11

The Automaton/Logic Connection, Symbolic Techniques

Most believe that computer science is a very young subject. In a sense, that is true - there was the theory of relativity, vacuum tubes, radio, and Tupperware well before there were computers. However, from another perspective, computer science is at least 150 years old! Charles Babbage[1] started building his Analytic Engine in 1834 which remained unfinished till his death in 1871. His less programmable Difference Engine No. 2 was designed between 1847 and 1849, and built to his specifications in 1991 by a team at London's Science Museum. As for the 'theory' or 'science' behind computer science, George Boole published his book on Boolean Algebra[2] in 1853.

Throughout the entire 150 years (or so) history of computer science, one can see an attempt on part of researchers to understand *reasoning* as well as *computation* in a unified setting. This direction of thinking is best captured by Hilbert in one of his famous speeches made in the early 1900s in which he challenged the mathematical community with 23 open problems. Many of these problems are still open, and some were solved only decades after Hilbert's speech. One of the conjectures of Hilbert was that the entire body of mathematics could perhaps be "formalized." What this meant is basically that mathematicians had no more creative work to carry out; if they wanted to discover a new result in mathematics, all they had to do was to program a computer to systematically crank out all possible proofs, and check to see whether the theorem whose proof they are interested in appears in one of these proofs!

[1] Apparently, Babbage is also credited with the invention of the 'cow-catcher' that you see in front of locomotive engines!

[2] Laws of thought. (You might add: to prevent loss of thought through loose thought).

In 1931, Kurt Gödel dropped his 'bomb-shell.[3] He formally stated and proved the result, "Such a device as Hilbert proposed is impossible!" By this time, Turing, Church, and others demonstrated the true limits of computing through concrete computational devices such as Turing machines and the Lambda calculus. The rest "is history!"

11.1 The Automaton/Logic Connection

Scientists now have a firm understanding of how computation and logic are inexorably linked together. The work in the mid 1960s, notably that of J.R. Büchi, furthered these connections by relating branches of mathematics known as *Presburger arithmetic* and branches of logic known as WS1S[4] with deterministic finite automata. Work in the late 1970s, notably by Pnueli, resulted in the adoption of *temporal logic* as a formal logic to reason about concurrency. Temporal logic was popularized by Manna and Pnueli through several textbooks and papers. Work in the 1980s, notably by Emerson, Clarke, Kurshan, Sistla, Sifakis, Vardi, and Wolper established deep connections between temporal logic and automata on infinite words (in particular Büchi automata). Work in the late 1980s, notably by Bryant, brought back yet another thread of connection between logic and automata by the proposal of using binary decision diagrams, essentially minimized deterministic finite automata for the finite language of satisfying instances of a Boolean formula, as a data structure for Boolean functions. The *symbolic model checking algorithm* proposed by McMillan in the late 1980s hastened the adoption of BDDs in verification, thus providing means to tackle the correctness problem in computer science. Also, spanning several decades, several scientists, including McCarthy, Wos, Constable, Boyer, Moore, Gordon, and Rushby, led efforts on the development of mechanical theorem-proving tools that provide another means to tackle the correctness problem in computer science.

11.1.1 DFA can 'scan' and also 'do logic'

In terms of practical applications, the most touted application domain for the theory of finite automata is in string processing – pattern matching, recognizing tokens in input streams, scanner construction, etc. However, the theory of finite automata is much more fundamental

[3] Some mathematicians view the result as their salvation.
[4] WS1S stands for *the weak monadic second-order logic of one successor.*

to computing. Most in-depth studies about computing in areas such as concurrency theory, trace theory, process algebras, Petri nets, and temporal logics rest on the student having a solid foundation on *classical automata*, such as we have studied so far. This chapter introduces some of the less touted, but nevertheless equally important, ramifications of the theory of finite automata in computing. It shows how the theory of DFA helps arrive at an important method for representing *Boolean functions* known as *binary decision diagrams*. The efficient representation as well as manipulation of Boolean functions is central to automated reasoning in several areas of computing, including computer-aided design and formal verification. In Chapter 21, we demonstrate how exploiting the "full power" of DFAs, one can represent logics with more power than propositional logic. In Chapter 22, we demonstrate how automata on infinite words can help reason about finite as well as infinite computations generated by finite-state devices. In this context, we briefly sketch the connections between automata on infinite words as well as temporal logics. We now turn to binary decision diagrams, the subject of this chapter.
Note: We use $\vee$ and $+$ interchangeably, depending on what looks more readable in a given context; they both mean the same (the *or* function).

11.2 Binary Decision Diagrams (BDDs)

Binary Decision Diagrams (BDDs) are bit-serial DFA for satisfying instances of Boolean formulas.[5] To better understand this characterization, consider the *finite* language

$$L_1 = \{abcd \mid d \vee (a \wedge b \wedge c)\}.$$

Since all finite languages (a finite number of finite strings in this case) are regular, a regular expression describing this language can be obtained by spelling out all the satisfying instances of $d \vee (a \wedge b \wedge c)$. This finite regular language is denoted by the following regular expression:

```
(1110+1111+0001+0011+0101+0111+1001+1011+1101)
```

By putting this regular expression in a file called `a.b.c+d`, we can use the following `grail` command sequence to obtain a minimal DFA for it, as shown in Figure 11.1(a):

[5] BDDs may also be viewed as optimized decision trees. We view BDDs as DFA following the emphasis of this book. Also note that strictly speaking, we must say *Reduced Ordered Binary Decision Diagrams* or ROBDDs. We use "BDD" as a shorthand for ROBDD.

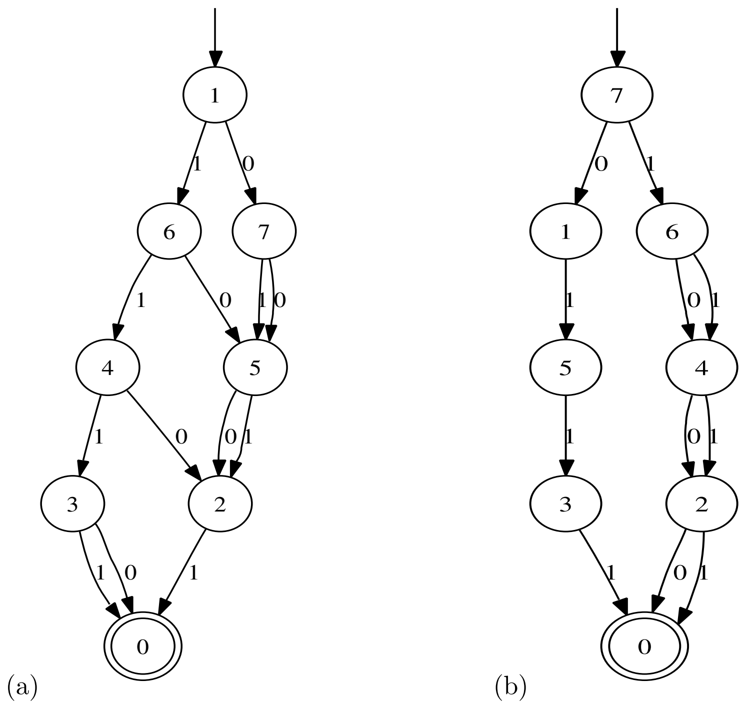


Fig. 11.1. Minimal DFAs for $d \vee (a \wedge b \wedge c)$ for (a) variable ordering *abcd*, and (b) *dabc*. The edges show transitions for inputs arriving according to this order.

```
cat a.b.c+d | retofm | fmdeterm | fmmin | perl grail2ps.perl -
   > a.b.c+d.ps
```

Now consider the language that merely changes the bit-serial order in which the variables are examined from *abcd* to *dabc*:

$$L_2 = \{dabc \mid d \vee (a \wedge b \wedge c)\}.$$

Using the regular expression

```
(0111+1000+1001+1010+1011+1100+1101+1110+1111)
```

as before, we obtain the minimal DFA shown in Figure 11.1(b). The two minimal DFAs seem to be of the same size. Should we expect this in general? The minimal DFAs in Figure 11.1 and Figure 11.2, are suboptimal as far as their role in decoding the binary decision goes, as they contain redundant decodings. For instance, in Figure 11.1(a),

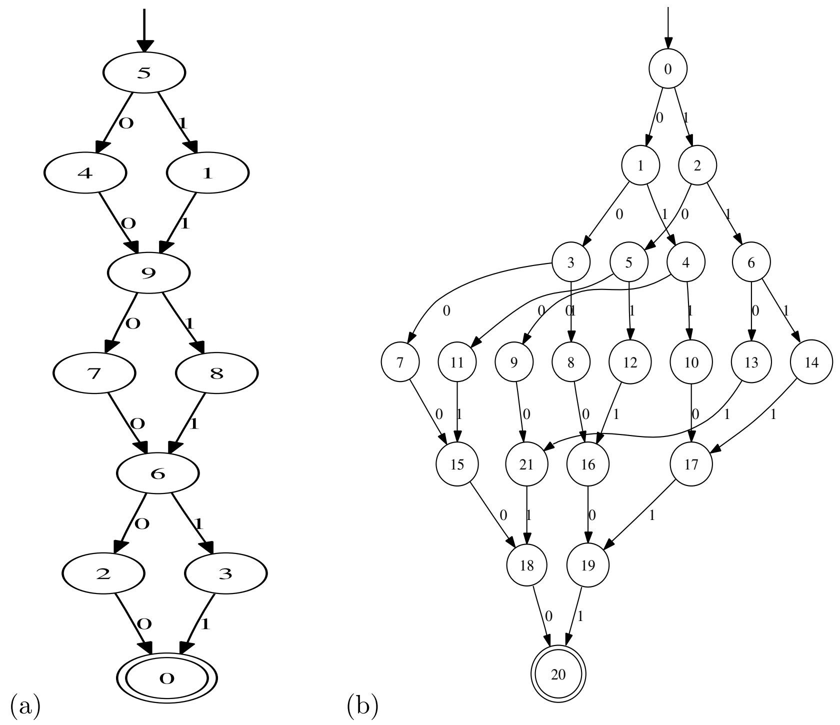

(a) (b)

Fig. 11.2. Minimal DFAs where the variable ordering matters

after $abc = 111$ has been seen, there is no need to decode d; however, this diagram redundantly considers 0 and 1 both going to the accepting state 0. In Figure 11.1(b), we can make node 6 point directly to node 0. Eliminating such redundant decodings, Figures 11.1(a) and (b) will, essentially, become BDDs; the only difference from a BDD at that point would be that BDDs explicitly include a 0 node to which all falsifying assignments lead to.

Let us now experiment with the following two languages where we shall discuss these issues even more, and actually present the drawing of a BDD.

$$L_{interleaved} = \{abcdef \mid a = b \wedge c = d \wedge e = f\}$$

has a regular expression of satisfying assignments

```
(000000+001100+000011+110000+001111+110011+111100+111111)
```

and

$$L_{noninterleaved} = \{acebdf \mid a = b \wedge c = d \wedge e = f\}$$

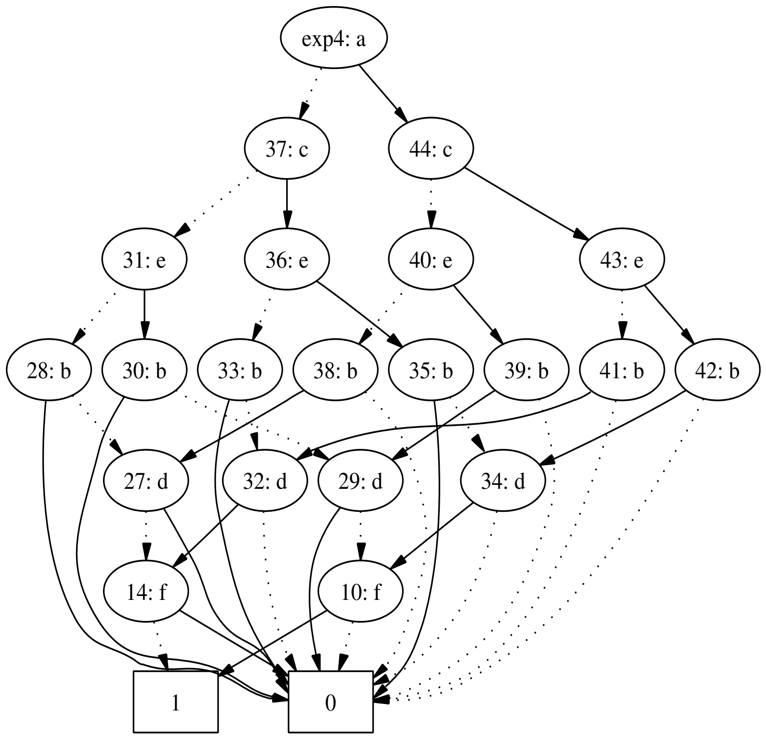


Fig. 11.3. BDD for $a = b \wedge c = d \wedge e = f$ for variable order *acebdf*

has

```
(000000+010010+001001+100100+011011+101101+110110+111111).
```

When converted to minimized DFAs, these regular expressions yield Figures 11.2(a) and (b), where the size difference due to the *variable orderings* is very apparent. The BDD for Figure 11.2(b) created using the BED tool appears in Figure 11.3. The commands used to create this BDD were:

```
bed> var a c e b d f                    % declares six variables
bed> let exp4 = (a=b) and (c=d) and (e=f) % defines the desired expn.
bed> upall exp4                         % builds the BDD -
bed> view exp4                          % displays the BDD
```

By comparing Figure 11.3 against Figure 11.2(b), one can see how, in general, BDDs eliminate redundant decodings.[6]

[6] The numbers inside the BDD nodes—such as the "14:" and "10:" in the nodes for variable f—may be ignored. They represent internal numberings chosen by the BED tool.

BDDs are efficient data structures for representing Boolean functions and computing the reachable states of state transition systems. In these applications, they are very often ‘robust,’ *i.e.*, their sizes remain modest as the computation advances. As many of these state transition systems have well over 2^{150} states (just to pick a large number!), this task cannot be accomplished in practice by explicitly enumerating the states. However, BDDs can often very easily represent such large state-spaces by capitalizing on an implicit representation of states as described in Section 11.3. However, BDDs can deliver this ‘magic’ only if a “good” variable ordering is chosen.

One also has to be aware of the following realities when it comes to using BDDs:

- The problem of determining an optimal variable ordering is NP-complete (see Chapter 20 for a definition of NP-completeness). [42]; this means that the best known algorithms for this task run in exponential worst-case time.
- In many problems, as the computation proceeds and new BDDs are built, variable orderings must be recomputed through *dynamic variable re ordering* algorithms, which are never ideal and add to the overhead.
- For certain functions (e.g., the middle bits of the result of multiplying two N-bit numbers), the BDD is provably exponentially sized, no matter which variable ordering is chosen.

Even so, BDDs find extensive application in representing as well as manipulating state transition systems realized in hardware and software. We now proceed to discuss how BDDs can be used to represent state transition relations and also how to perform reachability analysis.

11.3 Basic Operations on BDDs

BDDs are capable of efficiently representing transition relations of finite-state machines. In some cases, transition relations of finite-state machines that have of the order of 2^{100} states have been represented using BDDs. For example, a BDD that represents the transition relation for a 100-bit digital ripple-counter can be built using about 200 BDD nodes.[7] Such compression is, of course, achieved by *implicitly* representing the state space; an explicit representation (*e.g.*, using pointer based

[7] Basically, each bit of such a counter toggles when all the lower order bits are a 1, and thus all the BDD basically represents is an *and* function involving all the bits.

data structures) of a state-space of this magnitude is practically impossible. Given a transition relation, one can perform *forward* or *backward reachability* analysis. 'Forward reachability analysis' is the term used to describe the act of computing reachable states by computing the forward image ("image") of the current set of states (starting from the initial states). Backward reachability analysis computes the *pre-image* of the current set of states. One typically starts from the current set of states *violating* the desired property, and attempts to find a path back to the initial state. If such a path exists, it indicates the possibility of a computation that violates the desired property.

Each step in reachability analysis takes the current set of states represented by a BDD and computes the next set of states, also represented by a BDD. It essentially performs a *breadth-first* traversal, generating each breadth-first frontier in one step from the currently reached set of states. The commonly used formulation of traversal is in terms of computing the least fixed-point as explained in Section 11.3.2. When the least fixed-point is reached, one can query it to determine the overall properties of the system. One can also check whether desired system invariants hold in an incremental fashion (without waiting for the fixed-point to be attained) by testing the invariant after each new breadth-first frontier has been generated. Here, an *invariant* refers to a property that is true at every reachable state.

We will now take up these three topics in turn, first illustrating how we are going to represent state transition systems.

11.3.1 Representing state transition systems

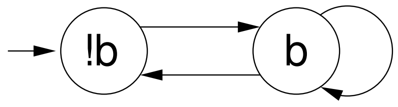


Fig. 11.4. Simple state transition system (example SimpleTR)

We capture transition systems by specifying a binary state transition relation between the *current* and *next* states, and also specifying a predicate capturing the initial states. If inputs and outputs are to be modeled, they are made part of the state vector. Depending on the problem being modeled, we may not care to highlight which parts of the state vector are inputs and which are outputs. In some cases, the entire state of the system will be captured by the states of inputs and outputs. Figure 11.4 presents an extremely simple state transition system, called SimpleTR. Initially, the state is `0`. Whenever the state is `0`, it can become `1`. When it is `1`, it can either stay `1` or become `0`. These requirements can be captured using a single Boolean variable b representing the current state, another Boolean variable b' representing the next state,[8] and an initial state predicate and a state transition relation involving these variables, as follows:

- The initial state predicate for SimpleTR is $\lambda b.\neg b$, since the initial state is `0`. Often, instead of using the lambda syntax, initial state predicates are introduced by explicitly introducing a named initial state predicate I and defining it by an equation such as $I(b) = \neg b$. For brevity,[9] we shall often say "input state represented by $\neg b$."
- The state transition relation for SimpleTR is $\lambda(b, b').\neg b b' + b b' + b\neg b'$, where each product term represents one of the transitions. The values of b and b' for which this relation is satisfied represent the present and next states in our example. In other words,
  - a move where b is false now and true in the next state is represented by $\neg b b'$.
  - a move where b is true in the present and next states is represented by $b b'$.
  - finally, a move where b is true in the present state and false in the next state is represented by $b\neg b'$.

  This expression can be simplified to $\lambda(b, b').(b + b')$. The above relation can also written in terms of a transition relation T defined as $T(b, b') = b + b'$. We shall hereafter say "transition relation $b + b'$." Notice that this transition relation is false for $b = 0$ and $b' = 0$, meaning there is no move from state 0 to itself (all other moves are present).

11.3.2 Forward reachability

The set of reachable states in SimpleTR starting from the initial state $\neg b$ can be determined as follows:

- Compute the set of states in the initial set of states.
- Compute the set of states reachable from the initial states in n steps, for $n = 1, 2, \ldots$.

[8] The 'primed variable' notation was first used by Alan Turing in one of the very first program proofs published by him in [89].

[9] Syntactic sugar can cause cancer of the semi-colon – Perlis

In other words, we can introduce a predicate P such that a state x is in P if and only if it is reachable from the initial state I through a finite number of steps, as dictated by the transition relation T. The above recursive recipe is encoded as

$$P(s) = (I(s) \vee \exists x.(P(x) \wedge T(x,s))).$$

This formula says that s is in P if it is in I, or there exists a state x such that x is in P, and the transition relation takes x to s.
Rewriting the above definition, we have

$$P = \lambda s.(I(s) \vee \exists x.(P(x) \wedge T(x,s))).$$

Rewriting again, we have

$$P = (\lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x,s))))) \; P.$$

In other words, P is a fixed-point of

$$\lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x,s)))).$$

Let us call this Lambda expression H:

$$H = \lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x,s)))).$$

In general, H can have multiple fixed-points. Of these, the *least fixed-point* represents exactly the reachable set of states, as next explained in Section 11.3.3.

11.3.3 Fixed-point iteration to compute the least fixed-point

As shown in Section 6.1, the least fixed-point can be obtained by "bottom refinement" using the functional obtained from the recursive definition. In the same manner, we will determine P, the least fixed-point of H, by computing its approximants that, in the limit, become P. Let us denote the approximants $P_0, P_1, P_2, \ldots$. We have $P_0 = \lambda x.false$, the "everywhere false" predicate. The next approximation to P is obtained by feeding P_0 to the "bottom refiner" (as illustrated in Section 6.1):

$$P_1 = \lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x,s))))P_0$$

which becomes $\lambda s.I(s)$. This approximant says that P is a predicate true of s whenever $I(s)$. While this is not true (P must represent the reachable state set and not the initial state alone), it is certainly a better answer than what P_0 denotes, which is that there are *no* states in the reachable state set! We now illustrate all the steps of this computation, taking SimpleTR for illustration. We use the abbreviation of not showing the lambda abstracted variables in each step.

- $I = \lambda b.\neg b.$
- $T = \lambda(b, b'). \ (b + b').$
- $P_0 = \lambda s.false$, which encodes the fact that "we've reached nowhere yet!"
- $P_1 = \lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x, s))))P_0.$
  This simplifies to $P_1 = I$, which is, in effect, an assertion that we've "just reached" the initial state, starting from P_0.
- Let's see the derivation of P_1 in detail. Expanding T and P_0, we have
  $P_1 = \lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge \ (x + s) \ ))) \ (\lambda x.false).$
- The above simplifies to $\neg b$.
- By this token, we are expecting P_2 to be all states that are zero or one step away from the start state. Let's see whether we obtain this result.
- $P_2 = \lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x, s))))P_1.$
  $= \lambda s.(\neg s \vee \exists x.(\neg x \wedge (x + s))).$
  $= \lambda s.1.$
- This shows that the set of states reached by all the breadth-first frontiers (combined) that are zero and one step away from the start state, includes every state. Another iteration would not change things; the[10] least fixed-point has been reached.

BED Commands for SimpleTR:

The BED commands given in Figure 11.5 compute the reachable set of states using forward reachability in our example. We can see that `P2`, the least fixed-point, is indeed *true* — namely, the characteristic predicate for the set of all states. (*Note: In BED, the primed variables must be declared immediately after the unprimed counterparts*). In addition to the explicit commands to calculate the least fixed-point, BED also provides a single command called `reach`. Using that, one can calculate the least fixed-point in one step. In our present example, `RS` and `P2` end up denoting the BDD for *true.*

```
let RS = reach(I,T)
upall RS
view RS
```

Section 11.3.4 discusses another example where the details of the fixed-point iteration using BED are discussed.

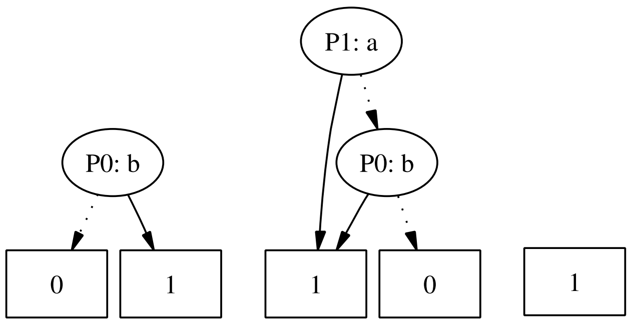
[10] We do not discuss many of the theoretical topics associated with computing fixed-points in the domain of state transition systems — such as why least fixed-points are unique, etc. For details, please see [20].

```
var b bp                 % Declare b and b'
let I = !b               % Declare init state
let t1 = !b and bp       % 0 --> 1
upall t1                 % Build BDD for it
view t1                  % View it
let t2 = b and bp        % 1 --> 1
let t3 = b and !bp       % 1 --> 0
let T = t1 or t2 or t3   % All three edges
upall T                  % Build and view the BDD
view T                   %

let   P0 = false
upall P0
view  P0

let   P1 = I or ((exists b. (P0 and T))[bp:=b])
upall P1
view  P1

let   P2 = I or ((exists b. (P0 and T))[bp:=b])
upall P2
view  P2
```



P0 P1 P2, the least fixed-point

Fig. 11.5. BED commands for reachability analysis on SimpleTR, and the fixed-point iteration leading up to the least fixed-point that denotes the set of reachable states starting from I

Why Stabilization at a Fixed-Point is Guaranteed

In every finite-state system modeled using a finite-state Boolean transition system, the least fixed-point is always reached in a finite number of steps. Let us try to argue this fact first using only a simple observation. The observation is that all the Boolean expressions generated during the course of fixed-point computation are over the same set of variables. Since there are exactly 2^{2^N} Boolean functions over N Boolean variables (see Illustration 4.5.2), eventually two of the approximants in the fixed-point computation process will have to be the same Boolean function. However, this argument does not address whether it is possible to have "evasive" or "oscillatory" approximants $P_i, P_{i+1}, \ldots, P_j$ such that $i \neq j$ and $P_j = P_i$. If this were possible, it would be possible to cycle through $P_i, \ldots, P_j$ without ever stabilizing on a fixed-point. Fortunately, this is not possible! Each approximant P_{i+1} is *more defined* than the previous approximant P_i, in the sense defined by the implication lattice defined in Illustration 4.5.3. With this requirement, the number of these ascending approximants is finite, and one of these would be the least fixed-point. See Andersson's paper [7] for additional examples of forward reachability. The book by Clarke et.al. [20] gives further theoretical insights.

11.3.4 An example with multiple fixed-points

Consider the state transition system in Figure 11.6 with initial state s0 (called MultiFP). The set of its reachable states is simply {s0} (and is characterized by the formula $a \wedge b$), as there is no reachable node from s0. Now, a fixed-point iteration beginning with the initial approximant for the reachable states set to $P0 = false$ will converge to the fixed-point $a \wedge b$. What are the other fixed-points one can attain in this system? Here they are:

- With the initial approximant set to {s0,s1}, which is characterized by b, the iteration would reach the fixed-point of $a \vee b$, which characterizes {s0,s1,s2}.
- Finally, we may iterate starting from the initial approximant being 1, corresponding to {s0,s1,s2,s3}. The fixed-point attained in this case is 1, which happens to be the greatest fixed-point of the recursive equation characterizing reachable states.

Hence, in this example, there are three distinct fixed-points for the recursive formula defining reachable states. Of these, the *least* fixed-point is $a \wedge b$, and truly characterizes the set of reachable states; $a \vee b$ is the intermediate fixed-point, and 1 is the greatest fixed-point. It is clear that $(a \wedge b) \Rightarrow (a \vee b)$ and $(a \vee b) \Rightarrow 1$, which justifies these fixed-point orderings. Figure 11.6 also describes the BED commands to produce this intermediate fixed-point.

```
var a ap b bp

let T = (a  and b  and ap  and bp)  or /* S0 -> S0 */
        (!a and b  and !ap and bp)  or /* S1 -> S1 */
        (a  and !b and ap  and !bp) or /* S2 -> S2 */
        (!a and !b and !ap and !bp) or /* S3 -> S3 */
        (!a and b  and ap  and !bp) or /* S1 -> S2 */
        (a and !b  and !ap and bp)  or /* S2 -> S1 */
        (!a and b  and ap  and bp)  or /* S1 -> S0 */
        (a and !b  and ap  and bp)     /* S2 -> S0 */

upall T
view T             /* Produces BDD for TREL 'T' */

let I = a and b
let P0 = b
let P1 = I or ((exists a. (exists b. (P0 and T)))[ap:=a][bp:=b])

upall P1
view P1
```

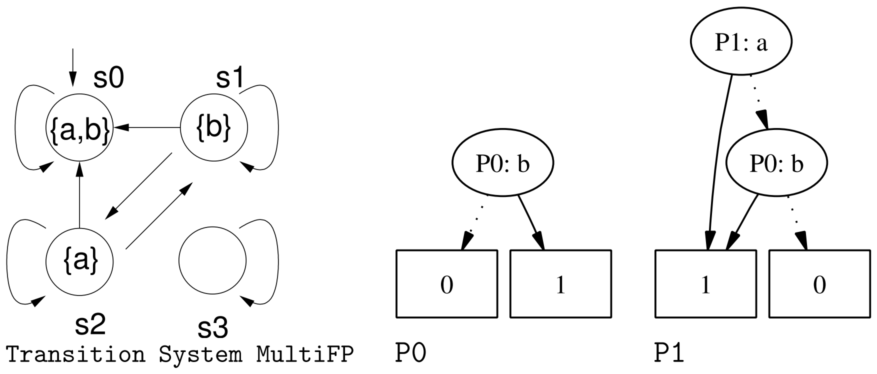


Fig. 11.6. Example where multiple fixed-points exist. This figure shows attainment of a fixed-point $a \vee b$ which is between the least fixed-point of $a \wedge b$ and the greatest fixed-point of 1. The figure shows the initial approximant P0 and the next approximant P1

11.3.5 Playing tic-tac-toe using BDDs

What good are state-space traversal techniques using BDDs? How does one obtain various interesting answers from real-world problems? While we cannot answer these questions in detail, we hope to leave this chapter with a discussion of how one may model a game such as tic-tac-toe and, say, compute the set of all draws in one fell swoop. Following through this example, the reader would obtain a good idea of how to employ mathematical logic to specify a transition system through constraints, and reason about it. We assume the reader knows the game of tic-tac-toe (briefly explained in the passing).

Modeling the players and the board:

We model two players, A and B. The state of the game board is modeled using a pair of variables $a_{i,j}, b_{i,j}$ (we omit the pairing symbols $\langle\rangle$ for brevity) for each square i, j where $i \in 3$ and $j \in 3$. We assume that player A marks square i, j with an o, by setting $a_{i,j}$ and resetting $b_{i,j}$, while player B marks square i, j with an x, by resetting $a_{i,j}$ and setting $b_{i,j}$. We use variable $turn$ to model whose turn it is to play (with $turn = 0$ meaning it is A's turn). The state transition relation for each square will be specified using the four variables $a_{i,j}, a_{i,j}p, b_{i,j}$, and $b_{i,j}p$. We model the conditions for a row or column remaining the same, using predicates $samerow_i$ and $samecol_i$. We define nine possible moves for both A and for B. For example, `M00` model's A's move into cell $0, 0$; Similarly, we employ `N00` to model B's move into cell $0, 0$, and so on for the remaining cells. The transition relation is now defined as a disjunction of the $M_{i,j}$ and $N_{i,j}$ moves. We now capture the constraint *atmostone* that says that, at most one player can play into any square. We then enumerate the gameboard for all possible wins and draws. In the world of BDDs, these computations are achieved through "symbolic breadth first" traversals. We compute the reachable set of states, first querying it to make sure that only the correct states are generated. Then we compute the set of states defining draw configurations. The complete BED definitions are given in Appendix B.

Chapter Summary

This chapter briefly reviewed the history of mathematical logic and pointed out the fact that in the early days of automata theory, mathematical logic and automata were discussed in a unified setting. This approach has immense pedagogical value which this book tries to restore to some extent. A practitioner who works on advanced hardware/software debugging method needs to know *both* of these topics well. For instance, automata theory has, traditionally, been considered an essential prerequisite for an advanced class on compilation. However, recent publications in systems/compilers (e.g., [121]) indicate the central role played by BDDs (see below) and related notions in mathematical logic.

We then discuss how Boolean formulas can be represented in a canonical fashion using the so-called 'reduced ordered binary decision diagrams,' or "BDDs" for short. We then present how finite-state machines can be represented and manipulated using BDDs. We show how reachable states starting from a set of start states can be computed using forward reachability, by using the notion of fixed-points introduced in Chapter 6. We finish the chapter with an illustration of how the game of tic-tac-toe may be modeled using BDDs, and how a tool called `BED` may be used to compute interesting configurations, such as all the *draw* positions, all possible *win* positions, etc.

BDDs are far richer in scope and application than we have room to elaborate here. The reader is referred to [14, 13] for an exposition of how BDDs are used in hardware and software design, how BDDs may be combined using Boolean operations through the *apply* operator, etc. An alternate proof of canonicity of BDDs appears in [14]. Our presentation of BDDs as automata draws from [22], and to some extent from [111].

Exercises

11.1. Similar to Figure 11.3, draw a BDD for all 16 Boolean functions over variables x and y. (Some of these functions are $\lambda(x, y).true$, $\lambda(x, y).false$, $\lambda(x, y).x$, $\lambda(x, y).y$, etc. Down this list, you have more "familiar" functions such as $\lambda(x, y).nand(x, y)$, and so on. Express these functions without the "lambda" part in BED syntax, and generate the BDDs using BED.)

11.2.

1. Obtain an un-minimized DFA (in the form of a binary tree) for the language

$$L = \{abc \mid a \Rightarrow b \wedge c\}$$

   picking the best variable ordering (in case two variable orderings are equal, pick the one that is in lexicographic order). Show the black-hole state also.
2. Minimize this DFA, and then show the additional steps that cast the minimized DFA into a BDD.

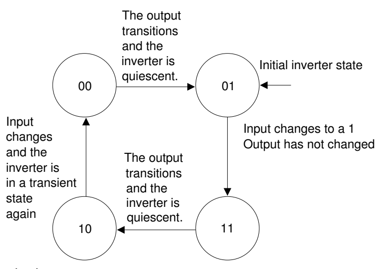
11.3. Consider the examples given in Figure 11.2. Construct similar examples for the *addition* operation. More specifically, consider the binary addition of two unsigned two-bit numbers a_1a_0 and b_1b_0, resulting in answer $c_2c_1c_0$. Generate a BDD for the carry output bit, c_2. Choose a variable ordering that minimizes the size of the resulting BDD and experimentally confirm using BED.

11.4. Repeat Exercise 11.3 to find out the variable ordering that maximizes the BDD size.

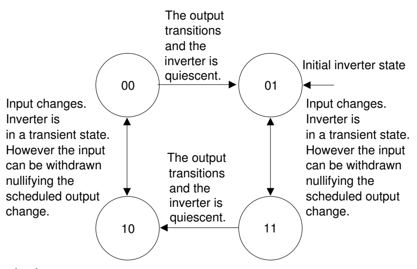
11.5. Represent the behavior of a nand gate, under the inertial delay model, as a state transition system. Encode this transition system using a BDD. Here are some general details on how to approach this problem.

The behavior of an inverter can be modeled using a pair of bits representing its input and output. (For a nand gate, we will need to employ three bits.) In the *transport delay* model, every input change, however short, is faithfully copied to the output, but after a small delay. There is another delay model called the *inertial* delay model in which "short" pulses may not make it to the output.

The behavior of an inverter under these delay models are shown in figures (a) and (b) below.


The output transitions and the inverter is quiescent.
Initial inverter state
00
01
Input changes and the inverter is in a transient state again
Input changes to a 1 Output has not changed
The output transitions and the inverter is quiescent.
10
11

(a)


The output transitions and the inverter is quiescent.
Initial inverter state
00
01
Input changes. Inverter is in a transient state. However the input can be withdrawn nullifying the scheduled output change.
The output transitions and the inverter is quiescent.
Input changes. Inverter is in a transient state. However the input can be withdrawn nullifying the scheduled output change.
10
11

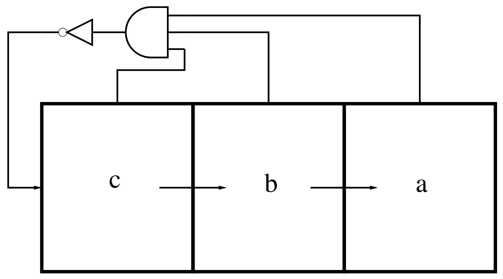
(b)

11.6. Draw a BDD for the transition relation of a two-bit binary counter with output bits a_1a_0 for initial state 00, counting in the usual $0, 1, 2, 3$ order. Repeat for a two-bit gray-code counter that counts 00, 01, 11, 10, and back to 00.

11.7.

1. With respect to the state transition relation of Figure 11.6(a), identify all the fixed-points of the recursive equation for reachability.
2. Given a state transition system (say, as a graph, as in Figure 11.6(a)), what is a general algorithm to determine the number of fixed-points of its recursive equation for reachability?

11.8. Consider a three-bit shift register based counter with the indicated next-state relation for its three bits:



next(a) = b ; next(b) = c ; next(c) = not(and(a,b,c))

1. Represent the next-state relation of this counter using a single ROBDD. Choose a variable ordering that minimizes the size of your ROBDD and justify your choice.
2. Compute the set of all reachable states using forward reachability analysis, using the `reach` command, starting at state `000`.
3. Justify the correctness of the answer you obtain. The answer you obtain must be a Boolean formula over a, b, c. Show that this formula is satisfied exactly for those states reachable by the system.

11.9. A three-bit Johnson counter[11] consists of a three-bit shift register where the final $\overline{Q}$ output is connected to the first D input. Starting from a reset state of 000, this counter will go through the sequence 100, 110, 111, 011, 001, and back to 000. For this counter, repeat what Exercise 11.8 asks.

11.10. Using BED, determine the shortest number of steps to win in Tic-Tac-Toe. Appendix B has a full description of the problem encoding.

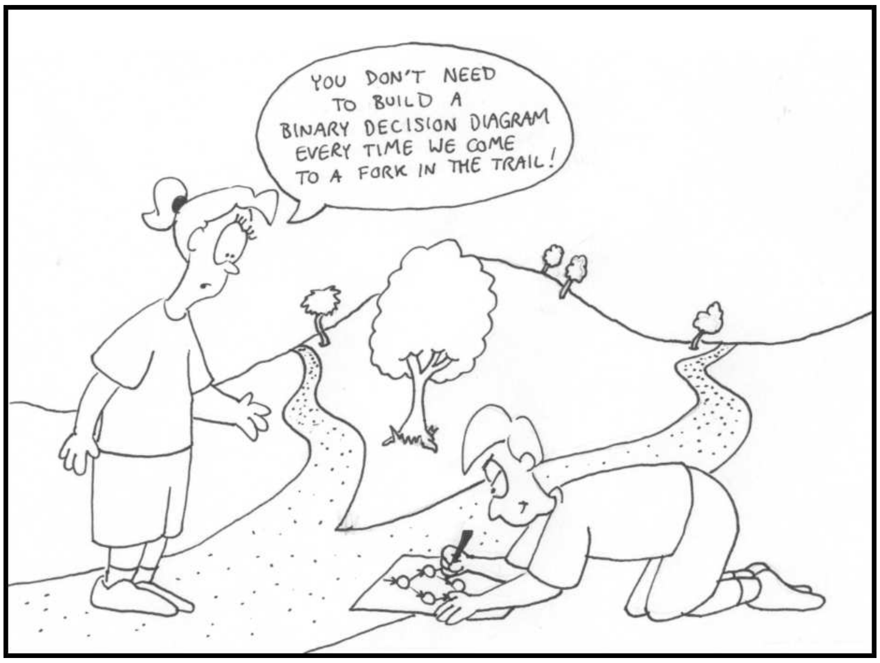
11.11. Check two conjectures concerning Tic-Tac-Toe, using BED: (i) if a player starts by marking the top-left corner, he/she may lose; (ii) if a player starts by marking the middle square, he/she may win.

11.12. Construct an example with four distinct fixed-points under forward reachability, and verify your construction similar to that explained in Figure 11.6.

11.13. Encode the Man-Wolf-Goat-Cabbage problem using BDDs. In this problem, a man has to carry a wolf, goat, and cabbage across a river. The man has to navigate the boat in each direction. He may carry no more than one animal/object on the boat (besides him) at a time. He must not leave the wolf and goat unattended on either bank,

[11] Named after Emeritus Prof. Bob Johnson, University of Utah.

nor must he leave the goat and cabbage unattended on either bank. The number of moves taken is to be minimal. Use appropriate Boolean variables to model the problem.


YOU DON'T NEED TO BUILD A BINARY DECISION DIAGRAM EVERY TIME WE COME TO A FORK IN THE TRAIL!