

Chapter 2

Safety Requirements

A reactive component interacts with the environment via inputs and outputs. A requirement for a component is a constraint on the possible input/output sequences. Requirements can be classified in two broad categories: *safety* requirements assert that “nothing bad ever happens,” and *liveness* requirements assert that “something good eventually happens.” For instance, in the leader election problem of Section 1.2.4. the main safety requirement is that no two nodes should ever declare themselves to be the leaders, and the main liveness requirement is that the protocol should eventually make one node to be the leader with the remaining nodes declaring themselves to be followers. The analysis problem is to check whether a specific implementation of the leader election, such as the one described in Figure 1.23. meets these requirements. For safety requirements, violation of a requirement can be demonstrated by a finite execution that illustrates the undesirable behavior. Typically, such requirements are captured by composing the system with a *monitor* that observes the inputs and outputs of the system, and enters an *error* state if an undesirable behavior is detected. The safety analysis problem then reduces to checking whether there is a possible execution of the system that leads the monitor to an error state. In this chapter, we will study how reachability problems can be used to formalize safety requirements, and analysis techniques for solving the reachability problems.

2.1 Safety Specifications

2.1.1 Invariants for Transition Systems

An invariant for a system classifies its states into *safe* and *unsafe*, and asserts that during the execution, no unsafe state is encountered. Since the concept of invariants, and tools for establishing invariants, do not specifically rely on the synchronous nature of interaction of reactive components, let us study it in a more general context of *(symbolic) transition systems*. A (symbolic) transition system is specified by variables, whose valuations describe possible states of the

system. Initialization of the system variables is specified by an initialization expression over the state variables. Transitions of the system describing how the state evolves, are specified using a Boolean expression over unprimed and primed copies of state variables.

SYMBOLIC TRANSITION SYSTEM

A (*symbolic*) *transition system* T has a finite set S of typed state variables, a Boolean expression $Init$ over S , and a Boolean expression $Trans$ over $S \cup S'$. A *state* $s \in Q_S$ of T is a valuation over S , a state $s \in Q_S$ is an *initial state* of T if s satisfies $Init$, and for states s and t , $s \rightarrow t$ is a *transition* of T if $[s, t']$ satisfies $Trans$.

With each synchronous reactive component $C = (I, O, S, Init, React)$, there is a naturally associated transition system with state variables S , initialization expression $Init$, and transition expression $\exists I. \exists O. React$. For example, consider the component `TriggeredCopy` (see Figure 1.4). The corresponding transition system has one state variable x of type `nat` with initialization specified by the expression $x = 0$. The transition expression is

$$\exists in, out. [out = (in? \rightarrow in \mid \perp) \wedge x' = (in? \rightarrow x + 1 \mid x)],$$

which simplifies to

$$(x' = x + 1) \vee (x' = x).$$

Sequential programs can also be modeled as transition systems. Consider the classical Euclid's algorithm for computing the greatest common divisor (GCD) of two natural numbers. To compute the GCD of given input numbers m and n , the algorithm initializes x to m and y to n , and repeatedly executes the following:

if $x > y$, decrement x by y ;
 if $y > x$, decrement y by x ;
 if x and y are equal, it is the desired answer, and stop.

This is captured by the transition system $GCD_{m,n}$, whose description is parameterized by the numbers m and n . It has 2 state variables x and y of type `nat`, initialization expression is $x = m \wedge y = n$, and the transition expression is

$$x' = [(x > y) \rightarrow x - y \mid x] \wedge y' = [(y > x) \rightarrow y - x \mid y].$$

An execution of a transition system starts in an initial state, and proceeds by following the transitions specified by $Trans$. States encountered during executions are *reachable* states of the system.

REACHABLE STATE OF TRANSITION SYSTEM

An *execution* of a transition system T consists of a finite sequence of the form s_0, s_1, \dots, s_k such that (1) for $0 \leq j \leq k$, each s_j is a state of T , (2) s_0 is an initial state of T , and (3) for $1 \leq j \leq k$, $s_{j-1} \rightarrow s_j$ is a transition of T . For such an execution, the state s_k is said to be a *reachable* state of T .

For a transition system T , an invariant φ is specified as a property, that is, as a Boolean expression over its state variables, and the invariant φ holds if all the reachable states of the system satisfy the invariant.

INVARIANT OF TRANSITION SYSTEM

For a transition system T , a *property* φ of T is a Boolean expression over its state variables S . The property φ is an *invariant* of T if every reachable state s of T satisfies φ .

Let us revisit the program computing the GCD of two natural numbers. The following property is an invariant of the transition system $\text{GCD}_{m,n}$:

$$\text{gcd}(m, n) = \text{gcd}(x, y),$$

where gcd represents the mathematical function that returns the greatest common divisor of its two arguments. The invariant captures the fact during the execution of the program, even though the values of the state variables x and y change, their gcd stays the same.

The invariant analysis problem is the following: given a symbolic transition system T and a property φ , check whether φ is an invariant of the system T . If it is not an invariant, then there must be some state s such that s is reachable and violates the property φ . In such a case, for debugging purposes, the analysis technique should produce an execution of T that leads to s . Such an execution is called a *counter-example*.

2.1.2 Role of Invariants in Controller Design

To illustrate the use of invariants as safety requirements in design of embedded controllers, let us consider a system of traffic lights for a railroad.

Specification of the railroad controller

Figure 2.1 shows two circular railroad tracks, one for trains that travel clockwise, and the other for trains that travel counterclockwise. At one place in the circle, there is a bridge which is not wide enough to accommodate both tracks. The two tracks merge on the bridge, and for controlling the access to the bridge, there is a signal at either entrance. If the signal at the western entrance is green, then a train coming from the west may enter the bridge; if the signal is red, the train must wait. The signal at the eastern entrance to the bridge controls trains coming from the east in the same fashion.

A train is modeled by the component `Train` in Figure 2.2. The state of the train indicates whether the train is away from the bridge, waiting at the signal, or on the bridge. We use nondeterminism to model the assumption that the train can be away for an unknown period of time: when the train is away, either the state stays unchanged, or the train issues an `arrive` event on its output to the

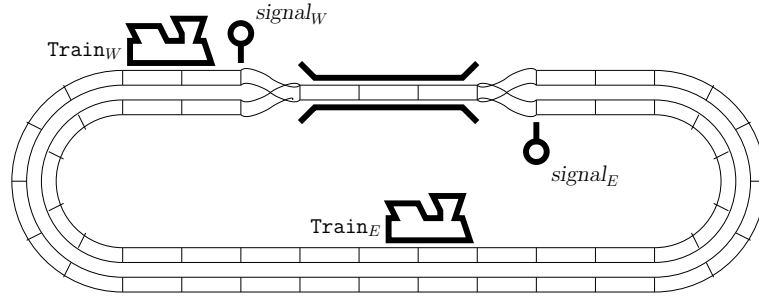


Figure 2.1: Railroad controller example

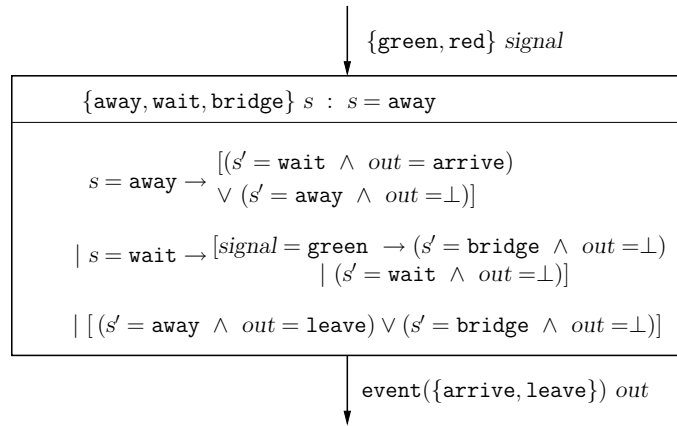


Figure 2.2: Modeling the train as a nondeterministic reactive component

railroad controller and updates the state to waiting. When the train is waiting, it checks the signal. When the signal is red, the train keeps waiting, and when the signal is green, the train proceeds onto the bridge. The train can stay on the bridge for an arbitrary number of rounds. When the train exits from the bridge, it outputs a **leave** event to the controller on its output and updates the state to **away**. There are two trains, one traveling clockwise and the other traveling counterclockwise, we create two instances of the train component, Train_W and Train_E .

We are asked to design a deterministic controller that prevents collisions between the two trains by ensuring that at all times, at most one train is on the bridge. That is, we want to design a deterministic synchronous reactive component Controller with input event variables out_W and out_E , and with output variables signal_W and signal_E . When composed with the models of the trains,

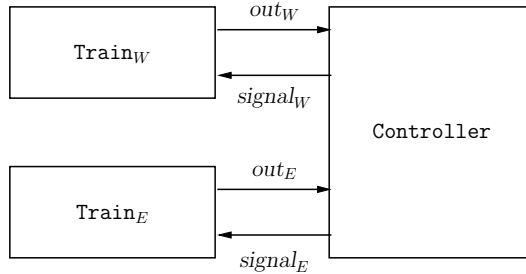


Figure 2.3: Composite system for the railroad controller

we get the composite system `RailRoadSystem`

$$\text{Controller} \parallel \text{Train}_W \parallel \text{Train}_E$$

shown in Figure 2.3. The controller should be designed so that the property

$$\text{TrainSafety} : \neg (\text{Train}_W.s = \text{bridge} \wedge \text{Train}_E.s = \text{bridge})$$

is an invariant of `RailRoadSystem`.

A first attempt at the design of the railroad controller

Figure 2.4 shows a first attempt at designing the railroad controller. The controller maintains two state variables `west` and `east` for the states of the two output signals `signalW` and `signalE`, respectively: in each round, the output variable is set to the value of the corresponding state variable. Initially, both signals are green. A signal turns red whenever a train approaches the opposite entrance to the bridge, and it turns back to green whenever that train exits from the bridge. If both trains approach the bridge in the same round, then only the west signal turns red, giving priority to the train approaching from east.

Unfortunately, the resulting railroad system

$$\text{RailRoadSystem1} = \text{Controller1} \parallel \text{Train}_W \parallel \text{Train}_E$$

does not satisfy the desired invariant `TrainSafety`. This is evidenced by the counter-example shown in Figure 2.5, which leads to a state with both trains on the bridge. If both trains approach the bridge simultaneously, then the east train is admitted to the bridge with the west signal red and the east signal green. When the east train exits from the bridge, the west signal turns green, allowing the west train proceed to the bridge. However, the east signal is still green. So if the east train returns while the west train is still on the bridge, the two trains will collide.

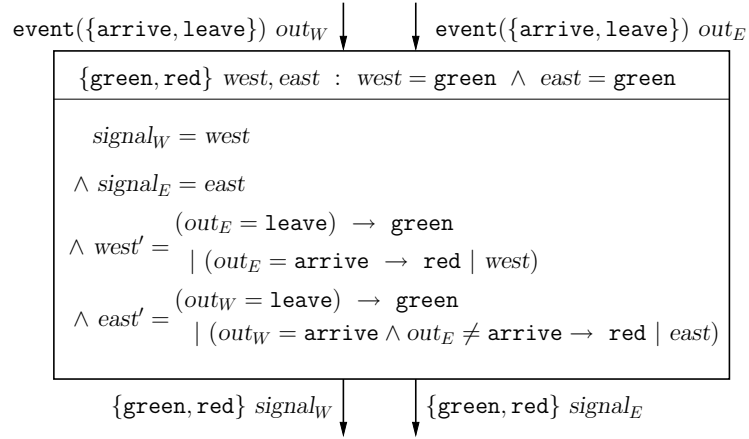


Figure 2.4: A first attempt at design of the railroad controller

A second attempt at the design of the railroad controller

Figure 2.6 shows another attempt at designing the controller. The controller `Controller2`, in addition to the state variables `east` and `west` for the signals, maintains Boolean state variables `near_W` and `near_E` to keep track of whether the respective trains need to use the bridge. Initially, `near_W` is 0. When the west train arrives near the bridge, it is updated to 1, and when the west train leaves the bridge, it is reset to 0. Observe that the state variable `s_W` of `Train_W` is **away** precisely when `near_W` is 0. Analogously, the variable `near_E` keeps track of the status of the east train.

The controller `Controller2` plays it safe by keeping the two signals red by default. Initially, the state variables `east` and `west` for the signals are red. When a train is away (as indicated by the corresponding `near` variable), the corresponding signal variable is set to red. When the east train is near, the east signal is turned green, provided the west signal is red. When both signals are red, and both trains issue `arrive`, thereby setting both `near` variables, the east train gets a preference: the variable `east` is changed to green, and this blocks the update of `west`, which is turned green only if the updated value of `east` is red.

For the composite system `RailRoadSystem2`, that is, `Controller2` \parallel `Train_W` \parallel `Train_E` the property `TrainSafety` is indeed an invariant.

2.1.3 Safety Monitors

For our railroad control example, suppose we have an additional “fairness” requirement that if a train arrives at a bridge, and waits for its signal to turn green, the other train should not be allowed to enter the bridge repeatedly. More

<i>west</i>	<i>east</i>	<i>Train_W.s</i>	<i>Train_E.s</i>	<i>signal_W</i>	<i>signal_E</i>	<i>out_W</i>	<i>out_E</i>
green	green	away	away				
				green	green	arrive	arrive
red	green	wait	wait				
				red	green	⊥	⊥
red	green	wait	bridge				
				red	green	⊥	leave
green	green	wait	away				
				green	green	⊥	arrive
green	green	bridge	wait				
				green	green	⊥	⊥
green	green	bridge	bridge				

Figure 2.5: An execution of `RailroadSystem1` that violates `TrainSafety`

specifically, while a train is waiting with its signal red, the other train should not leave the bridge twice. This is clearly a requirement regarding the sequence of inputs and outputs of the railroad system, but it cannot be formulated as an invariant verification problem directly. It can, however, be stated as an invariant verification problem if we add another component, `WestFairMonitor`, shown in Figure 2.7.

The state variable s is initially 0. When the west train arrives, it changes to 1. If the east train leaves the bridge it changes to 2, and if the east train leaves the bridge again, it changes to 3. In state 1 or 2, if the west signal is turned green, the state is reset to 0. An execution in which the monitor state gets updated to 3, there is a violation of the desired fairness requirement with respect to the west train. To check if there is a violation, we can check whether the property `WestFairMonitor.s ≠ 3` is an invariant of the composite system `RailroadSystem || WestFairMonitor`. To ensure fairness with respect to the east train, we can compose the system with an appropriately instantiated version of the monitor.

In general, a *safety monitor* for a reactive component C consists of another component that observes the behavior of the component C , without influencing it, together with an expression over its state variables that we expect to be an invariant of the system consisting of the composition of the component C and the monitor.

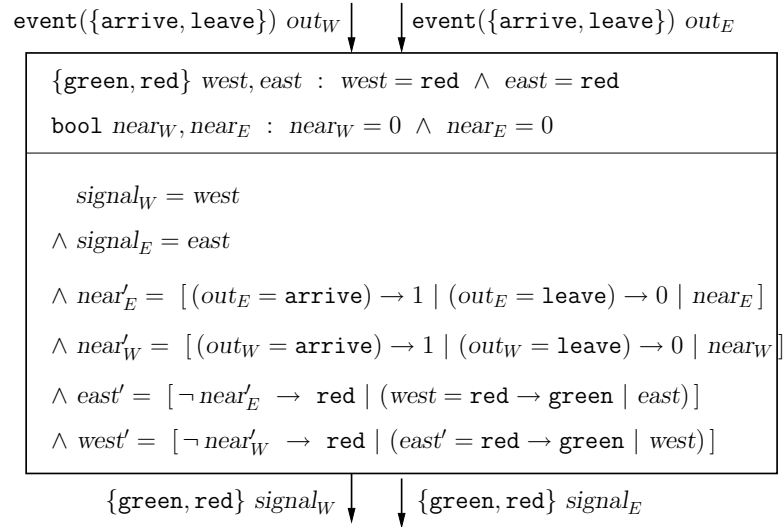


Figure 2.6: A safe controller for the railroad problem

SAFETY MONITOR

A safety monitor for a synchronous reactive component $C = (I, O, S, \text{Init}, \text{React})$ consists of (1) a synchronous reactive component M such that the set of input variables of M is a subset of $I \cup O$ and the set of output variables of M is disjoint from $I \cup O$, and (2) a property φ over the state variables of M . The component C satisfies the monitor (M, φ) , if the property φ is an invariant of the composed system $C \parallel M$.

The requirement that the inputs to the monitor M are the inputs and outputs of the component C means that M can observe the behavior of C . The requirement that the outputs of M are neither the inputs nor the outputs of C ensures that the behavior of C is not influenced by M , and also that it is compatible with M . We design the monitor so that it enters an error state when the observed sequence of inputs/outputs violates the safety requirement, and this is captured by the expression φ that is required to be an invariant.

Not all requirements can be expressed as safety monitors. For the railroad crossing example, consider the controller that always keeps both the traffic lights red. Such a controller satisfies the invariant **TrainSafety**. To rule out such solutions that avoid bad situations by not attempting to do anything good, we need to impose additional requirements such as “if both trains are waiting, then the controller must allow some train to eventually enter the bridge.” Such a requirement is called a *response* requirement. In this requirement, we have not asserted any bound on the number of rounds the trains have to wait. As a result, a finite execution in which both trains are waiting in the last, say

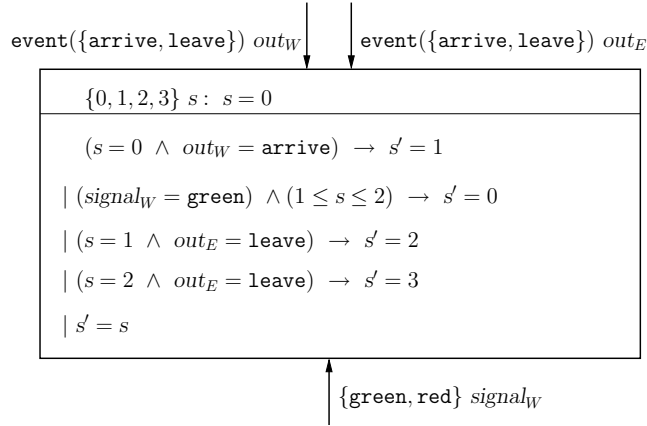


Figure 2.7: Fairness monitor `WestFairMonitor` for the west train

10, rounds of the execution, cannot be considered a violation of the response requirement. Indeed, there may be a correct implementation of the controller that is slow in its processing of requests, and needs 11 rounds to turn the signal to green. In general, no finite execution can demonstrate that the response requirement is truly violated. This is not a safety requirement, and thus, cannot be expressed using monitors and invariants. If we change the requirement to a *bounded response* requirement such as “if both trains are waiting and both signals are red, then the controller must turn one of the signals to green in the next round,” it can be captured by a safety monitor. We will study specification and analysis of response and other forms of liveness requirements in Chapter ??.

2.1.4 Exercises

1. The composed system `RailRoadSystem1` has 4 state variables, *east* and *west*, each of which can take 2 values, and `TrainW.s` and `TrainE.s`, each of which can take 3 values. Thus, `RailRoadSystem1` has 36 states. How many of these 36 states are reachable?
2. Does the second attempt to design the railroad controller satisfy the fairness requirement captured by the monitor `WestFairMonitor`. That is, is the property `WestFairMonitor.s ≠ 3` an invariant of the composite system `RailroadSystem2 || WestFairMonitor` ? If not, show an execution that violates the requirement.
3. The design of the controller `Controller2` always gives priority to the east train. Suppose we want to resolve the contention in a round-robin manner: a train should be admitted on the bridge if either the other train is away or if both need to get on the bridge, then the most recently admitted train is the other train. First, specify this requirement using a safety monitor.

Then, design a railroad controller that satisfies this requirement (along with the basic safety requirement `TrainSafety`).

2.2 Verifying Invariants

In the invariant verification problem, we are given a transition system T and a property φ , and we want to check whether φ is an invariant of T . If not, we should output a counter-example that demonstrates the reachability of a state violating the property.

2.2.1 Complexity of Invariant Verification

Before we proceed to consider some techniques for checking invariants of transition systems, let us consider the computational complexity of the invariant verification problem. To make a precise claim about the computational complexity, we must fix the language for writing initialization and transition expressions.

First, let us start with systems all of whose variables have Boolean types. We have already been using logical connectives to write Boolean expressions. When all variables are Booleans, the expressions formed using logical connectives are called *propositional* expressions.

Propositional satisfiability

Let us review the rules for constructing propositional expressions. Given a set V of Boolean variables, the set of *propositional expressions* is defined by the following rules:

1. The constants 0 and 1, and the variables in V , are propositional expressions.
2. If x is a variable in V , and e is a propositional expression, then so is $x = e$.
3. If e is a propositional expression then so is $\neg e$ (negation).
4. If e_1 and e_2 are propositional expressions, then so are $e_1 \wedge e_2$ (conjunction), $e_1 \vee e_2$ (disjunction), and $e_1 \rightarrow e_2$ (implication).
5. If e , e_1 , and e_2 are propositional expressions then so is the if-then-else expression $e \rightarrow e_1 \mid e_2$.

Expressions also use parentheses to allow correct parsing. Given a propositional expression e over V , and a valuation s over V , to check whether s satisfies e , we can simply evaluate e according to the rules of logical operators using the values assigned by s to the variables appearing in e . This evaluation can be done in time linear in the length of the expression e . To check satisfiability of a propositional expression e , we must find a valuation that satisfies e . This check is computationally expensive, and in the worst-case is exponential in the

number of variables in V : if V has n variables, there are 2^n possible valuations over V . The precise complexity class of checking satisfiability of propositional expressions is NP-complete. Here NP stands for problems that can be solved by *nondeterministic* algorithms in time polynomial in the size of the input, and *complete* refers to the fact that the problem is a canonical representative of all such problems: if propositional satisfiability can be solved efficiently, then so can be every problem in the class NP. Whether propositional satisfiability can be solved by a polynomial-time algorithm remains a long-standing open problem in computer science. The complement problem of satisfiability is validity: given a propositional expression e , do all valuations satisfy e ? The propositional validity problem is coNP-complete.

Note that for reactive components such as **Delay** and **Relay** with only Boolean variables, the initialization and reaction expressions are indeed propositional. Use of output hiding, and mapping reaction expressions to transition expressions, introduces existential quantification. To allow existential quantification, let us precisely define the set of *existential propositional expressions* consisting of expressions that have a prefix of existential quantifiers followed by a propositional expression: if e is a propositional expression over a set V of Boolean variables, and x_1, x_2, \dots, x_n are variables in V , then $\exists x_1. \exists x_2. \dots \exists x_n. e$ is an existential propositional expression over the variables $V \setminus \{x_1, \dots, x_n\}$. Adding existential quantification does not affect the complexity of checking satisfiability: the expression $\exists x_1. \exists x_2. \dots \exists x_n. e$ is satisfiable precisely when e is satisfiable.

Invariant verification of propositional transition systems

Propositional transition systems have only Boolean variables and their dynamics are specified using propositional formulas.

PROPOSITIONAL SYMBOLIC TRANSITION SYSTEM

A symbolic transition system $(S, Init, Trans)$ is *propositional* if each state variable in S is Boolean, $Init$ is a propositional expression over S , and $Trans$ is an existential propositional expression over $S \cup S'$.

A propositional transition system with n state variables has 2^n states. Unlike satisfiability of propositional formulas, which requires finding just one satisfying assignment, establishing that some property is an invariant of a propositional transition system, we need to establish that all reachable states satisfy the invariant. This is intuitively a more complex problem than satisfiability as well as validity. Its precise complexity class is known as PSPACE that denotes computations that require space polynomial in the size of the input. Invariant verification, or dually, reachability, for a propositional transition system, is a canonical problem for this class. Establishing PSPACE upper bound, and showing that the problem is computationally hard with PSPACE lower bound, would both require detours into complexity theory. We will just note the result:

Theorem 2.1 [Complexity of Propositional Invariant Verification] *Given a propositional transition system $T = (S, \text{Init}, \text{Trans})$, and a propositional expression φ over S , the problem of checking whether φ is an invariant of T is PSPACE-complete.*

It is worth noting that the PSPACE complexity of the propositional invariant verification problem is due to the fact that the state of a propositional transition system is encoded using Boolean variables, and is not an artifact of the choice of propositional expressions to encode initialization and transition expressions. The complexity would stay the same even if we require the system to be deterministic, and even if the transitions are specified in an executable manner (for example, as a sequence of assignments, one for each of the Boolean variables).

For analysis purpose, a variable of an enumerated type can be replaced by Boolean variables in an obvious manner. For example, consider the state variable s of the `Train` component that takes 3 possible values `away`, `wait`, and `bridge` (see Figure 2.2). We can encode the variable s using two Boolean variables s_0 and s_1 , using values 00, 01, and 10 to encode the three possibilities for s . Expressions of the form $s = \text{away}$ are replaced by $\neg s_0 \wedge \neg s_1$; expressions of the form $s = \text{wait}$ are replaced by $\neg s_0 \wedge s_1$; and expressions of the form $s = \text{bridge}$ are replaced by $s_0 \wedge \neg s_1$. The complexity of the verification problem for transition systems with Boolean and enumerated variables is, thus, no different than for propositional transition systems.

Undecidability

Let us consider the invariant verification problem for transition systems that are not finite-state. In this general case, the verification problem is *undecidable*. This means that we cannot hope to have a completely algorithmic solution to solve the invariant verification problem, and we will have to settle for approximate or heuristic strategies that may not work in all cases. The undecidability of the problem follows from the results in computability theory. The halting problem for Turing machines, and for sequential programs, is undecidable. Computation of a program can easily be encoded as a transition system and halting can be captured as a violation of an invariant. In fact, if we limit ourselves to transition systems that have only counter variables, invariant verification problem is undecidable, where a counter variable is a variable of type `nat` that in one transition can only be incremented, decremented, or tested for being equal to zero.

2.2.2 Simulation-based Testing

The most commonly used industrial technique for analyzing systems is testing using simulation. Given a user-specified value k for the number of steps of the simulation, the algorithm generates an execution of the transition system containing k transitions, and checks if the invariant holds for every state visited

during this execution. In general, the transition system has many executions of a given length. For instance, in transition systems corresponding to synchronous reactive components, each state can have multiple successors due to inputs and/or due to nondeterminism within the component. In such a case, the simulator must resolve the choice in some way, by using randomization, for instance.

The simulation-based algorithm can process transition systems represented in many different possible ways ranging from source code to internal representation. What is needed is a way of generating an initial state from the representation, and a way of generating a successor state of a given state. More specifically, the simulation-based algorithm relies on the implementation of the following operations.

- States of the transition system are of the type `state`. The constant `null` specifies a dummy state.
- Given a transition system T , the function $ChooseInitState(T)$ returns *some* initial state and the dummy state `null` if T has no initial states.
- Given a transition system T and a state s of T , the function $ChooseSuccState(s, T)$ returns *some* successor state of s , that is, some state t such that $s \rightarrow t$ is a transition of T , and the dummy state `null` if s has no outgoing transitions.
- The algorithm relies on one operation on the representation of properties: the Boolean function $Satisfies$ takes a property and a state, and returns 1 if the input state satisfies the property.

The simulation algorithm is presented in Figure 2.8. The variable `exec` is an array of states, which the algorithm fills one by one, for the specified number of steps. At each step, the function $Satisfies$ is used to check if the current state violates the desired invariant. Note that if no violation is found, the algorithm cannot make any conclusions about whether or not the invariant holds. In such a case, we can run it repeatedly to gain more confidence in the correctness of the system. However, such analysis cannot prove that the system indeed satisfies the invariant.

Representing transition systems

The actual running time of the simulation-based algorithm depends on how long it takes to execute functions such as $ChooseSuccState$ that depend on the representation of the transition system. If the transition system is presented in a purely declarative manner, then computing the set of initial states and successor states is computationally demanding. For example, if the initial states are given as an (unconstrained) propositional expression $Init$, then finding some initial state amounts to finding a satisfying valuation of $Init$. Since propositional satisfiability is NP-complete, this task is computationally hard. As we discussed in Chapter 1, the modeling languages used in practice are executable, and thus,

```

Input: A transition system  $T$ , a property  $\varphi$ , and an integer  $k > 0$ ;
Output: If a violation of the invariant is encountered, return a
        counter-example;

array [state] exec;
nat  $j = 0$ ;
state  $s = ChooseInitState(T)$ ;

if  $s = \text{null}$  then return;
exec[ $j$ ] =  $s$ ;
if  $Satisfies(s, \varphi) = 0$  then return exec;
for  $j = 1$  to  $k$  do {
     $s = ChooseSuccState(s, T)$ ;
    if  $s = \text{null}$  then return;
    exec[ $j$ ] =  $s$ ;
    if  $Satisfies(s, \varphi) = 0$  then return exec;
}.

```

Figure 2.8: Simulation-based invariant verification

initial states and successors of a given state can be computed efficiently by executing the model description. One possible representation for the transition system, then, is the original description of the system in the source modeling language itself. With this choice, the implementation of a function such as *ChooseSuccState* is an interpreter that analyzes the source model each time it is called. The more pragmatic choice is to compile the source model to an efficiently executable language such as C.

State Compaction

The memory used by any analysis algorithm that stores states is obviously affected by how an individual state is represented and stored. For the simulation algorithm, the states are stored in the array *exec*. The most natural data structure for representing a state is a record type with a field for each of the state variables. However, this is too wasteful: if a system has, say, 50 state variables, allocating a word to each field on a 32-bit machine would mean 400 bytes per state, and storing thousands of states would be impractical. As a result, a state is represented using low-level bit encoding. For each state variable, the analysis tool first computes an upper bound on the number of bits. A Boolean variable needs just one bit, a variable ranging over an enumerated type with 3 values needs 2 bits, and for a variable storing speed of a car, in miles per hour upto one decimal point, 10 bits should suffice. The state then is encoded as a sequence of bits.

2.2.3 Proving Invariants

Inductive invariants

Consider a transition system $T = (S, Init, Trans)$ and a property φ . If we can establish that φ holds initially, and is preserved during every transition, then, by induction, it should hold at every state encountered along every execution, and thus, should be an invariant of T . Showing that the property holds initially amounts to establishing that

every state satisfying $Init$ also satisfies φ .

This condition can be succinctly stated as: the implication expression $Init \rightarrow \varphi$ is valid. Showing that the property is preserved by every transition amounts to establishing that

if a state s satisfies φ , and $s \rightarrow t$ is a transition of T , that is, if $[s, t]$ satisfies the transition expression $Trans$, then t should satisfy φ .

This condition can be formalized as: the implication $\varphi \wedge Trans \rightarrow \varphi'$ is valid. Here, φ is an expression over the state variables S , $Trans$ is an expression over $S \cup S'$ capturing the transition relation between unprimed and primed state variables, and φ' is obtained from φ by substituting every state variable x with its primed version x' .

Properties that hold initially and are preserved by the transition relation are called inductive invariants:

INDUCTIVE INVARIANT

A property φ of a transition system $T = (S, Init, Trans)$ is an *inductive invariant* if (1) $Init \rightarrow \varphi$ is valid, and (2) $\varphi \wedge Trans \rightarrow \varphi'$ is valid.

Let us consider the program for greatest common divisor again. For the transition system $GCD_{m,n}$ consider the property $\text{gcd}(m, n) = \text{gcd}(x, y)$. To show that this is an inductive invariant, let us first consider the implication corresponding to initialization. We want to show that if the state satisfies the initialization expression $(x = m \wedge y = n)$ then the property $\text{gcd}(m, n) = \text{gcd}(x, y)$ must hold. This is indeed the case. Now let us focus on the preservation by the transition relation. Assuming that $\text{gcd}(m, n) = \text{gcd}(x, y)$ holds, and the variables x' and y' satisfy the transition expression

$$x' = [(x > y) \rightarrow x - y \mid x] \wedge y' = [(y > x) \rightarrow y - x \mid y],$$

we want to show that $\text{gcd}(m, n) = \text{gcd}(x', y')$ holds. This can be established by a case analysis. For example, consider the case when $x > y$. In this case, we know that x' is $x - y$ and y' is y . So we need to show that $\text{gcd}(x - y, y) = \text{gcd}(x, y)$, which indeed is the case.

As another example, let us revisit the leader election protocol from Section 1.4.2. For a set P of nodes and a set E of directed links that induce a strongly-connected graph over the nodes, consider the system **SyncLE** defined as the composition

$$\parallel_{n \in P} \text{SyncLENode}_n \parallel \text{SyncNetwork}_{P,E}.$$

Let n be a node. Consider the assertion that the value of its state variable `id` equals the identifier of some node in P :

$$\varphi_1 : \text{SyncLENode}_n.\text{id} \in P.$$

Let us check if φ_1 is an inductive invariant of the system. Initially, the value of `SyncLENoden.id` equals n , and thus, φ_1 holds. Now, consider a state s that satisfies φ_1 . We want to show that if the system executes one transition, φ_1 continues to hold. If the only assumption about s is that it satisfies φ_1 , that is, the value of `SyncLENoden.id` in state s is identifier of some node in P , can we conclude that, after one step of the protocol, the value of `SyncLENoden.id` will still be in P ? Not really. In one step, the node n receives a set of identifiers sent by its neighboring nodes, and updates its `id` variable to the maximum of its current value and the incoming values. To conclude that the new value will be equal to the identifier of one of the nodes in the network, we need the assumption that the incoming values also belong to the set P . But the assertion φ_1 does not capture this. That is, the property φ_1 , though is an invariant of the system, is not *strong enough* to be inductive. The *inductive strengthening* of the property is

$$\varphi_2 : \bigwedge_{m \in P} \text{SyncLENode}_m.\text{id} \in P.$$

The property φ_2 asserts that for *every* node n , the value of the `id` variable of node n is in P . The property φ_2 is stronger than φ_1 : if φ_2 holds, clearly so does φ_1 . Furthermore, check that φ_2 is inductive. It holds initially. Now consider a state s that satisfies φ_2 . Now consider any node n . The updated value of `SyncLENoden.id` after one transition will be equal to the value of `SyncLENoden.id` in state s , or the value of `SyncLENodem.id` in state s , for some node m with a link to node n . Since in state s , by assumption, all the `id` variables take values from the set P , we can conclude that the new value of `SyncLENoden.id` will be in P . This implies that φ_2 is indeed inductive.

Proof rule for establishing invariants

The method of inductive strengthening is a general-purpose and powerful technique for establishing invariants.

PROOF RULE FOR INVARIANTS

To establish that a property φ is an invariant of the transition system T , find a property ψ such that (1) ψ is an inductive invariant of T , and (2) the implication $\psi \rightarrow \varphi$ is valid.

The above proof rule is *sound*, that is, it is a correct method for establishing invariants. If the property ψ is an inductive invariant, then all reachable states of T must satisfy ψ . If $\psi \rightarrow \varphi$ is valid, then every state satisfying ψ must also satisfy φ . It follows that φ is an invariant of T if both the assumptions are satisfied.

The proof rule is also *relatively complete*. What this means is that if we have a way of establishing validity of Boolean expressions for an assertion language that is expressive enough, then if φ is indeed an invariant, then there does exist an inductive property ψ that is stronger than φ , and thus, the proof rule can be used to verify the invariant. In particular, suppose we can find a Boolean expression ψ that holds only in those states of T that are reachable. Clearly, if φ is an invariant, then it holds in all reachable states, and the implication $\psi \rightarrow \varphi$ must hold. The property ψ capturing precisely the reachable states is inductive: initial states are reachable, and executing one transition from a reachable state leads to a reachable state. If the assertion language for expressions can describe the set of reachable states, then we know that the proof rule gives a complete method.

In practice, to prove invariants of a system in a rigorous manner, the user must identify the inductive strengthening. Establishing that ψ is indeed inductive and that $\psi \rightarrow \varphi$ holds, one can either use a “paper-and-pencil” argument, or use an automated theorem prover.

Proof of the synchronous leader election protocol

To illustrate a more realistic proof, let us consider the correctness property for the system SyncLE for solving the leader election problem in synchronous networks. Consider the following property φ_{leader} which asserts that, for every node n , after N rounds, the value of the `id` variable equals the highest identifier in P :

$$\varphi_{leader} = \bigwedge_{n \in P} (\text{SyncLENode}_n.r = N \rightarrow \text{SyncLENode}_n.id = \max P).$$

This property is not inductive. To strengthen it, we must assert that, at the beginning of each round j , the value of the `id` variable for node n is the maximum of identifiers of nodes at distance less than j . Let $\text{dist}_E(m, n)$ denote the length of the shortest path from node m to node n according to the links in E . The distance of node from itself is 0. Consider the following property ψ :

$$\psi = \bigwedge_{n \in P} \text{SyncLENode}_n.id = \max \{m \mid \text{dist}_E(m, n) < \text{SyncLENode}_n.r\}$$

The property ψ does imply the desired invariant φ_{leader} , this is because the distance between any pair of nodes is at most $N-1$, and hence, when $\text{SyncLENode}_n.r$ equals N , the set $\{m \mid \text{dist}_E(m, n) < \text{SyncLENode}_n.r\}$ must equal the set P of all identifiers.

Let us check whether the property ψ is an inductive invariant of **SyncLE**. It holds initially, the only node at distance 0 from a node n is n itself. Unfortunately, it is not strong enough to be preserved in every transition. Consider a state s satisfying the above property, and a node n . Suppose the value of the round variable r of n is $j < N$. Then, we know that the value of id for n will be $\max\{m \mid \text{dist}(m, n) < j\}$. In one round, the node n receives the current values of id variables of all its neighbors, and updates its id variable to the maximum of its current value and all the values received. Since the node n increments the variable r , we need to show that the updated value of id is $\max\{m \mid \text{dist}(m, n) < j + 1\}$. Now, any node, say n_1 , at distance less than $j + 1$ from n must be at distance less than j from one of the neighbors, say n_2 , of n . Do we know that the value of id of node n_2 must be at least n_1 in state s ? We know, from the property ψ , that the value of id of node n_2 is the maximum of identifiers of all nodes at distance less than the value of the round variable r of n_2 . We would be done, if we could conclude that the value of the round variable r of n_2 in state s is j . But this is not really stated in the property ψ , and the proof fails. The needed strengthening also asserts that the values of the round variables of all the nodes are equal. The following property ψ_{leader} is indeed an inductive invariant of the system **SyncLE**, and implies φ_{leader} :

$$\psi_{leader} = \psi \wedge \bigwedge_{m, n \in P} \text{SyncLENode}_m.r = \text{SyncLENode}_n.r$$

2.3 Enumerative Search

Invariant verification problem is similar to a reachability problem in a graph. Given a transition system T and a property φ , we want to know if there is an execution starting from an initial state leading to a state that violates φ . This problem can be viewed as finding a path in a graph whose vertices correspond to states of T and whose edges correspond to transitions of T . In classical graph search algorithms, the input graph is represented by listing all its vertices and edges. In invariant verification, the graph is given implicitly, for instance, by the initialization and transition expressions of the transition system, and may not be finite. Indeed, we do not want to build the graph explicitly, and explore the graph only as much as needed.

Reachable subgraph

For a transition system T , the graph consisting of the reachable states of T and transitions out of these states constitutes the reachable subgraph of the system.

Let us revisit the controller **Controller2** for the railroad crossing example again. For the component **RailRoadSystem2**, there are 6 state variables: **Train_W.s** and **Train_E.s**, each of which range over **{away, wait, bridge}**; **west** and **east**, each of which range over **{green, red}**; and Boolean variables **near_W** and **near_E**. As a result, the system has 144 possible states. It turns out that very few of these

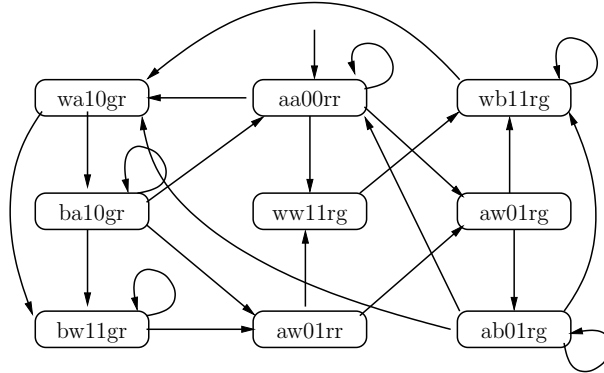


Figure 2.9: Reachable states for RailRoadSystem2

states are reachable. Figure 2.9 shows the *reachable* portion of the graph. Each state is denoted by listing the values of $\text{Train}_W.s$, $\text{Train}_E.s$, near_W , near_E , *west*, and *east*, in that order. We use *a*, *w*, *b*, *g*, and *r*, as abbreviations for **away**, **wait**, **bridge**, **green**, **red**, respectively. The initial state then is *aa00rr* denoting that both trains are away, both *near* variables are 0, and both signals are **red**. The four transitions out of this state correspond to the possibilities of one, or both, or neither of the two trains arriving. One of the transitions is a self-loop to *aa00rr*, and the other 3 lead to the states *wa10gr*, *aw01rg*, and *ww11rg*. We then systematically consider all the possible transitions from these 3 new states, and continue till no new states are discovered. Only 9 out of 144 states are found to be reachable.

As another example, consider the synchronous leader election. Suppose we choose values for the set P of nodes and the set E of network links. The component **SyncLE** still has infinitely many states, since the type of the variables such as r and id of each of the node components is **nat**. However, during an execution of the system, the value of the round variable r ranges over $\{1, 2, \dots, N\}$, where N is the number of nodes in P , and the value of the identifier variable id ranges over P . Thus, for a given network, the number of reachable states of the system is finite.

These examples indicate that the analysis algorithm for the invariant verification problem should explore only the reachable states of the system starting with the initial states. Such a search procedure is called *on-the-fly* since it examines states and transitions in an incremental manner. It is *enumerative* as it processes states individually.

On-the-fly depth-first-search

As in case of the simulation-based exploration, our presentation of the on-the-fly algorithm does not rely on a specific representation of transition systems. In the

case of enumerative search, what we need is that there is a way to systematically enumerate all the initial states of a transition system, and for given a state s , there is a way to systematically enumerate all the successors of s , that is, states t such that $s \rightarrow t$ is a transition. As in case of simulation-based exploration, states of the transition system are of the type `state`. We will use the following functions that access the representation of the transition systems:

- Given a transition system T , the function $FirstInitState(T)$ returns the first initial state of T , according to the chosen enumeration of the initial states, and the dummy state `null` if T has no initial states.
- Given a transition system T and an initial state s , the function $NextInitState(s, T)$ returns the initial state following s in the chosen enumeration of the initial states, and `null` if no such state exists.
- Given a transition system T and a state s of T , the function $FirstSuccState(s, T)$ returns the first successor state of s , according to the chosen enumeration of the set of states t such that $s \rightarrow t$ is a transition of T , and the dummy state `null` if s has no outgoing transitions.
- Given a transition system T , a state s , and a state t that is a successor of s , the function $NextSuccState(s, t, T)$ returns the state following t in the chosen enumeration of the set of successor states of s , and the dummy state `null` if t is the last state in this list.

For instance, if T has a state variable x of type `nat`, and the initialization specifies that $x \geq 10$, then T has infinitely many initial states., but they can be enumerated as 10, 11, 12, In such a case, $FirstInitState$ can return 10, and for $n \geq 10$, $NextInitState$ on input n , can return $n+1$. The ability to systematically list all the initial states and the successors of a state is necessary for enumerative search. If T has a state variable x of type `real`, and the initialization specifies that $0 \leq x \leq 1$, there are uncountably many initial states, one every real number in the interval $[0, 1]$. For such a system it would not be possible to apply a systematic search exploring states one by one. The simulation-based exploration is still possible, as one can implement `ChooseInitState` to return a randomly chosen real number upto a given precision of the floating-point representation of numbers..

A transition system T for which the set of initial states, and the set of successor states of each state, can be effectively enumerated is called *countably-branching*: whenever there is a choice in extending an execution, the number of choices is countable.

The classical depth-first-search algorithm for on-the-fly exploration of (countably-branching) transition systems is depicted in Figure 2.10. It relies on the following data structures.

- The variable *Reach* stores the set of reachable states, and is of type `set(state)`. The operations used on this set data structure are: an initialization constant `EmptySet` that corresponds to the empty set; a Boolean

```

Input: A countably-branching transition system  $T$  and property  $\varphi$ ;
Output: If  $\varphi$  is an invariant of  $T$  return 1, else return a counter-
example;

set(state) Reach = EmptySet;
stack(state) Pending = EmptyStack;
state  $s = \text{FirstInitState}(T)$ ;

while  $s \neq \text{null}$  do {
  if  $\text{Contains}(\textit{Reach}, s) = 0$  then
    if  $\text{DFS}(s) = 0$  then return  $\text{Reverse}(\textit{Pending})$ ;
     $s = \text{NextInitState}(s, T)$ ;
  };
return(1).

bool function  $\text{DFS}(\text{state } s)$ 
  Insert( $s, \textit{Reach}$ );
  Push( $s, \textit{Pending}$ );
  if  $\text{Satisfies}(s, \varphi) = 0$  then return 0;
  state  $t = \text{FirstSuccState}(s, T)$ ;
  while  $t \neq \text{null}$  do {
    if  $\text{Contains}(\textit{Reach}, t) = 0$  then
      if  $\text{DFS}(t) = 0$  then return 0;
       $t = \text{NextSuccState}(s, t, T)$ ;
    };
  Pop( $\textit{Pending}$ );
  return 1.

```

Figure 2.10: On-the-fly depth-first search algorithm for invariant verification

membership function **Contains** that takes a set and a state, and returns 1 if the input state belongs to the input set; and an insertion procedure **Insert** that takes a state and a set, and updates the input set by adding the input state to it.

- The variable *Pending* stores the sequence of states from which exploration is in progress, and is of type **stack(state)**. The operations used on this stack data structure are: an initialization constant **EmptyStack** that corresponds to the empty stack; a procedure **Push** that takes a state and a stack, and updates the input stack by adding the input state at its top; a procedure **Pop** that takes a stack, and updates it by removing the top element, if any; and the function **Reverse** takes a stack and returns the sequence of states it contains from bottom to top.

The algorithm maintains the set *Reach* of the states it has encountered so far. All the states in this set are guaranteed to be reachable states of the transition system T . The initial states of the transition system are supplied to the

algorithm one by one by the functions *FirstInitState* and *NextInitState*. The algorithm initiates search from every initial state that is not already visited, by calling the recursive function *DFS*. When *DFS* is called with an input state s , it adds it to the set *Reach*, and pushes it onto the top of the stack *Pending*. At any time the stack contains a sequence of states such that the state at the bottom is an initial state, and every state has a transition from the state immediately below it. If the input state s violates the property φ , then the algorithm has discovered a violation, and all pending invocations of *DFS* terminate. The stack contains an execution of the transition system that demonstrates reachability of the state violating the invariant, and thus, the reversed stack can be output as a counter-example. If the input state s satisfies the desired invariant, then the algorithm examines all the outgoing transitions from s . The successor states of s are supplied to the algorithm one by one by the functions *FirstSuccState* and *NextSuccState*. The algorithm calls *DFS* recursively from those successor states that are not already visited. If all the *DFS* calls terminate without encountering a violating state, then the algorithm has visited all the reachable states, and returns 1, indicating that φ is indeed an invariant.

For a reachable state s , the algorithm calls *DFS* with input s at most once, and thus it processes every transition out of a reachable state at most once. If the number of reachable states of a transition system is finite, then the algorithm is guaranteed to terminate, and its running time will be linearly proportional in the number of reachable states and transitions. Even when the number of reachable states is not finite, the algorithm may encounter a violation, and terminate with a counter-example. However, if the system satisfies the invariant, and the number of reachable states is not finite, then the algorithm will not terminate. These properties are summarized in the following theorem:

Theorem 2.2 [On-the-fly Depth-first-search for Invariant Verification] *Given a countably-branching transition system T and a property φ , the depth-first-search algorithm of Figure 2.10 has the following properties: (1) if the algorithm returns 1, then the property φ is an invariant of T , (2) if the algorithm returns a sequence of states, then its output is an execution of T ending in a state that violates φ , and (3) if the number of reachable states of T is finite, then the algorithm terminates, and the number of calls to *DFS* is bounded by the number of reachable states.*

We conclude the discussion of the enumerative search by discussing some commonly used implementation techniques employed by enumerative model checkers for improving the efficiency of the depth-first-search algorithm. As in case of simulation-based exploration, the representation of the transition system should allow fast computation of initial states and successor states of a given state, and memory requirements of the algorithm can be reduced by compact encoding of states.

Hashing

The most commonly used data structure for storing the set *Reach* of states already visited is a *hash-table*. A hash-table consists of a hashing function that maps each state to an integer between 0 and N , for a suitably chosen positive integer N , and an array of length N , where each entry is a list of states. To insert a state s in the hash-table, first the hashing function is used to map s to an index j , and then the state s is added to the list at the j -th element of the array. To check if a state s is already in the hash-table, the hashing function is used to map s to an index j , and the list at the j -th element of the array is scanned to check if it contains s . If the hashing function is chosen well, then the number of states mapped to the same index is small, and as a result, both **Insert** and **Contains** take near-constant time.

While hashing is an effective technique to store the set of explored states, often the number of reachable states is too large to be stored in memory. In such cases, an approximate strategy, known as *bit-state hashing*, can be used. This approach uses a hash table of size N whose j -th entry is a single bit. The insertion of a state s , which is mapped to an integer j between 0 and N by the hash function, is implemented by setting the j -th bit of the hash table to 1. All hash collisions are ignored. Suppose that two states s and t are mapped to the same index j , and s is inserted in the hash table first. When the state t is encountered, as the j -th bit of the hash table is already set, the membership test **Contains** returns a positive answer. Consequently, the depth-first-search algorithm does not explore the successors of t . Hence, only a fraction of the set of reachable states is explored. The algorithm may return *false positives*—it may return 1 even when the invariant is not satisfied, but no *false negatives*—every violating execution that it discovers indeed signals a counter-example illustrating a violation of the invariant. What fraction of the reachable region is visited by bit-state hashing depends on the choice of the table size and the hash function. The performance of bit-state hashing can be improved dramatically by using two bit-state hash tables that employ independent hash functions. Each explored state is stored in preferably both, but at least in one hash table, so that a collision occurs only if both table entries are already occupied.

2.3.1 Exercises

1. **Stateless Search:** To save memory, sometimes the on-the-fly depth-first-search algorithm is modified so that it does not store any states. That is, we modify the algorithm of Figure 2.10 by removing the set *Reach*, and when a state is encountered there is no test to check if it was visited before. This form of search is called *stateless search*. Show that the partial correctness of the algorithm is still preserved: the claims (1) and (2) of Theorem 2.2 continue to hold. How is the termination (claim (3) of Theorem 2.2) affected?
2. **Breadth-first search:** A breadth-first search algorithm first examines all

the initial states, then all the successors of the initial states, and so on. To implement such an algorithm we need two functions on the representation of transition systems: *InitStates* returns a list of initial states of a given transition system, and *SuccStates* returns a list of the successors of a given state of a transition system. This is feasible for *finitely-branching* transition systems, namely, transition systems for which the number of initial states is finite, and each state has only finitely many successor states. Given a finitely branching transition system T and a property φ , develop a breadth-first search algorithm to check whether φ is an invariant of T , and state precisely its correctness and termination properties.

2.4 Symbolic Search

As the invariant-verification problem is computationally hard, we cannot hope to find a polynomial-time solution. There are, however, heuristics that perform well on many instances of the invariant-verification problem that occur in practice. One such heuristic is based on a symbolic reachability analysis of the transition system. Instead of considering explicitly represented states one at a time, a symbolic search algorithm processes sets of states represented by constraints. For example, for an integer variable x , the constraint $20 \leq x \leq 99$ identifies the set $\{20, 21, \dots, 99\}$ of 80 states. Such a symbolic representation can be succinct, and with suitable operations that can manipulate the symbolic representation, we can develop a symbolic search algorithm that can solve the invariant verification problem.

2.4.1 Symbolic Breadth-first Search

First, let us identify the operations that we need for symbolic search.

Operations on regions

We call a symbolically represented set of states a *region*. Given a set V of typed variables, a set of states over V is represented as a region of type **reg**. In the symbolic representation of a transition system with state variables S , the initial states are represented by a region *Init* over S , and the transitions are represented by a region *Trans* over $S \cup S'$.

The data type **reg** for regions supports the following operations

- Given regions A and B , **Disj**(A, B) returns the region that contains those states that are either in A or in B .
- Given regions A and B , **Conj**(A, B) returns the region that contains those states that are in both the regions A and B .
- Given regions A and B , **Diff**(A, B) returns the region that contains those states that are in A but not in B .

- Given a region A , $\text{IsEmpty}(A)$ return 1 if the region A contains no states, and 0 otherwise.
- Given regions A and B , $\text{IsSubset}(A, B)$ returns 1 if every state in the region A is also contained in the region B , and 0 otherwise.
- Given a region A over variables V and a variable x in V , $\text{Exists}(A, x)$ returns the region over the variables $V \setminus \{x\}$, that contains a valuation s precisely when there is a value m of the type of x such that the valuation $s[x \mapsto m]$ is in A . The existential-quantification operation extends to sets of variables: given a region A over variables V and variables $x_1, \dots, x_n \in V$, $\text{Exists}(A, \{x_1, \dots, x_n\})$ denotes the repeated application $\text{Exists}(\dots(\text{Exists}(A, x_1), \dots), x_n)$.
- Given a region A over variables V , a variable $x \in V$ and a variable $y \notin V$ of the same type as x , $\text{Rename}(A, x, y)$ returns the region over $(V \cup \{y\}) \setminus \{x\}$ that contains valuations of the form $s[x \mapsto y]$ with s in A . This operation also generalizes to sets of variables. Given a region A over variables V , variables $x_1, \dots, x_n \in V$ and variables $y_1, \dots, y_n \notin V$ such that each y_j is of the same type as x_j , $\text{Rename}(A, \{x_1, \dots, x_n\}, \{y_1, \dots, y_n\})$ represents the repeated application $\text{Rename}(\dots(\text{Rename}(A, x_1, y_1), \dots), x_n, y_n)$.

Image computation

The core of symbolic search is *image computation*: given a region A over state variables, we want to compute the region that contains all the states that can be reached from the states in A using one transition. The desired operation Post can be implemented using intersection, renaming and existential quantification as follows. Given a region A , we first conjoin it with Trans , a region over unprimed and primed state variables containing all the transitions. The intersection $\text{Conj}(A, \text{Trans})$ is a region over $S \cup S'$, and contains all the transitions that originate in states in A . Then, we project the result onto the set S' of primed state variables by existentially quantifying the variables in S . The result is the region containing states that can be reached from states in A using one transition. However, it is a region over the primed variables. Renaming each primed variable x' to x gives us the desired region $\text{Post}(A)$.

SYMBOLIC IMAGE COMPUTATION

Consider a transition system with state variables S and transitions represented by the region Trans over $S \cup S'$. Given a region A over S , the post-image of A , defined by

$$\text{Post}(A, \text{Trans}) = \text{Rename}(\text{Exists}(\text{Conj}(A, \text{Trans}), S), S', S)$$

is a region over S that contains precisely those states t for which there is a transition $s \rightarrow t$ of T for some state s in A .

Input: A transition system T given by a region $Init$ for initial states,
 a region $Trans$ for transitions, and a region φ for the property;

Output: Is the property φ an invariant of the transition system T ?

```

reg  $Reach = Init$ ;
reg  $New = Init$ ;
while  $IsEmpty(New) = 0$  do {
  if  $IsSubset(New, \varphi) = 0$  then return 0;
   $New = Diff(Post(New, Trans), Reach)$ ;
   $Reach = Disj(Reach, New)$ ;
};
return 1.

```

Figure 2.11: Symbolic breadth-first search algorithm for invariant verification

As an example, suppose the system has a single variable x of type `real`, and the transition region is given as $x' = 2x + 1$. Consider the region A given as $0 \leq x \leq 10$. In the first step of the image computation, we conjoin A with $Trans$, and this gives the region $0 \leq x \leq 10 \wedge x' = 2x + 1$. Applying existential quantification of x , and simplifying the result, we get $1 \leq x' \leq 21$. The final step renames x' to x , and we get the region $1 \leq x \leq 21$.

Iterative image computation

Now we are ready to describe the symbolic breadth-first-search algorithm. The algorithm of Figure 2.11 computes successive approximations of the set of reachable states by repeatedly applying the image computation, starting with the initial region. The region $Reach$ stores the set of states found reachable so far, and the region New represents the states newly found reachable. The successive values of New capture the minimum number of transitions needed to reach a state: in the j -th iteration of the loop, New contains precisely those states s for which the shortest execution from an initial state leading to s involves j transitions. Initially, both the regions $Reach$ and New are set to $Init$. If any state in New violates the invariant, that is, if the region New is not a subset of φ , then the algorithm stops reporting invariant violation. If the region New is empty, then the algorithm can terminate reporting successful verification of the invariant. Otherwise, to find the states that are reachable in one more step, the algorithm applies the `Post` operator to the region New , and removes those states that were already known to be reachable using the set-difference operation on regions. The values of the region $Reach$ in the successive iterations of the algorithm are depicted in Figure 2.12.

If the symbolic breadth-first-search algorithm terminates, then its answer is correct. If the transition system violates the invariant, and the shortest counterexample contains j transitions, then after j iterations of the while-loop, the

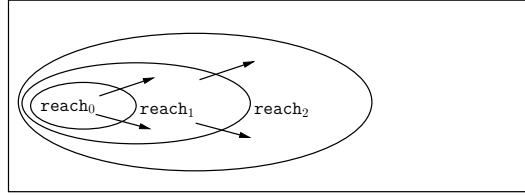


Figure 2.12: Symbolic breadth-first computation for invariant verification

algorithm will find this violation and terminate. If the transition system satisfies the invariant, then the algorithm can terminate only if it discovers all the reachable states after a finite number of iterations. This is obviously the case if the number of reachable states of T is finite.

Theorem 2.3 [Symbolic Breadth-first-search for Invariant Verification] *Given a transition system T and a property φ , the symbolic breadth-first-search algorithm of Figure 2.11 has the following properties: (1) If the algorithm terminates, then the returned value correctly indicates whether or not the property φ is an invariant of T , (2) If there exists an integer j such that either there is a state that violates φ and is reachable by an execution with j transitions, or every reachable state is reachable by an execution with at most j transitions, then the algorithm terminates after j iterations of the while-loop.*

A natural choice for a symbolic representation of regions is Boolean expressions. An expression is usually represented by its parse tree, or by a directed acyclic graph that allows sharing of common subexpressions to avoid duplication of syntactically identical subexpressions. In particular, for propositional transition systems, we can use propositional expressions as a data type for regions. The operations such as `Disj` and `Conj` correspond to the logical connectives such as disjunction and conjunction. To take set-difference of regions A and B , we can take conjunction of A with the negation of B . Renaming corresponds to textual substitution. Finally, existential-quantifier elimination `Exists(A, x)`, where x is a Boolean variable, corresponds to $A_0 \vee A_1$, where the expression A_0 is obtained from A by replacing each occurrence of x with 0, and the expression A_1 is obtained from A by replacing each occurrence of x with 1. All these operations individually can be implemented efficiently over expressions. However, the check `IsEmpty(A)` corresponds to checking satisfiability of the propositional expression A , and is computationally expensive. and the test `IsSubset(A, B)` corresponds to checking validity of the propositional implication $A \rightarrow B$, and is again computationally expensive. More importantly, in the context of the iterative breadth-first-search of our symbolic algorithm, the main drawback of representing the regions $Reach$ and New as expressions, is that there is no simplification at each step. The expressions will get more and more complex due to the operations applied in each iteration of the loop, and their size grows geo-

metrically with the number of iterations. The data structure of ordered binary decision diagrams offers a possible remedy.

2.4.2 Ordered Binary Decision Diagrams

Ordered binary decision diagrams (OBDDs) provide a compact and canonical representation for propositional expressions (or, equivalently, for boolean functions). The OBDD-representation of propositional expressions is best understood by first considering a related structure called an *ordered binary decision graph* (OBDG).

Ordered binary decision graphs

Let V be a set containing k Boolean variables. A propositional expression e over V represents a function from bool^k to bool . For a variable x in V , the following equivalence, called the *Shannon expansion* of e around the variable x , holds:

$$e \equiv (\neg x \wedge e[x \mapsto 0]) \vee (x \wedge e[x \mapsto 1]).$$

Since the Boolean expressions $e[x \mapsto 0]$ and $e[x \mapsto 1]$ are Boolean functions with domain bool^{k-1} , the Shannon expansion can be used to recursively simplify a Boolean function. This suggests representing Boolean functions as decision graphs.

A decision graph is a directed acyclic graph with two types of vertices, *terminal* vertices and *internal* vertices. The terminal vertices have no outgoing edges, and are labeled with one of the Boolean constants, 0 and 1. Each internal vertex is labeled with a variable in V , and has two outgoing edges, a *left* edge and a *right* edge. Every path from an internal vertex to a terminal vertex contains, for each variable x , at most one vertex labeled with x . Each vertex u represents a Boolean function $e(u)$. Given a valuation q for all the variables in V , the value of the Boolean function $e(u)$ is obtained by traversing a path starting from u as follows. Consider an internal vertex v labeled with x . If $q(x)$ is 0, we choose the left-successor of v ; if $q(x)$ is 1, we choose the right-successor of v . If the path terminates in a terminal vertex labeled with 0, the value $q(e(u))$ is 0; if the path terminates in a terminal vertex labeled with 1, the value $q(e(u))$ is 1.

Ordered decision graphs are decision graphs in which we choose a linear order \prec over V , and require that the labels of internal vertices appear in an order that is consistent with \prec . The requirement that the labels of the children are greater than the label of a vertex ensures that every OBDG is acyclic. Note that there is no requirement that every variable should appear as a vertex label along a path from the root to a terminal vertex, but simply that the sequence of vertex labels along a path from the root to a terminal vertex is monotonically increasing according to \prec . The semantics of OBDGs is defined by associating Boolean expressions with the vertices.

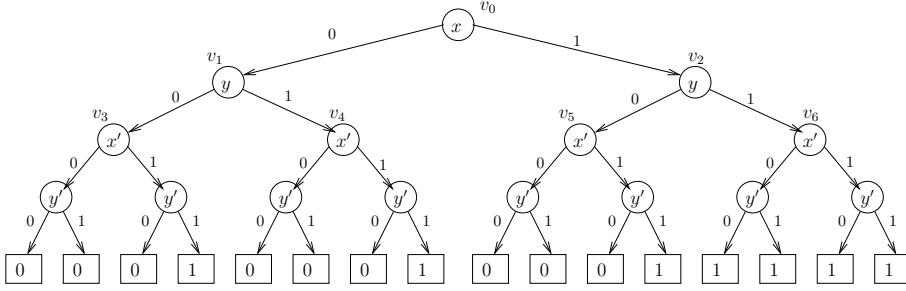


Figure 2.13: Ordered Binary Tree for $(x \wedge y) \vee (x' \wedge y')$

ORDERED BINARY DECISION GRAPH

Let V be a finite set of Boolean variables, and \prec be a total order over V . An *ordered binary decision graph* B over (V, \prec) consists of (1) [Vertices] a finite set U of vertices that is partitioned into two sets; *internal* vertices U^I and *terminal* vertices U^T , (2) [Root] a root vertex u^0 in U , (3) [Labeling] a labeling function $label$ that labels each internal vertex with a variable in V , and each terminal vertex with a constant in $\{0, 1\}$; (4) [Left edges] a left-child function $left$ that maps each internal vertex u to a vertex $left(u)$ such that if $left(u)$ is an internal vertex then $label(u) \prec label(left(u))$, and (5) [Right edges] a right-child function $right$ that maps each internal vertex u to a vertex $right(u)$ such that if $right(u)$ is an internal vertex then $label(u) \prec label(right(u))$. For such an OBDG, each vertex u has an associated Boolean expression over V : $e(u)$ equals $label(u)$ if u is a terminal vertex, and equals

$$[\neg label(u) \wedge e(left(u))] \vee [label(u) \wedge e(right(u))]$$

otherwise. Define $e(B) = e(u^0)$ for the root u^0 .

A Boolean constant is represented by an OBDG that contains a single terminal vertex labeled with that constant. Figure 2.13 shows one possible OBDG for the expression $(x \wedge y) \vee (x' \wedge y')$ with the ordering $x \prec y \prec x' \prec y'$. The left-edges are labeled with 0, and the right-edges are labeled with 1. The OBDG of Figure 2.13 is, in fact, a tree. Figure 2.14 shows a more compact OBDG for the same expression with the same ordering of variables.

Two OBDGs B and C are *isomorphic* if the corresponding labeled graphs are isomorphic. Two OBDGs B and C are *equivalent* if the Boolean expressions $e(B)$ and $e(C)$ are equivalent. If B is an OBDG over (V, \prec) , and u is a vertex of B , then the subgraph rooted at u is also an OBDG over (V, \prec) . Two vertices u and v of the OBDG B are *isomorphic*, if the subgraphs rooted at u and v are isomorphic. Similarly, two vertices u and v are *equivalent*, if the subgraphs rooted at u and v are equivalent. The ordered binary decision graphs of Figures 2.13 and 2.14 are not isomorphic, but are equivalent. In Figure 2.13, the subgraph

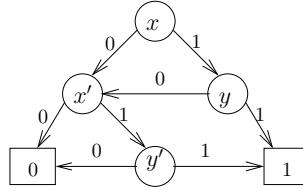


Figure 2.14: Ordered Binary Decision Diagram for $(x \wedge y) \vee (x' \wedge y')$

rooted at vertex v_3 is an OBDG that represents the Boolean expression $x' \wedge y'$. The subgraphs rooted at vertices v_3 , v_4 , and v_5 , are isomorphic. On the other hand, the vertices v_5 and v_6 are not isomorphic to each other.

Ordered binary decision diagrams

An ordered binary decision diagram (OBDD) is obtained from an OBDG by applying the following two steps:

1. Identify isomorphic subgraphs.
2. Eliminate internal vertices with identical left and right children.

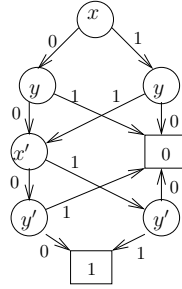
Each step reduces the number of vertices while preserving equivalence. For instance, consider the OBDG of Figure 2.13. Since vertices v_3 and v_4 are isomorphic, we can delete one of them, say v_4 , and redirect the right-edge of the vertex v_1 to v_3 . Now, since both edges of the vertex v_1 point to v_3 , we can delete the vertex v_1 redirecting the left-edge of the root v_0 to v_3 . Continuing in this manner, we obtain the OBDD of Figure 2.14. It turns out that the above transformations are sufficient to obtain a *canonical* form.

ORDERED BINARY DECISION DIAGRAM

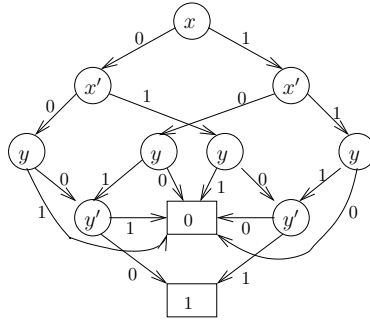
An *ordered binary decision diagram* over a totally ordered set (V, \prec) of Boolean variables is an ordered binary decision graph B over (V, \prec) with vertices U and root u^0 such that (1) [No isomorphic subgraphs] if u and v are two distinct vertices in U , then u is not isomorphic to v , and (2) [No redundancy] for every internal vertex u , the two children $left(u)$ and $right(u)$ are distinct vertices.

The next theorem asserts the basic facts about representing Boolean expressions using OBDDs. Every Boolean function has a unique, upto isomorphism, representation as an OBDD. Furthermore, the OBDD of a Boolean expression has the least number of vertices among all OBDGs for the same expression using the same ordering.

Theorem 2.4 [Existence, Uniqueness, and Minimality of OBDDs] *Let V be a set of variables and \prec be a total order over V .*



Ordering: $x < y < x' < y'$



Ordering: $x < x' < y < y'$

Figure 2.15: Two OBDDs for $(x = y) \wedge (x' = y')$.

1. If e is a Boolean expression over V , there exists an OBDD B over $(V, <)$ such that $e(B)$ and e are equivalent,
2. If B and C be two OBDDs over $(V, <)$, then they are equivalent iff they are isomorphic, and
3. If B be an OBDD over $(V, <)$, and C is an OBDDG over $(V, <)$ such that the two are equivalent, then C contains at least as many vertices as B .

Checking equivalence of two OBDDs, with the same variable ordering, corresponds to checking isomorphism, and hence, can be performed in time linear in the number of vertices. The Boolean constant 0 is represented by an OBDD with a single terminal vertex labeled with 0, and the Boolean constant 1 is represented by an OBDD with a single terminal vertex labeled with 1. A Boolean expression represented by an OBDD B is satisfiable iff the root of B is not a terminal vertex labeled with 0. A Boolean expression represented by an OBDD B is valid iff the root of B is a terminal vertex labeled with 1. Thus, checking satisfiability or validity of Boolean expressions is particularly easy, if we use OBDD representation. Contrast this with representation as propositional expressions, where satisfiability is NP-complete and validity is coNP-complete.

The size of the OBDD representation of a Boolean expression may be exponential in the number of variables. The size of the OBDD representing a given predicate depends on the choice of the ordering of variables. Consider the expression $(x = y) \wedge (x' = y')$. Figure 2.15 shows two OBDDs for two different orderings. This example illustrates that the ordering can influence the size dramatically: one ordering may result in an OBDD whose size is linear in the number of variables, while another ordering may result in an OBDD whose size is exponential in the number of variables.

Choosing an optimal ordering of variables can lead to exponential saving, however, computing the optimal ordering itself is computationally hard. There are

Boolean functions whose OBDD representation does not depend on the chosen ordering, and the OBDD representation of some functions is exponential in the number of variables, irrespective of the ordering. An example of the former variety is the parity function, while of the latter variety is the multiplication function.

- **Parity.** Given a valuation s to a set V of Boolean variables, the parity function returns 1 precisely when the number of variables x with $s(x) = 1$ is even. If V contains k variables, then irrespective of the chosen ordering \prec , the OBDD for the parity function contains $2k + 1$ vertices.
- **Multiplication.** Consider the set of $2k$ variables $\{x_0, \dots, x_{k-1}, y_0, \dots, y_{k-1}\}$, and for $0 \leq j < 2k$, let $Mult_j$ denote the Boolean function that denotes the j -th bit of the product of the two k -bit inputs, one encoded by the bits x_n and another encoded by the bits y_n . For every ordering \prec of the variables, there exists an index $0 \leq j < 2k$ such that the OBDD for $Mult_j$ has at least $2^{k/8}$ vertices.

Operations on OBDDs

Let us turn our attention to implementing regions as OBDDs. Every vertex of an OBDD is itself an OBDD rooted at that vertex. This suggests that an OBDD can be represented by an index to a global data structure that stores vertices of all the OBDDs such that no two vertices are isomorphic. There are two significant advantages to this scheme, as opposed to maintaining each OBDD as an individual data structure. First, checking isomorphism, or equivalence, corresponds to comparing indices, and does not require traversal of the OBDDs. Second, two non-isomorphic OBDDs may have isomorphic subgraphs, and hence, can share vertices.

Let V be an ordered set of k Boolean variables. The type of OBDDs is `bdd`, which is a pointer or an index to the global data structure `BDDPool`. The type of `BDDPool` is `set(bddnode)`, and it stores the vertices of OBDDs. The vertices of OBDDs have type `bddnode` which equals $([1..k] \times \text{bdd} \times \text{bdd}) \cup \text{bool}$. The type `bddnode` supports the following operations:

- The operation $label(u)$, for an internal vertex u , returns the first component of u , which is the index of the variable labeling u .
- The operation $left(u)$, for an internal vertex u , returns the second component of u , which is a pointer to the global data structure `BDDPool` that points to the left-successor of u .
- The operation $right(u)$, for an internal vertex u , returns the third component of u , which is a pointer to the global data structure `BDDPool` that points to the right-successor of u .

The type `set(bddnode)`, apart from usual operations such as `Insert` and `Contains`, also supports

```

function AddNode
Input: Index  $j$  in  $[1..k]$ , OBDDs  $B_0, B_1$  of type bdd.
Output: OBDD  $B$  such that  $e(B)$  is equivalent to  $(\neg x_j \wedge e(B_0)) \vee$ 
       $(x_j \wedge e(B_1))$ .

if  $B_0 = B_1$  then return  $B_0$ ;
if  $\text{Contains}((j, B_0, B_1), \text{BDDPool}) = 0$  then
  Insert $((j, B_0, B_1), \text{BDDPool})$ ;
return  $\text{index}(j, B_0, B_1)$ .

```

Figure 2.16: Creating OBDD vertices

- For a vertex u in BDDPool , $\text{index}(u)$ returns a pointer to u .
- The operation $\text{BDDPool}[B]$ returns the root vertex of the OBDD B .

For such a representation, given a pointer B of type **bdd**, we write $e(B)$ to denote the propositional expression associated with the OBDD that B points to. To avoid duplication of isomorphic nodes while manipulating OBDDs, it is necessary that new vertices are created using the function *AddNode* of Figure 2.16. If no two vertices in the global set BDDPool were isomorphic before an invocation of the function *AddNode*, then even after the invocation, no two vertices in BDDPool are isomorphic. The global set BDDPool initially contains only two terminal vertices, and internal vertices are added only using *AddNode*.

To be able to build an OBDD-representation of a given propositional expression, and to implement the primitives of the symbolic reachability algorithm, we need a way to construct conjunctions and disjunctions of OBDDs. We give a recursive algorithm for obtaining conjunction of OBDDs. The algorithm is shown in Figure 2.17.

Consider two vertices u and v , and we wish to compute the conjunction $e(u) \wedge e(v)$. If one of them is a terminal vertex, then the result can be determined immediately. For instance, if u is the terminal vertex labeled with 0, then the conjunction is also the terminal vertex 0. If u is the terminal vertex labeled with 1, then the conjunction is equivalent to $e(v)$.

The interesting case is when both u and v are internal vertices. Let j be the minimum of the indices labeling u and v . Then, x_j is the least variable that the function $e(u) \wedge e(v)$ can depend on. The label of the root of the conjunction is j , the left-successor is the OBDD for $(e(u) \wedge e(v))[x_j \mapsto 0]$, and the right-successor is the OBDD for $(e(u) \wedge e(v))[x_j \mapsto 1]$. Let us consider the left-successor. Observe the equivalence

$$(e(u) \wedge e(v))[x_j \mapsto 0] \equiv e(u)[x_j \mapsto 0] \wedge e(v)[x_j \mapsto 0].$$

If u is labeled with j , the OBDD for $e(u)[x_j \mapsto 0]$ is the left-successor of u . If the label of u exceeds j , then the function $e(u)$ does not depend on x_j , and the

```

Input: bdd  $B_0, B_1$ .
Output: bdd  $B$  such that  $e(B)$  is equivalent to  $e(B_0) \wedge e(B_1)$ .

table[(bdd  $\times$  bdd)  $\times$  bdd] Done = EmptyTable
return Conj( $B_0, B_1$ ).

bdd Conj(bdd  $B_0, B_1$ )
  bddnode  $u_0, u_1$ ; bdd  $B, B_{00}, B_{01}, B_{10}, B_{11}$ ; [1 . . .  $k$ ]  $j_0, j_1$ 
   $u_0 = \text{BDDPool}[B_0]$ ;
   $u_1 = \text{BDDPool}[B_1]$ ;
  if  $u_0 = 0$  or  $u_1 = 1$  then return  $B_0$ ;
  if  $u_0 = 1$  or  $u_1 = 0$  then return  $B_1$ ;
  if Done[( $B_0, B_1$ )]  $\neq \perp$  then return Done[( $B_0, B_1$ )];
  if Done[( $B_1, B_0$ )]  $\neq \perp$  then return Done[( $B_1, B_0$ )];
   $j_0 = \text{label}(u_0)$ ;  $B_{00} = \text{left}(u_0)$ ;  $B_{01} = \text{right}(u_0)$ 
   $j_1 = \text{label}(u_1)$ ;  $B_{10} = \text{left}(u_1)$ ;  $B_{11} = \text{right}(u_1)$ 
  if  $j_0 = j_1$  then  $B = \text{AddNode}$  ( $j_0, \text{Conj}(B_{00}, B_{10}),$ 
     $\text{Conj}(B_{01}, B_{11})$ );
  if  $j_0 < j_1$  then  $B = \text{AddNode}$  ( $j_0, \text{Conj}(B_{00}, B_1), \text{Conj}(B_{01}, B_1)$ );
  if  $j_0 > j_1$  then  $B = \text{AddNode}$  ( $j_1, \text{Conj}(B_0, B_{10}), \text{Conj}(B_0, B_{11})$ );
  Done[( $B_0, B_1$ )] =  $B$ ;
  return  $B$ .

```

Figure 2.17: Algorithm for taking conjunction of OBDDs

OBDD for $e(u)[x_j \mapsto 0]$ is u itself. The OBDD for $e(v)[x_j \mapsto 0]$ is computed similarly, and then the function *Conj* is applied recursively to compute the conjunction according to the expression above.

The above described recursion may call the function *Conj* repeatedly with the same two arguments. To avoid unnecessary computation, a table is used that stores the arguments and the corresponding result of each invocation of *Conj*. When *Conj* is invoked with input arguments u and v , it first consults the table to check if the conjunction of $e(u)$ and $e(v)$ was previously computed. The actual recursive computation is performed only the first time, and the result is entered into the table.

A *table* data structure stores values that are indexed by keys. If the type of values stored is **value**, and the type of the indexing keys is **key**, then the type of the table is **table**[**key** \times **value**]. This data type supports the retrieval and update operations like arrays: $T[k]$ is the value stored in the table T with the key k , and the assignment $T[k] = m$ updates the value stored in T for the key k . The constant table **EmptyTable** has the default value \perp stored with every key. Tables can be implemented as arrays or as hash-tables. The table used by the algorithm uses a pair of OBDDs as a key, and stores OBDDs as values.

Let us analyze the time-complexity of the algorithm of Figure 2.17. Suppose

the OBDD pointed to by B_0 has n_0 vertices and the OBDD pointed to by B_1 has n_1 vertices. Let us assume that the implementation of the set $BDDPool$ supports constant time membership tests and insertions, and the table $Done$ supports constant-time creation, access, and update. Then, within each invocation of $Conj$, all the steps, apart from the recursive calls, are performed within constant time. Thus, the time-complexity of the algorithm is the same, within a constant factor, of the total number of invocations of $Conj$. For any pair of vertices, the function $Conj$ produces two recursive calls only the first time $Conj$ is invoked with this pair as input, and zero recursive calls during the subsequent invocations. This gives an overall time-complexity of $O(n_0 \cdot n_1)$.

Theorem 2.5 [OBDD Conjunction] *Given two OBDDs B_0 and B_1 , the algorithm of Figure 2.17 correctly computes the OBDD for $e(B_0) \wedge e(B_1)$. If the OBDD pointed to by B_0 has n_0 vertices and the OBDD pointed to by B_1 has n_1 vertices, then the time-complexity of the algorithm is $O(n_0 \cdot n_1)$.*

2.4.3 Exercises

1. The symbolic breadth-first-search algorithm of Figure 2.11 is a *forward search* algorithm that computes the set of states reachable from the initial states by repeatedly applying the image computation operator $Post$. Define a *pre-image* computation Pre : given a region A , $Pre(A, Trans)$ is a region over S that contains precisely those states s for which there is a transition $s \rightarrow t$ of T for some state t in A . Develop a *backward-search* algorithm for the invariant verification problem that starts with the states that violate the desired invariant, and computes the set of states that can reach the violating states by repeatedly applying the pre-image computation operator pre .
2. Suppose we want to modify the symbolic breadth-first-search algorithm of Figure 2.11 so that when it finds a violation of the property φ , it outputs an execution as a counter-example. Which additional operations on regions will be needed for this purpose? Using these operations modify the algorithm so that it outputs a counter-example.
3. Consider the Boolean expression

$$(x_1 \wedge x_2 \wedge x_3) \vee (\neg x_2 \wedge x_4) \vee (\neg x_3 \wedge x_4)$$

Choose a variable ordering for the variables $\{x_1, x_2, x_3, x_4\}$, and draw the resulting OBDD. Can you reduce the size of the OBDD by reordering the variables?

4. Let V be the set $\{x_0, x_1, y_0, y_1, z_0, z_1, c\}$. Choose an appropriate ordering of the variables, and construct the OBDD for the requirement that the output $z_1 z_0$, together with the carry bit c , is the sum of the inputs $x_1 x_0$ and $y_1 y_0$. Is your choice of ordering optimal?

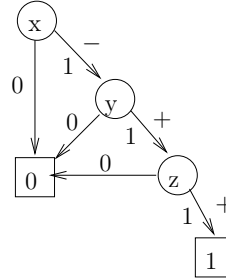


Figure 2.18: A decision graph with complement edges

5. An ordered binary decision graph with *complement edges* (CBDG) is an ordered binary decision graph B with an additional $-$ component that classifies each right-edge as positive $+$ or negative $-$. The predicate $e(u)$, for an internal vertex u , is redefined so that $e(u)$ equals $(\neg \text{label}(u) \wedge e(\text{left}(u))) \vee (\text{label}(u) \wedge e(\text{right}(u)))$ if the right-edge of u is positive, and $(\neg \text{label}(u) \wedge e(\text{left}(u))) \vee (\text{label}(u) \wedge \neg e(\text{right}(u)))$ otherwise. Thus, when the right-edge is negative, we negate the function associated with the right-child. For instance, in Figure 2.18, the vertex labeled with y represents the function $y \wedge z$, while the root represents the function $(x \wedge \neg(y \wedge z))$.

- (1) Define ordered binary decision diagrams with complement edges (CBDD) as a subclass of CBDGs such that every boolean function has a unique representation as a CBDD. (2) Is there a function whose CBDD representation is smaller than its OBDD representation? (3) Suppose we store vertices of all the functions in the same global pool. Show that CBDD representation uses less space than OBDDs. (4) Show that the canonicity property is not possible if we allow complementing left-edges also.
6. The time complexity of the algorithm of Figure 2.17 is proportional to the product of the number of vertices in the component OBDDs. Show that the size of the OBDD representing conjunction of two OBDDs grows as the product of the sizes of the components, in the worst case.
7. We have discussed the algorithm for taking conjunction of two OBDDs. Algorithms for other operations on OBDDs can be developed in a similar manner. (1) Let e be a propositional expression, x be a variable, and $m \in \text{bool}$ be a value. Give an algorithm to construct the OBDD representation of $e[x \mapsto m]$, given the OBDD-representation of e . (2) Give algorithms for computing the disjunction and existential-quantifier elimination for OBDDs.