ordered) Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are a derivative of the decision trees. We have seen that decision trees are used by AI for learning and classification: i.e. to represent knowledge in a compact way. BDDs are used for the same purpose, for a restricted domain: representing boolean functions.

1.1 Boolean Functions

Boolean functions have many applications; the most prominent are probably in VLSI digital circuits design, optimization and verification.

The Boolean algebra is named in honor of Georges Boole (1815–1864), who invented it in order to formalize, using the language of mathematics, the classic Aristotelian logic. The Boolean algebra operates on two values only, which we can call "false" and "true", or "0" and "1". A boolean variable can have one of these two values. We assume the reader is familiar with the fundamental boolean functions "and" ($\wedge$), "or" ($\vee$), "not" ($\neg$), "implies" ($\Rightarrow$) and "equivalent" ($\Leftrightarrow$).

An n-ary boolean function takes n boolean variables as input and generates a single boolean value. We will generally denote the n input values by x_1, x_2, ..., x_n. We will also denote the string $x_1, \ldots, x_n$ as $\overline{x}$. Table 1 shows the boolean function "and".

x_1	x_2	$x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1

Table 1: The binary boolean function "and" maps a pair of boolean values to a single boolean value.

1.2 Decision Trees for Boolean Functions

We can view a boolean function of n arguments as a *classifier over binary strings of length* n: each string represents the input values; a string is mapped to either 0 or 1. Then we can build a decision tree to classify such strings based on the component characters. Figure 1 shows two different decision trees for the "and" boolean function.

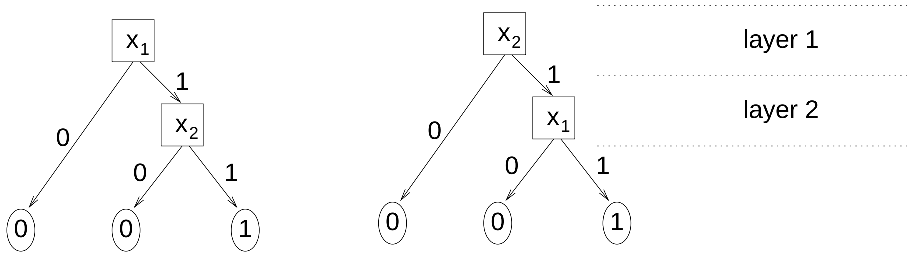


Figure 1: Two decision trees which compute the same binary boolean function, "and".

We will now impose some constraints on how the decision tree is built:

1. We *order* all the input values once and forever (this is why we will call the resulting diagrams *Ordered*).

2. We ask that a decision tree for a function with n inputs always has n "layers"; we also ask that on layer i we only inquire about the value of variable x_i[1].

Note that the ordering influences dramatically the size of the resulting tree; there is no known efficient algorithm to find the best ordering (this problem is NP-hard[2]).

Exercise: *give an example of a function of n inputs which has two trees with widely different sizes, when the trees differ in the ordering of the variables.*

Notation: from now on, to reduce the clutter, we will represent the arrows labeled "0" with dashed lines.

Exercise: *using a counting argument, prove that there are boolean functions which require very large decision trees (i.e. a function of n variables requires $\Omega(2^n/n)$ nodes in a tree). Start by counting the number of boolean functions of n variables.*

We can view such decision trees as DFAs which decide languages $\subseteq \{0,1\}^n$.

1.3 Ordered Binary Decision Diagrams

OBDDs were introduced in 1986 by Randal Bryant, in the paper "*Graph-Based Algorithms for Boolean Function Manipulation*", IEEE Transactions on Computers, C-35(8). Randy Bryant is the Dean of the CMU School of Computer Science.

Bryant imposed one additional condition:

3. The tree should not contain two identical subtrees.

[1]If a function is independent on some input variable (i.e. the variable is a don't care), some layers may be empty.

[2]NP-hardness will be defined later in a lecture in the last part of this course.

Let us consider the boolean function which compares two two-bit values for equality; this function has four inputs variables $f(x_1, x_2, y_1, y_2) = (x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$. The decision tree and OBDD for this function are depicted in Figure 2.

Exercise: *argue that the OBDD is always smaller than the corresponding tree.*

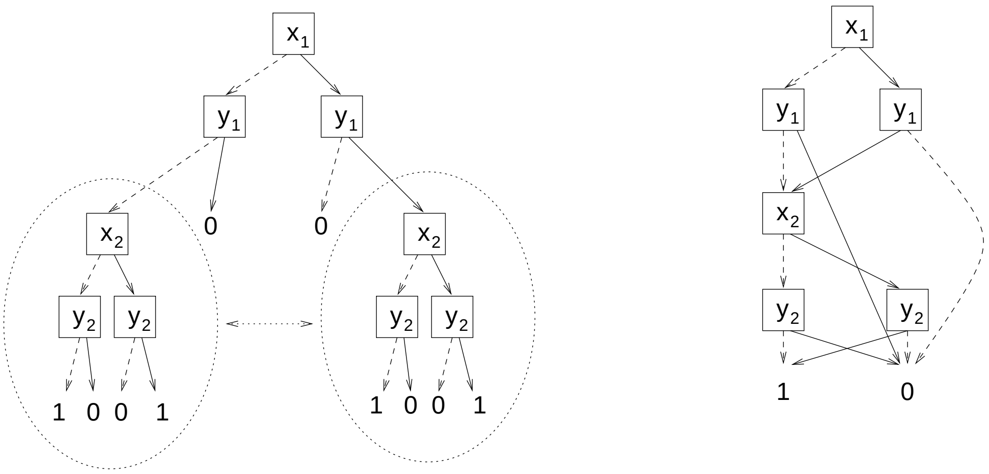


Figure 2: The decision tree (left) and the OBDD (right) for the equivalence function for two strings of 2 bits each, for the same ordering. All isomorphic (i.e. identical) subtrees in the decision tree are merged in the OBDD.

In the cited paper Bryant shows:

- How to build OBDDs quickly given a decision tree (a linear-time algorithm).
- That two OBDDs are identical if and only if the boolean functions they represent are also identical.
- How to implement boolean function computations on OBDDs in linear time (e.g. how to compute $f(\overline{x}) \wedge g(\overline{x})$, $f(\overline{x}) \vee g(\overline{x})$, etc., without going from OBDDs to boolean functions and back.)

Exercise: *prove that, although OBDDs are always smaller than the corresponding trees, there still are boolean functions which require OBDDs of exponential size in the number of inputs.*

OBDDs with thousand of vertices can be represented and manipulated efficiently by computers.

1.4 Other Applications of OBDDs

1.4.1 Representing Languages $L \subseteq \{0, 1\}^n$

We can represent such a language by its *characteristic function*: $f(\overline{x}) = 1$ iff $\overline{x} \in L$. The OBDDs is thus a deterministic finite automaton accepting L.

Exercise: *Is the OBDD the smallest automaton (in terms of numbers of states) which recognizes L? Why?*

1.4.2 Representing Relations

(The paragraphs with small fonts, 1.4.2 and 1.4.3 are optional.)

We can use OBDDs to represent relations over finite domains.

- To represent n-ary relations over $\{0, 1\}$ we can represent the characteristic function of the relation: $f_R(x_1, \ldots, x_n) = 1$ iff $R(x_1, \ldots, x_n)$, which is a boolean function.
- To represent relations over a bigger finite domain D^n we can encode the elements of D using $m = \log_2 |D|$ bits, and represent the relation as m relations over $\{0, 1\}^n$.
- We can further generalize this for the case of relations over different finite domains $D_1 \times D_2 \times \ldots \times D_n$.

1.4.3 Representing Arbitrary Digital Circuits

We can use BDDs to represent functions with many (k) boolean outputs (formally, this is a vector of boolean functions $(f_1(\overline{x}), \ldots, f_k(\overline{x})))$. For this purpose, we build the OBDDs for each output bit separately. Next we merge the OBDDs such that:

- They use the same set of nodes;
- There are no isomorphic subgraphs;
- The resulting automaton has *several start states*: one for computing each output bit.

For instance, we can represent an adder of 2 two-bit values like in Figure 3.

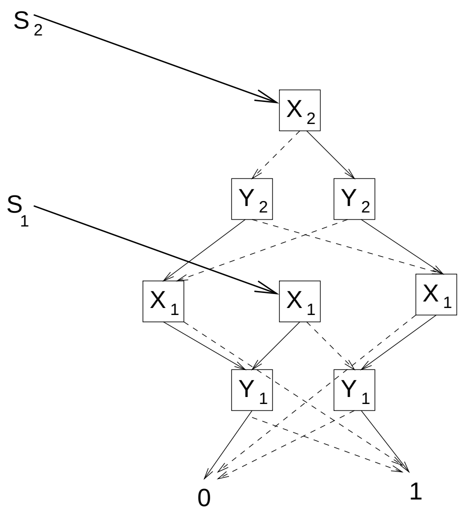
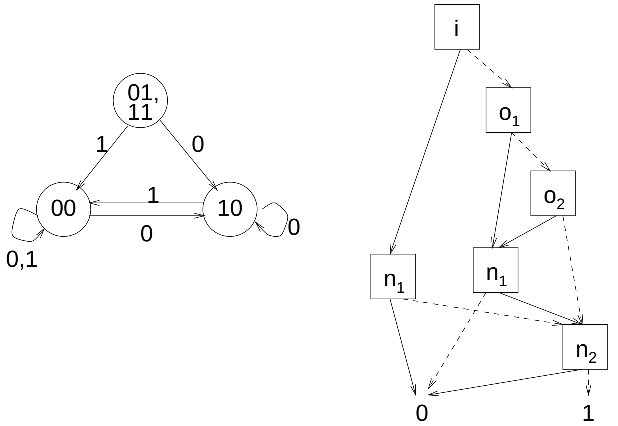


Figure 3: OBDD for representing a two-bit binary adder: $S_2 S_1 = X_2 X_1 + Y_2 Y_1$. The OBDD has as many roots as there are computed bits.

1.4.4 Representing NFAs

We can use BDDs to represent *non*deterministic finite automata. For this purpose, we must encode each state and each input symbol as a binary string. Let's say each input symbol is encoded as a string $i_1, \ldots, i_n$, and the states are encoded as $s_1, \ldots, s_m$. Then we can represent the fact that we make a transition from a state o to a state n on the input symbol i as a function $\delta(i_1, \ldots, i_n, o_1, \ldots, o_m, n_1, \ldots, n_m) = 1$.

Figure 2 shows such an example.



old		in	new	
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
1	0	0	1	0
1	0	1	0	0
0	1	1	0	0
1	1	1	0	0
0	1	0	1	0
1	1	0	1	0

Table 2: A non-deterministic finite automaton over $\Sigma = \{0, 1\}$ and its OBDD representation. For simplification we have encoded one state with two different encodings. $o_1 o_2$ is the encoding of the old state, while $n_1 n_2$ is the encoding of the new state. The OBDD represents the transitions $o \xrightarrow{i} n$. The ones of the transition function are represented the table. The inputs for which the transition function δ of the BDD in Figure 2 is 1. The first row in the table, for example, indicates that $delta(0, 0, 0, 0, 0) = 1$, which means that $00 \xrightarrow{0} 00$.

We can implement interesting computations on NFAs (e.g. product, union of their languages, etc.) by manipulating just their OBDD representation.

2 Model Checking

Model checking is an automatic technique for verifying finite-state reactive systems, such as sequential digital circuits or communication protocols. (A reactive FA is an automaton whose inputs come from the environment, and not from a "tape".)

This technique was pioneered (among others) by Edmund Clarke, professor in the CS Dept of CMU, in 1981 (E.M. Clarke and E.A. Emerson. "*Synthesis if Synchronization Skeletons for Branching Time Temporal Logic*", in Logic of Programs workshop, Yorktown Heights, NY, May 1981.).

Model checking is used by hardware design companies to test the designs for correctness. Model checking has been successfully used to find bugs in published standards: e.g. the IEEE Futurebus+ standard 896.1–1991. Model Checking is currently a very hot topic of research. Model checking's capabilities are still 15 years behind what modern technology VLSI circuits currently manipulates. This is due to the *state-explosion* problem, explained a little later.

OBDDs are used in model checking extensively, to represent the state-transition graph, as described above.

2.1 Kripke Automata and Traces

The basic object under scrutiny by Model Checking is the NFA. Each state is *labeled* with some propositions that hold when the automaton is in that state. For example, for an automaton controlling traffic lights, a state may be labeled "light is red". Model checking generally ignores completely the events which cause the transitions: the resulting model is called Kripke Automaton.

Model Checking tries to prove statements about *all possible evolutions* of the automaton. It is thus convenient to see the computation of the automaton as a trace, or an *infinite computation tree.* The tree indicates the possible sequence of paths the automaton may take; the actual execution will be just *one* of those paths. We illustrate a Kripke Automaton and part of its computation tree in Figure 4.

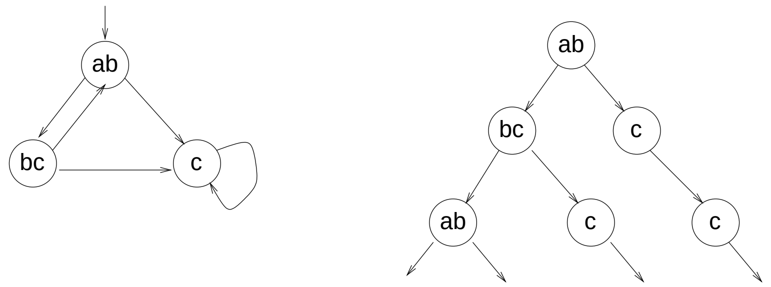


Figure 4: A Kripke Automaton is an NFA whose transition events (symbols) are ignored. Each state is labeled with some atomic propositions that hold in that state. Model Checking reasons about the infinite computation tree of such an automaton.

2.2 Composing Automata

Real complex systems are usually described as a collection of multiple interacting NFAs (or IO automata). Figure 5 illustrates this fact for the TCP protocol.

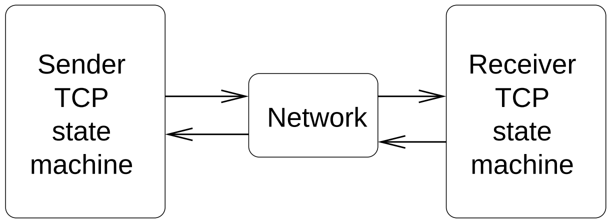


Figure 5: The system we are modeling is often composed of several interacting automata. For example, a complete model of the TCP protocol should include both communicating parties and the network between them. The whole protocol is a collection of three independent state-machines, whose transitions are sometimes correlated (e.g. if the sender emits a packet, the network receives it).

The composition of several automata can be represented as a bigger automaton, whose states are the product of the component states[3].

[3]This is similar to the construction of the automaton accepting the union of two languages with given

Exercise: *try to formalize this precisely.*

Real systems are actually composed of hundreds, and sometimes hundreds of thousands of such automata. The product of n automata with s states has s^n states. This is an exponential blow-up in the total number of states (compared to the components, which have $n \cdot s$ states in all). This problem is called *state explosion*, and is the main reason why model checking still cannot verify real microprocessors in entirety. A lot of research is being done in this area, and now even systems with as much as 10^{20} states can be checked.

2.3 Temporal Logics

Temporal logic is used to express properties of the traces of the execution of a Kripke automaton. Here are some examples of *linear time logical operators* and *path quantifiers* (we will not define these formally):

- $\mathbf{F}p$: the property p will be true sometime in the future
- $\mathbf{G}p$: the property p will always be true in the future
- $\mathbf{A}$: for every path starting in the "current" state
- $\mathbf{E}$: there is some path starting in the "current" state

Using these we can create composite operators. Figure 6 shows the meaning of some of the most widely used operators.

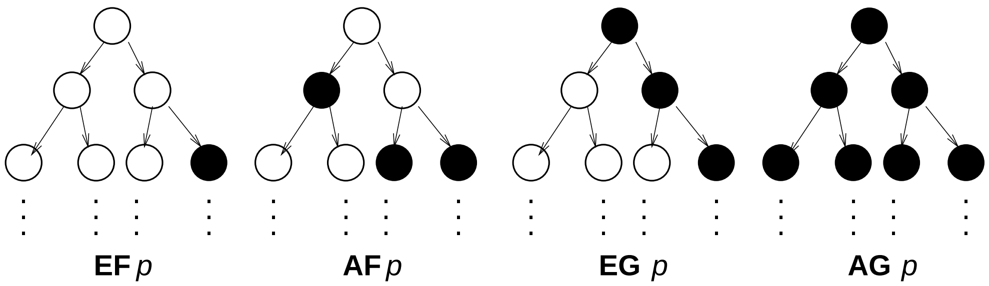


Figure 6: The four most widely used operators of the Computation Tree Logic. We shaded the states where the property p holds. The "current" state is the root of the tree.

We can read these operators as follows:

- $\mathbf{EF}p$: the property p will eventually be true on some path starting from the current state.
- $\mathbf{AF}p$: on all paths starting from the current state p will be true at some point.
- $\mathbf{EG}p$: there is a path starting at the current state where p always holds.
- $\mathbf{AG}p$: on all paths from now on, all states have property p.

Here are some example formulas:

automata.

- **EF**(¬ Green ∧¬ Red): It is possible to reach a state where the light is neither green nor red.
- **AG**(Push ⇒ **AF** Green): If (a pedestrian) pushes the button, the light will eventually become green.
- **AG**(**AF** Red): Any way the system evolves, the light will turn red infinitely often.

2.4 An example of Model Checking

The Model Checking problem is: given a Kripke automaton and a logical formula f expressed in temporal logic, find all states on the computation paths where f holds. For some of the temporal logics there are efficient (polynomial-time) algorithms that can check that.

Figure 7 shows an example where we try to see if the proposition $\mathbf{AF}b$ holds for the starting state.

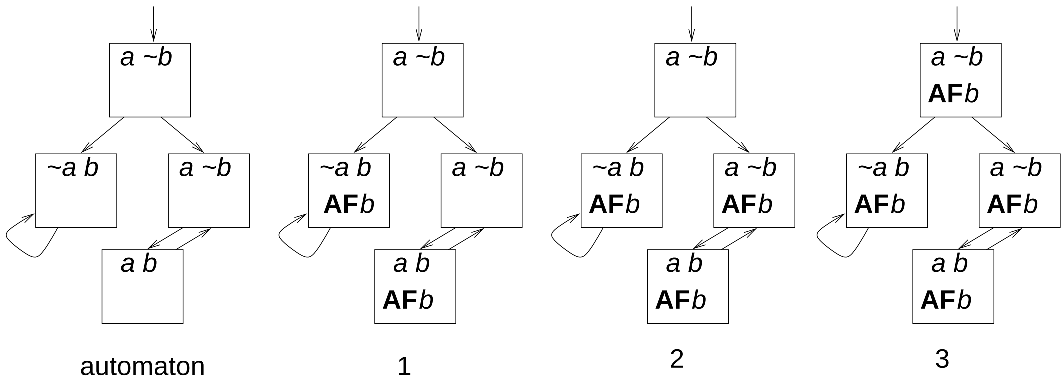


Figure 7: An example application of a model-checking algorithm. Each state is labeled with the propositions that hold. We try to verify if $\mathbf{AF}b$ holds for the starting state. The figure shows the evolution of the three computation steps. In each step we propagate information backwards on the transition arrows. Because at the end of the algorithm the starting state is labeled with that formula, we can say "yes".

The algorithm can be described informally like follows:

- Decompose into pieces the formula we want to check.
- Label all states where some of the pieces are true with the parts which are true.
- At each step, propagate information along the transitions.
- When no changes occur, stop.

In the example in Figure 7, if all successors of a state are labeled with $\mathbf{AF}b$, we also label the state itself with $\mathbf{AF}b$.

3 Conclusion

OBDDs and model checking both operate on Finite Automata. Both have extensive applications, and both are still areas of intense research.