Appendix C
Model Checking

C.1 Introduction

Model checking is becoming a popular technique that has been successfully applied to the verification of industrial hardware and software systems. It is a formal verification technique based on the exhaustive exploration of the states of a system. Actually, the term *model checking* refers to several different problems, theories, and techniques, e.g., from finite-state to infinite-state model checking, from liner-time to branching-time model checking, from model checking based on automata theory to model checking based on Kripke structures, from explicit-state to symbolic model checking, from symbolic model checking based on BDDs to symbolic model checking based on satisfiability decision procedures.

This appendix focuses on finite-state systems. It describes a formal framework based on Kripke Structures and branching-time temporal logic. We discuss symbolic model checking and its implementation through BDDs because these techniques are of interest for practical applications where model checking has to deal with large state spaces. Indeed, symbolic model checking is routinely applied in industrial hardware design and is beginning to be used in other application domains (see [131] for a survey).

C.2 Intuitions

Intuitively, the model checking problem is the problem of determining whether a property holds in a model of a system. As an example, consider an interlocking system, i.e., a system that controls the equipment in a railway station: the signals to the trains, the position of the switches, the signals at railroad crossings, etc. A desired property is that the interlocking system never allows trains to crash, e.g., by letting a train enter a route crossed by a railroad crossing that is open to cars. The interlocking system can be modeled by a finite-state machine. The property that a train should never crash can be represented in temporal logic. A *model checker* (see Figure C.1) is a software system that takes as input a model of a system and

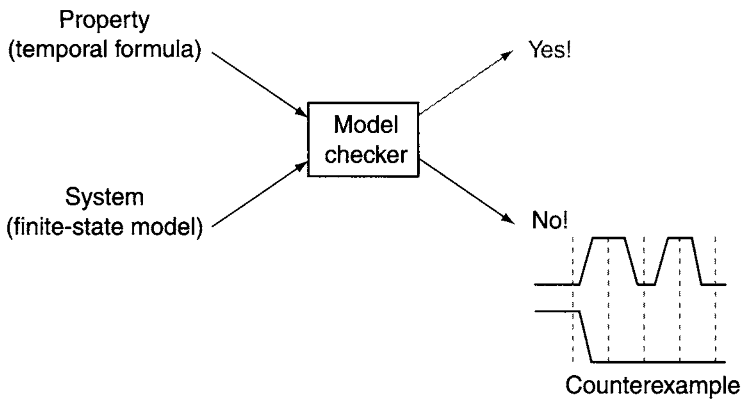

Figure C.1 Model checker.

a description of a property to be checked, and returns "yes" if the property is true in the model. If the property does not hold, the model checker returns a counterexample, i.e., a description of a behavior of the system that does not satisfy the property.

Example C.1 Figure C.2 shows a very simple example of a model of an interlocking system and of some properties that should be checked. The interlocking system can be in three states: (1) the signal to the train is red and the crossing road is closed, (2) the signal is green and the road is closed, and (3) the signal is red and the road

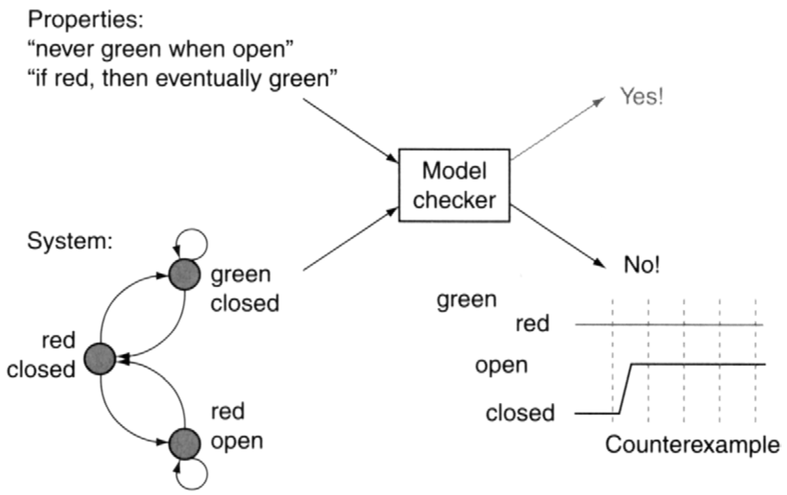

Figure C.2 Examples of model checking.

is open. A safety requirement is that "the signal is never green when the road is open." Given this property as input, the model checker should return "yes." Indeed, there is no state of the system where the road is open and the signal is green.

We are also interested in the *liveness requirements*, i.e., properties that assure that the system "works," such as the fact that "if the signal is red, then eventually in the future it will be green." This corresponds to the fact that we want to allow some train to proceed sooner or later. In this case the model checker finds a counterexample. It is represented by the infinite sequences of states that start with red and closed, and then have red and open forever.

■

Some remarks are in order. When we say that a finite-state model of a system is given as input to the model checker, we do not mean that the data structure is provided explicitly as input to the model checker. In the DWR domain, there are cases with 10^{277} states. In practice, it is not feasible to give the finite-state model explicitly. Instead, different languages are used to describe finite-state models. In this appendix, we will describe the model checking performed on state-transition systems, without discussing the languages used to describe them.

C.3 The Model Checking Problem

In model checking, models of systems can be formalized as Kripke structures. A *Kripke Structure* K is a 4-tuple (S, S_0, R, L), where:

- S is a finite set of *states*.
- $S_0 \subseteq S$ is a set of initial states.
- $R \subseteq S \times S$ is a binary relation on S, the *transition relation*, which gives the possible transitions between states. We require R to be total, i.e., for each state $s \in S$ there exists a state $s' \in S$ such that $(s, s') \in R$.
- $L : S \mapsto 2^{\mathcal{P}}$ is a *labeling function*, where $\mathcal{P}$ is a set of proposition symbols. L assigns to each state the set of atomic propositions true in that state.

A Kripke structure encodes each possible evolution of the domain (or behavior) as a *path*, i.e., an infinite sequence $s_0, s_1, s_2, \ldots$ of states in S such that $s_0 \in S_0$ and for each i, $(s_i, s_{i+1}) \in R$.

Example C.2 Figure C.3 shows a simple example of a Kripke structure.

$S = \{s_0, s_1, s_2\}$
$S_0 = \{s_0\}$
$R = \{(s_0, s_1), (s_0, s_2), (s_1, s_1), (s_1, s_0), (s_2, s_2), (s_2, s_0)\}$
$L(s_0) = \emptyset$, $L(s_1) = \text{green}$, $L(s_2) = \text{open}$

■

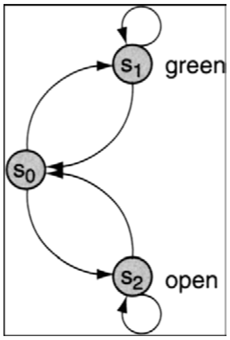

Figure C.3 Example of a Kripke structure.

Properties to be checked can be formalized in a temporal logic called Computation Tree Logic (CTL) [171]. Given a finite set $\mathcal{P}$ of atomic propositions, CTL formulas are inductively defined as follows.

1. Every atomic proposition $p \in \mathcal{P}$ is a CTL formula.
2. If p and q are CTL formulas, then so are the following formulas.
   (a) $\neg p, p \vee q$
   (b) $\mathbf{AX}p, \mathbf{EX}p$
   (c) $\mathbf{A}(p\mathbf{U}q)$, and $\mathbf{E}(p\mathbf{U}q)$

X is the "next time" temporal operator and **A** and **E** are path quantifiers; the formula **AX**p or **EX**p means that p holds in every or in some immediate successor of the current state. **U** is the "until" temporal operator; the formula $\mathbf{A}(p\mathbf{U}q)$ or $\mathbf{E}(p\mathbf{U}q)$ means that for every path or for some path there exists an initial prefix of the path such that q holds at the last state of the prefix and p holds at all the other states along the prefix.[1] Formulas **AF**p and **EF**p (where the temporal operator **F** stands for "future" or "eventually") are abbreviations of $\mathbf{A}(\top\mathbf{U}p)$ and $\mathbf{E}(\top\mathbf{U}p)$ (where $\top$ stands for truth), respectively. **EG**p and **AG**p (where **G** stands for "globally" or "always") are abbreviations of $\neg\mathbf{AF}\neg p$ and $\neg\mathbf{EF}\neg p$, respectively.

Example C.3 Consider the example of the Kripke structure in Figure C.3. Recall the safety and the liveness requirements expressed in Example C.1 and Figure C.2. To represent the safety requirement "the signal is never green when the road is open," let us use the formula **AG**(open → ¬green). Consider the liveness requirement "if the signal is red, then eventually it will be green." If we want to say that the signal eventually will be green for *all* of the system's evolutions, then we write the formula **AG**(¬green → **AF**green). If we want the weaker requirement that *there exists*

1. Given a path $s_0, s_1, \ldots$, a prefix of the path is any sequence $s_0, \ldots, s_i$, with $i \geq 0$.

a system's evolution such that the signal will eventually be green, then the formula is **AG**(¬green → **EF**green). Consider the formulas ¬green → **AF**green and ¬green → **EF**green. They state that we must start in one of the two states s_0 and s_2 (where the signal is red), and then we will eventually reach a state where the signal is green.

■

We give semantics to CTL formulas in terms of Kripke Structures. Let p be a CTL formula. $K, s \models p$ is defined inductively as follows.

If $p \in \mathcal{P}$, then $K, s \models p$ iff $p \in L(s)$

$K, s \models \neg p$ iff $K, s \not\models p$

$K, s \models p \vee q$ iff $K, s \models p$ or $K, s \models q$

$K, s \models \mathbf{AX}p$ iff for all paths $\pi = s_0, s_1, s_2, \ldots$ such that $s = s_0$, we have $K, s_1 \models p$

$K, s \models \mathbf{EX}p$ iff there exists a path $\pi = s_0, s_1, s_2, \ldots$ such that $s = s_0$, such that $K, s_1 \models p$

$K, s \models \mathbf{A}(p\mathbf{U}q)$ iff for all paths $\pi = s_0, s_1, s_2, \ldots$ such that $s = s_0$, there exists $i \geq 0$ such that $K, s_i \models q$ and for all $0 \leq j < i$, $K, s_j \models p$

$K, s \models \mathbf{E}(p\mathbf{U}q)$ iff there exists a path $\pi = s_0, s_1, s_2, \ldots$ such that $s = s_0$, and a number $i \geq 0$ such that $K, s_i \models q$ and for all $0 \leq j < i$, $K, s_j \models p$

We say that p is true in a state s of the Kripke structure K if $K, s \models p$. We say that p is true in K ($K \models p$) if $K, s \models p$ for each $s \in S_0$.

The *model checking problem* for a CTL formula p and a Kripke structure K is the problem of determining whether p is true in K.

Example C.4 In the example in Figure C.3, it follows that $K \models$ **AG**(open → ¬green), $K \not\models$ **AG**(¬green → **AF**green), and $K \models$ **AG**(¬green → **EF**green).

■

C.4 Model Checking Algorithms

Algorithms for model checking exploit the structure of CTL formulas. For instance, an atomic formula p is model checked by verifying that $p \in L(s)$ for all $s \in S_0$. As another example, model checking **AX**p or **EX**p is performed by model checking p in all states or in some state s' such that $(s, s') \in R$, for each $s \in S_0$. As a further example, **A**(p**U**q) or **E**(p**U**q) can be model checked by exploiting the fact that:

$$\begin{aligned} p\mathbf{U}q = q \vee \\ & (p \wedge \mathbf{X}q) \vee \\ & (p \wedge \mathbf{X}p \wedge \mathbf{XX}q) \vee \\ & \ldots \end{aligned}$$

```
1. MCheckEF($p$,$K$)
2.   $CurrentStates \leftarrow \emptyset$;
3.   $NextStates \leftarrow$ States($p$,$K$);
4.   while $NextStates \neq CurrentStates$ do
5.     if ($S_0 \subseteq NextStates$)
6.       then return($True$);
7.     $CurrentStates \leftarrow NextStates$;
8.     $NextStates \leftarrow NextStates \cup$ pre-img-ef($NextStates$,$K$);
9.   return($False$);
```

Figure C.4 Model checking $\mathbf{EF}p$.

As a simple example, we show in Figure C.4 a possible algorithm for model checking the CTL formula $\mathbf{EF}p$, with $p \in \mathcal{P}$. Given a Kripke Structure $K = (S, S_0, R, L)$ and a propositional formula p, the algorithm starts by computing the set of states where p holds (line 3):

$$\textsc{States}(p, K) = \{s \in S : p \in L(s)\} \tag{C.1}$$

Next, MCheckEF explores the state space of K. It accumulates in *NextStates* the states returned by pre-img-ef (line 8). Given a set of states $States \subseteq S$, pre-img-ef returns the set of states that have at least one immediate successor state in *States*:

$$\textsc{pre-img-ef}(States, K) = \{s \in S : \exists s'. (s' \in States \wedge R(s, s'))\} \tag{C.2}$$

Notice that $\mathbf{EF}p$ always holds in each state in *NextStates*. The loop terminates successfully if *NextStates* contains all the initial states (termination condition at line 5). MCheckEF returns *False* if *NextStates* does not contain all the initial states and there are no new states to explore, i.e., $NextStates = CurrentStates$. Indeed, if this happens, then $K \not\models p$ because there exists $s \in S_0$ such that $K, s \not\models p$ and we have no further states to explore. It can be shown that there exists a least fixed point and that the algorithm is guaranteed to terminate.

A possible algorithm for model checking the CTL formula $\mathbf{AF}p$ can be obtained from MCheckEF simply by replacing pre-img-ef with pre-img-af, where:

$$\textsc{pre-img-af}(States, K) = \{s \in S : \forall s'. (R(s, s') \rightarrow s' \in States)\} \tag{C.3}$$

Compare this algorithm with the Strong-Plan algorithm in Chapter 17.

Example C.5 Let us model check the formula **EF**green in the Kripke Structure in Figure C.3. MCheckEF assigns state s_1 to the variable *NextStates*. After the first iteration, *NextStates* is $\{s_0, s_1\}$, and then the algorithm stops returning *True*.

As another example let us check the formula **AF**green. Initially s_1 is assigned to the variable *NextStates*, but at the first iteration PRE-IMG-AF returns the empty set, then the loop terminates and the algorithm returns *False*.

■

C.5 Symbolic Model Checking

Most often, realistic models of systems need huge numbers of states. For example, an interlocking system may have something like 10^{200} states. Symbolic model checking [99] has been devised to deal with large state spaces. It is a form of model checking in which propositional formulas are used for the compact representation of finite-state models, and transformations over propositional formulas provide a basis for efficient exploration of the state space. Most often, the use of symbolic techniques allows for the analysis of large systems, even systems with 10^{200} states [99]. The fundamental ideas of symbolic model checking are the following.

1. Model checking is performed by exploring sets of states, rather than single states.
2. The model checking problem is represented symbolically: the sets of states and the transitions over sets of states are represented by logical formulas.

In order to represent a model checking problem symbolically, we need to represent symbolically the sets of states of a Kripke Structure, its transition relation, and the model checking algorithms.

Symbolic Representation of Sets of States. A vector of distinct propositional variables $\boldsymbol{x}$, called *state variables*, is devoted to the representation of the states of a Kripke structure. Each of these variables has a direct association with a proposition symbol of $\mathcal{P}$. Therefore, in the rest of this section we will not distinguish between a proposition and the corresponding propositional variable. For instance, in the Kripke structure in Example C.2 (see page 563), $\boldsymbol{x}$ is the vector $\langle \mathsf{green}, \mathsf{open} \rangle$. A state is the set of propositions of $\mathcal{P}$ that hold in the state. For each state s, there is a corresponding assignment of truth values to the state variables in $\boldsymbol{x}$: each variable in s is *True*, and all other variables are *False*. We represent s with a propositional formula $\xi(s)$, whose unique satisfying assignment of truth values corresponds to s. For instance, the formula representing state s_1 in Example C.2 is $\mathsf{green} \wedge \neg \mathsf{open}$. This representation naturally extends to any set of states $Q \subseteq S$ as follows:

$$\xi(Q) = \bigvee_{s \in Q} \xi(s)$$

That is, we associate a set of states with the disjunction of the formulas representing each of the states. The satisfying assignments of $\xi(Q)$ are exactly the assignments representing the states of Q.

A remark is in order. We are using a propositional formula to represent the set of assignments that satisfy it (and hence to represent the corresponding set of states), but we do not care about the actual syntax of the formula used, and thus in the following discussion we will not distinguish among equivalent formulas that represent the same sets of assignments. Although the actual syntax of the formula may have a computational impact, in the next section we will show that the use of formulas for representing sets of states is indeed practical.

The main efficiency of the symbolic representation is that the cardinality of the represented set is not directly related to the size of the formula. For instance, $\xi(2^{\mathcal{P}})$ and $\xi(\emptyset)$ are the formulas *True* and *False*, respectively, independent of the cardinality of $\mathcal{P}$.

As a further advantage, the symbolic representation can provide an easy way to ignore irrelevant information. For instance, notice that the variable open does not appear in the formula $\xi(\{s_0, s_2\}) = \neg\text{green}$. For this reason, a symbolic representation can have a dramatic improvement over an explicit state representation that enumerates the states of the Kripke Structure. This is what allows symbolic model checkers to handle finite-state models that have very large numbers of states (see, e.g., [99]).

Another advantage of the symbolic representation is the natural encoding of set-theoretic transformations (e.g., union, intersection, complementation) into propositional operations, as follows:

$$\begin{aligned}\xi(Q_1 \cup Q_2) &= \xi(Q_1) \vee \xi(Q_2)\\ \xi(Q_1 \cap Q_2) &= \xi(Q_1) \wedge \xi(Q_2)\\ \xi(S - Q) &= \xi(S) \wedge \neg\xi(Q)\end{aligned}$$

Symbolic Representation of Transition Relations. We represent transition relations through the vector of state variables $\boldsymbol{x} = \langle x_1, \ldots, x_n\rangle$ and a further vector $\boldsymbol{x}' = \langle x'_1, \ldots, x'_n\rangle$ of propositional variables, called *next-state variables.* We write $\xi'(s)$ for the representation of the state s in the next-state variables. $\xi'(Q)$ is the formula corresponding to the set of states Q. In the following, $\Phi[\boldsymbol{x} \leftarrow \boldsymbol{y}]$ is the parallel substitution in formula Φ of the variables in vector $\boldsymbol{x}$ with the corresponding variables in $\boldsymbol{y}$. We define the representation of a set of states in the next variables as follows:

$$\xi'(s) = \xi(s)[\boldsymbol{x} \leftarrow \boldsymbol{x}'].$$

We call the operation $\Phi[\boldsymbol{x} \leftarrow \boldsymbol{x}']$ *forward shifting* because it transforms the representation of a set of current states in the representation of a set of next states. The dual operation $\Phi[\boldsymbol{x}' \leftarrow \boldsymbol{x}]$ is called *backward shifting.* In the following, we call the variables in $\boldsymbol{x}$ the *current-state variables* to distinguish them from the next-state variables.

For the interlocking example in Figure C.3, the single transition from state s_0 to state s_1 is represented by the formula

$$\xi(\langle s_0, s_1\rangle) = \xi(s_0) \wedge \xi'(s_1),$$

that is,

$$\xi(\langle s_0, s_1\rangle) = (\neg\mathsf{green} \wedge \neg\mathsf{open}) \wedge (\mathsf{green}' \wedge \neg\mathsf{open}')$$

The transition relation R of a Kripke Structure is a set of transitions and is thus represented by the formula in the variables of $\boldsymbol{x}$ and of $\boldsymbol{x}'$,

$$\xi(R) = \bigvee_{r \in R} \xi(r),$$

in which each satisfying assignment represents a possible transition.

Symbolic Representation of Model Checking Algorithms. In order to make explicit that the formula $\xi(Q)$ contains the variables $x_1, \ldots, x_n$ of $\boldsymbol{x}$, in the following we will use the expression $Q(\boldsymbol{x})$ to mean $\xi(Q)$. Similarly, we will use the expression $Q(\boldsymbol{x}')$ to mean $\xi'(Q)$. Let $S(\boldsymbol{x})$, $R(\boldsymbol{x}, \boldsymbol{x}')$, and $S_0(\boldsymbol{x})$ be the formulas representing the states, the transition relation, and the initial states of a Kripke structure, respectively.

In the following, we will use quantification in the style of the logic of Quantified Boolean Formulas (QBFs). QBFs are a definitional extension to propositional logic, in which propositional variables can be universally and existentially quantified. If Φ is a formula and v_i is one of its variables, then the existential quantification of v_i in Φ, written $\exists v_i. \Phi(v_1, \ldots, v_n)$, is equivalent to $\Phi(v_1, \ldots, v_n)[v_i \leftarrow \mathit{False}] \vee \Phi(v_1, \ldots, v_n)[v_i \leftarrow \mathit{True}]$. Analogously, the universal quantification $\forall v_i. \Phi(v_1, \ldots, v_n)$ is equivalent to $\Phi(v_1, \ldots, v_n)[v_i \leftarrow \mathit{False}] \wedge \Phi(v_1, \ldots, v_n)[v_i \leftarrow \mathit{True}]$. QBFs allow for an exponentially more compact representation than propositional formulas.

The symbolic representation of the *image* of a set of states Q, i.e., the set of states reachable from any state in Q with one state transition, is the result of applying the substitution $[\boldsymbol{x}' \leftarrow \boldsymbol{x}]$ to the formula $\exists \boldsymbol{x}. (R(\boldsymbol{x}, \boldsymbol{x}') \wedge Q(\boldsymbol{x}))$:

$$(\exists \boldsymbol{x}. (R(\boldsymbol{x}, \boldsymbol{x}') \ \wedge \ Q(\boldsymbol{x})))[\boldsymbol{x}' \leftarrow \boldsymbol{x}]$$

Notice that, with this single operation, we symbolically simulate the transition from any of the states in Q. The dual backward image is the following:

$$(\exists \boldsymbol{x}'. (R(\boldsymbol{x}, \boldsymbol{x}') \ \wedge \ Q(\boldsymbol{x}')))$$

From the definition of PRE-IMG-EF(Q) (see Equation C.2), we have therefore that ξ(PRE-IMG-EF(Q)) is:

$$\exists \boldsymbol{x}'. (R(\boldsymbol{x}, \boldsymbol{x}') \wedge Q(\boldsymbol{x}'))$$

while ξ(PRE-IMG-AF(Q)) (see Equation C.3) is:

$$\forall \boldsymbol{x}'. (R(\boldsymbol{x}, \boldsymbol{x}') \rightarrow Q(\boldsymbol{x}'))$$

In both cases, the resulting formula is obtained as a one-step computation and can often describe compactly a large set of states.

Given the basic building blocks just defined, the algorithms presented in the previous section can be symbolically implemented by replacing, within the same control structure, each function call with the symbolic counterpart and by casting the operations on sets into the corresponding operations on propositional formulas.

C.6 BDD-Based Symbolic Model Checking

BDDs provide a way to implement the symbolic representation mechanisms presented in the previous section (e.g., tautology checking, quantification, shifting).

A BDD is a directed acyclic graph (DAG). The terminal nodes are either *True* or *False* (alternatively indicated with 0 and 1, respectively). Each nonterminal node is associated with a Boolean variable and with two BDDs that are called the *left* and *right branches.* Figure C.5 shows some simple BDDs for the interlocking example. At each nonterminal node, the right or left branch is depicted as a solid or dashed line and represents the assignment of the value *True* or *False* to the corresponding variable.

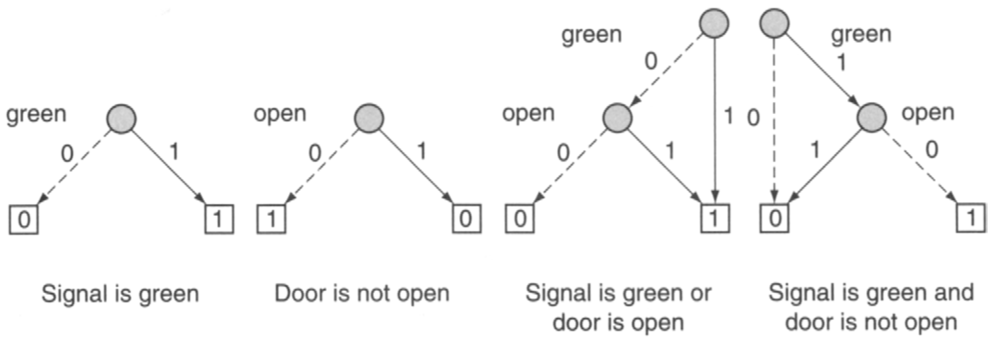

Figure C.5 BDDs for the Boolean formulas green, ¬open, green ∨ open, and green ∧ ¬open.

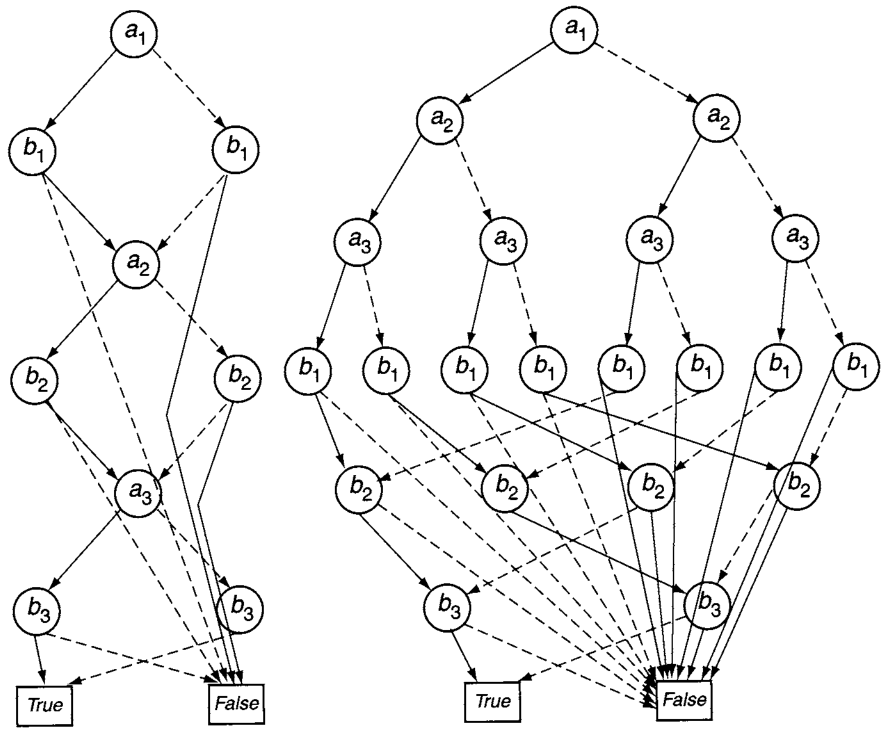

Figure C.6 Two BDDs for the formula $(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$.

Given a BDD, the value corresponding to a given truth assignment to the variables is determined by traversing the graph from the root to the leaves, following each branch indicated by the value assigned to the variables. A path from the root to a leaf can visit nodes associated with a subset of all the variables of the BDD. The reached leaf node is labeled with the resulting truth value. If v is a BDD, its size $|v|$ is the number of its nodes. If n is a node, we use $var(n)$ to denote the variable indexing node n. BDDs are a canonical representation of Boolean formulas if (1) there is a total order $<$ over the set of variables used to label nodes, such that for any node n and respective nonterminal child m, their variables must be ordered, i.e., $var(n) < var(m)$; and (2) the BDD contains no subgraphs that are isomorphic to the BDD itself.

The choice of variable ordering may have a dramatic impact on the dimension of a BDD. For example, Figure C.6 depicts two BDDs for the same formula $(a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2) \wedge (a_3 \leftrightarrow b_3)$ obtained with different variable orderings.

BDDs can be used to compute the results of applying the usual Boolean operators. Given a BDD that represents a formula, it is possible to transform it to obtain the BDD representing the negation of the formula. Given two BDDs representing two formulas, it is possible to combine them to obtain the BDD representing the conjunction or the disjunction of the two formulas. For instance, Figure C.5 shows how the BDD representing the formula green ∧ ¬open can be obtained from the BDDs representing the formulas green and ¬open. Substitution and quantification on Boolean formulas can also be performed as BDD transformations.