3

Safety Requirements

A reactive component interacts with the environment via inputs and outputs. A requirement for a component is a specification of acceptable or desired sequences of outputs in response to inputs. Design of high-assurance systems demands that requirements should be stated explicitly and as precisely as possible. 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 2.4.3, the main safety requirement is that no two nodes should ever declare themselves to be the leaders, and the main liveness requirement is that some node should eventually declare itself to be the leader, and the remaining nodes should eventually declare themselves to be the followers. Given a specific solution to the leader election problem, such as the one described in figure 2.35, the *verification* problem is to check whether the given implementation 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 verification problem then reduces to checking whether there is some execution of the system that leads the monitor to an error state. In this chapter, we will first study how reachability problems can be used to formalize safety requirements, and then we will explore verification techniques for establishing correctness of systems with respect to safety requirements.

3.1 Safety Specifications

3.1.1 Invariants of Transition Systems

A safety requirement for a system classifies its states into *safe* and *unsafe* and asserts that an unsafe state is never encountered during an execution of the system. Since the concept of such requirements and the tools for establishing

correctness of systems with respect to such requirements do not specifically rely on the synchronous nature of interaction among reactive components, let us study them in a more general context of *transition systems*.

Transition Systems

A transition system is specified using variables whose valuations describe possible states of the system. The initialization describes the initial values for each of the system variables. Transitions of the system describe how the state evolves and are typically specified using a sequence of assignments and conditional statements that update the state variables, possibly using additional local variables. Following the convention analogous to the definition of synchronous reactive components, we use *Init* to denote the syntactic description of the initialization, with an associated semantics $[\![\mathit{Init}]\!]$ denoting the corresponding set of initial states. Similarly, *Trans* denotes the syntactic description of the transitions, and the associated semantics $[\![\mathit{Trans}]\!]$ is a set of pairs of states.

Transition System

A *transition system* T has:

- a finite set S of typed state variables defining the set Q_S of states,
- an initialization *Init* defining the set $[\![\mathit{Init}]\!] \subseteq Q$ of initial states, and
- a transition description *Trans* defining the set $[\![\mathit{Trans}]\!] \subseteq Q_S \times Q_S$ of transitions between states.

Synchronous Reactive Components as Transition Systems

With each synchronous reactive component $C = (I, O, S, \mathit{Init}, \mathit{React})$, there is a naturally associated transition system: the set of state variables is S; the initialization is *Init*; and the transition description *Trans* is obtained from the reaction description *React* by declaring the input and output variables to be local variables, where the input variables are assigned nondeterministically chosen values. Consequently, the set of transitions contains pairs of states (s, t) such that $s \xrightarrow{i/o} t$ is a reaction for some input i and some output o.

For example, consider the component `TriggeredCopy` of figure 2.5. In the corresponding transition system, the set of state variables is $\{x\}$. The initialization is given by the assignment $x := 0$. The transition description is obtained by declaring the variables *in* and *out* as local and letting the input take every possible value using the `choose` construct:

```
local event(bool) in, out;
  in := choose {0, 1, ⊥};
  if in? then {out! in; x := x + 1}.
```

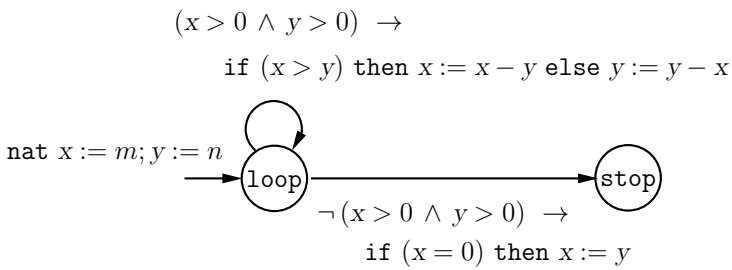


Figure 3.1: Euclid's GCD Program

In any given state, if the input event is present, then the value of x is incremented, and if the input event is absent, then the value of x stays unchanged. Thus, the corresponding transition system has transitions (n, n) and $(n, n+1)$ for every natural number n.

Programs as Transition Systems

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. Given two input numbers m and n, the algorithm computes their GCD using two variables x and y, both of type `nat`. The program executes the following code:

$x := m;\ y := n;$
`while` $(x > 0 \ \wedge \ y > 0)$
  `if` $(x > y)$ `then` $x := x - y$ `else` $y := y - x;$
`if` $(x = 0)$ `then` $x := y.$

The variable x contains the desired answer when the program terminates.

The program can be modeled as an extended-state machine shown in figure 3.1. In the initial mode, denoted `loop`, the system repeatedly decreases either x or y as long as the condition $(x > 0 \wedge y > 0)$ is true, and when the condition is false, it switches to the terminal mode `stop` and changes the variable x as needed so that it contains the desired answer. Note that the modes correspond to *program locations*, and such a representation of a program by an extended-state machine is sometimes called the *control-flow-graph* of the program.

We can used the extended-state machine representation of the program to associate a transition system with it. For given input numbers m and n, the behavior of the GCD program is captured by the transition system $\texttt{GCD}(m, n)$, whose description is parameterized by the numbers m and n. The state variables are x and y of type `nat` and the mode ranging over $\{\texttt{loop}, \texttt{stop}\}$. The sole initial state is $(m, n, \texttt{loop})$. Consider a state s of the form $(j, k, \texttt{loop})$. If $j > k > 0$, then the state s has a transition to state $(j-k, k, \texttt{loop})$; if $k \geq j > 0$,

then the state s has a transition to state $(j, k-j, \texttt{loop})$; if $j = 0$, then the state s has a transition to state $(k, k, \texttt{stop})$; and if $j > 0$ and $k = 0$, then the state s has a transition to state $(j, k, \texttt{stop})$. A state in which the mode equals `stop` has no outgoing transitions.

Reachable States

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 States of Transition System
>
> An *execution* of a transition system T consists of a finite sequence of the form $s_0, s_1, \ldots s_k$, 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}, 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 example, for $m = 6$ and $n = 4$, the transition system $\texttt{GCD}(6, 4)$ has the following execution

$$(6, 4, \texttt{loop}) \;\rightarrow\; (2, 4, \texttt{loop}) \;\rightarrow\; (2, 2, \texttt{loop}) \;\rightarrow\; (2, 0, \texttt{loop}) \;\rightarrow\; (2, 0, \texttt{stop}).$$

All the reachable states are the states appearing in this execution.

Invariants

For a transition system T, a *property* is a Boolean-valued expression over the state variables of T. A state q of T *satisfies* the property φ if φ evaluates to 1 when all variables are assigned values according to the valuation q. The set of all states that satisfy the property φ is denoted $[\![\varphi]\!]$.

Let us revisit the program computing the GCD of two natural numbers. Consider the following property of the transition system $\texttt{GCD}(m, n)$ (see figure 3.1):

$$\varphi_{gcd} \;:\; \texttt{gcd}(m, n) \;=\; \texttt{gcd}(x, y),$$

where `gcd` represents the mathematical function that returns the greatest common divisor of its two arguments. The expression represents exactly those states in which the `gcd` of the values of x and y in that state equals the `gcd` of the parameters m and n (that is, the inputs to the GCD program).

A property is said to be an *invariant* of the transition system if all the reachable states of the system satisfy the property. For our example GCD program, the

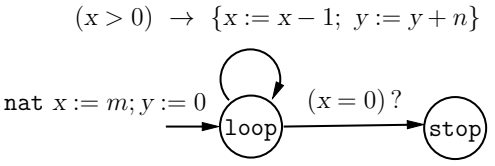


Figure 3.2: Program for Multiplication

property φ_{gcd} is indeed an invariant of the transition system $\texttt{GCD}(m, n)$. This invariant captures the core logic of the program: during the execution of the program, even though the values of the state variables x and y change, their `gcd` stays the same. Since $\texttt{gcd}(p, 0)$ equals p, for every natural number p, it follows that when the variable x has the value 0, $\texttt{gcd}(x, y)$ is the same as y. By a symmetric argument, when the variable y has the value 0, $\texttt{gcd}(x, y)$ is the same as x. This means that the update to the variable x when the system terminates by switching to the mode `stop` is the desired answer. Thus, the implication

$$(\mathit{mode} \;=\; \texttt{stop}) \;\rightarrow\; (\texttt{gcd}(m, n) \;=\; x)$$

is an invariant of the transition system $\texttt{GCD}(m, n)$.

> Invariant of Transition System
>
> For a transition system T, a *property* φ of T is an *invariant* of T if every reachable state of T satisfies φ.

If we denote the set of reachable states of the transition system T by $\mathit{Reach}(T)$, then a property φ is an invariant exactly when the set-inclusion $\mathit{Reach}(T) \subseteq [\![\varphi]\!]$ holds. The dual of the notion of an invariant property is the concept of a *reachable* property: a property φ of a transition system T is *reachable* if *some* reachable state of T satisfies φ. In other words, a property φ is reachable when the intersection $\mathit{Reach}(T) \cap [\![\varphi]\!]$ is a non-empty set. From the definitions, it follows that:

> A property φ of a transition system T is an invariant if and only if the negated property $\neg\,\varphi$ is not reachable.

As an another example, consider the cruise controller from section 2.4.2 and consider the following property:

$$\varphi_{range}: \; \texttt{minSpeed} \leq \texttt{SetSpeed}.s \leq \texttt{maxSpeed}.$$

It says that the state variable s of the component `SetSpeed` is guaranteed to be between the threshold values given by `minSpeed` and `maxSpeed`. This property is an invariant of the system. This is because whenever the component `SetSpeed` updates its state variable s, to initialize in response to the input event *cruise*,

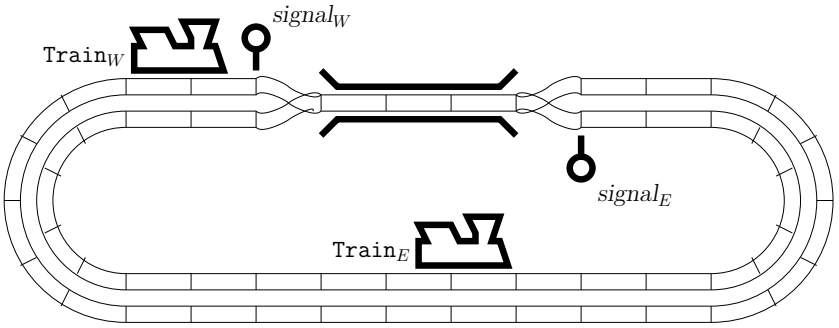


Figure 3.3: Railroad Controller Example

increment in response to the input event *inc*, or decrement in response to the input event *dec*, it checks to ensure that the updated value is in the interval [`minSpeed`, `maxSpeed`].

The invariant verification problem is the following: given a 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 the state 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 *counterexample* to the claim that the property φ is an invariant and, equivalently, a *witness* to the claim that the property $\neg\,\varphi$ is reachable.

Exercise 3.1 : Given two natural numbers m and n, consider the program `Mult` that multiplies the input numbers using two variables x and y, of type `nat`, as shown in figure 3.2. Describe the transition system `Mult`(m, n) that captures the behavior of this program on input numbers m and n, that is, describe the states, initial states, and transitions. Argue that when the value of the variable x is 0, the value of the variable y must equal the product of the input numbers m and n, that is, the following property is an invariant of this transition system:

$$(mode = \texttt{stop}) \;\rightarrow\; (y = m \cdot n)$$

■

3.1.2 Role of Requirements in System Design

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

Specification of the Railroad Controller

Figure 3.3 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,

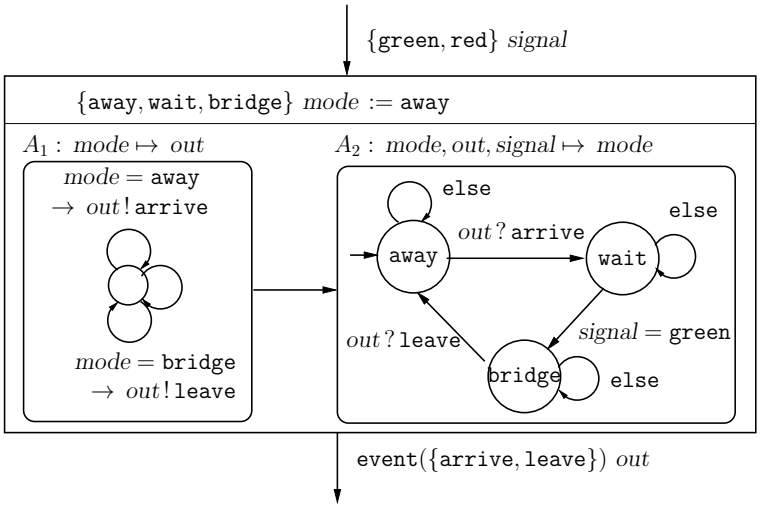


Figure 3.4: Modeling the Train as a Nondeterministic Reactive Component

there is a bridge that 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 each entrance. If the signal at the western entrance is green, then a train coming from the west may enter the bridge, and 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 3.4. The state of the train, captured by the enumerated variable *mode*, 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 output event with the value `arrive` and updates the state to waiting. When the train is waiting, it checks the signal. If the signal is red, then the train keeps waiting, and if the signal is green, then 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 issues an output event with the value `leave` and updates the state to `away`.

The reactions of the train component can naturally be described using an extended-state machine with three modes corresponding to *away*, *wait*, and *bridge*. However, specifying the update as a single task would create an await dependency of the output event on the input signal. To avoid this, the component specification of figure 3.4 splits the reaction description into two tasks. The first task A_1 computes the value of the output variable *out*, and this does not depend on the input variable *signal*. The task A_1 is nondeterministic: when

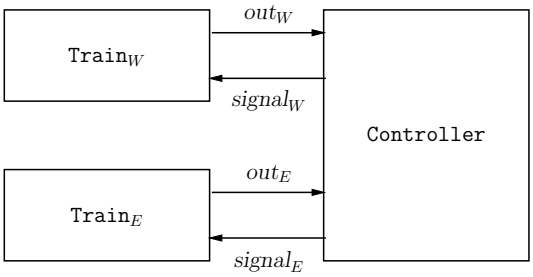


Figure 3.5: Composite System for the Railroad Controller

the mode is *away*, the output can be absent or present with the value `arrive`; when the mode is *wait*, the output is absent; when the mode is *bridge*, the output can be either absent or present with the value `leave`. This description is captured by the single-mode extended-state machine in figure 3.4. Recall that for a mode-switch, absence of a guard condition means that the mode-switch is always enabled (that is, by default, the guard condition is the constant 1 that is satisfied in every state), and absence of an associated update means that state variables do not change and event outputs are absent. The second task A_2 updates the mode based on the output computed by the task A_1 and the value of the input *signal*. In the mode `away`, when the guard-condition $out\,?\,\texttt{arrive}$ holds, the mode is updated to `wait`. The condition `else` on the self-loop is an abbreviation for the negated condition $\neg\,(out\,?\,\texttt{arrive})$. In general, the guard-condition `else` on a self-loop on a mode is satisfied exactly when none of the guard-conditions of the mode-switches out of this mode is satisfied. The mode-switches out of the modes `wait` and `bridge` are similar.

Since there are two trains, one traveling clockwise and the other traveling counterclockwise, we create two instances of the train component, $\texttt{Train}_W$ and $\texttt{Train}_E$.

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. More specifically, 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, we get the composite system

$$\texttt{RailRoadSystem} \;=\; \texttt{Controller} \parallel \texttt{Train}_W \parallel \texttt{Train}_E$$

shown in figure 3.5. Note that, irrespective of the await dependencies of the controller, there will be no cycles in await dependencies in these three components, and thus the above composition is well defined.

The controller should be designed so that the property

$$\texttt{TrainSafety}: \;\neg\,(\,mode_W \;=\; \texttt{bridge} \;\wedge\; mode_E \;=\; \texttt{bridge}\,)$$

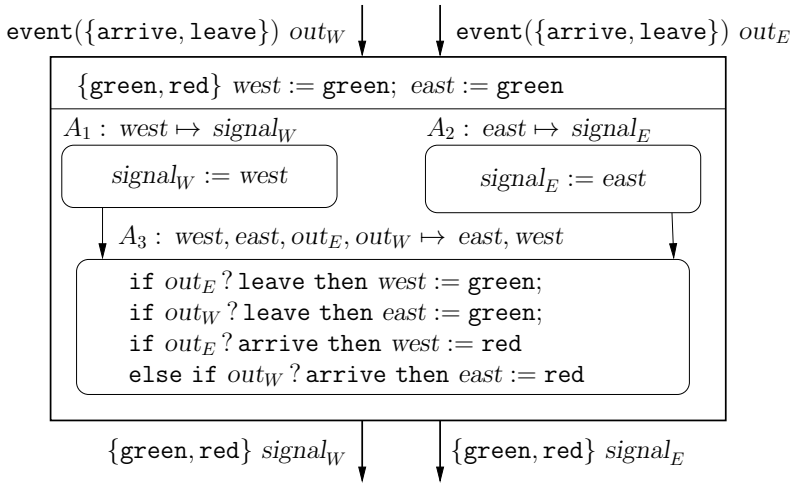


Figure 3.6: A First Attempt at Design of the Railroad Controller

is an invariant of `RailRoadSystem`. Here, the state variables $mode_W$ and $mode_E$ are the state variables of the two instances of the train component.

A First Attempt at the Design of the Railroad Controller

Figure 3.6 shows a first attempt at designing the railroad controller. The controller `Controller1` maintains two state variables *west* and *east* for the states of the two output signals $signal_W$ and $signal_E$, respectively, and in each round, the output variable is set to the value of the corresponding state variable. Initially, both signals are green. A signal is set to red whenever a train approaches the opposite entrance to the bridge, and it is set 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 the east. The update is split into three tasks: the tasks A_1 and A_2 output the values of the respective signals without waiting for any input, and the task A_3 then updates the state variables based on the input events.

Unfortunately, the resulting railroad system

$$\texttt{RailRoadSystem1} \;=\; \texttt{Controller1} \parallel \texttt{Train}_W \parallel \texttt{Train}_E$$

does not satisfy the desired invariant `TrainSafety`. This is evidenced by the counterexample shown in figure 3.7, 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 to proceed to the bridge. However, the east signal is still

west	*east*	$mode_W$	$mode_E$	$signal_W$	$signal_E$	out_W	out_E
green	green	away	away				
				green	green	arrive	arrive
red	green	wait	wait				
				red	green	$\perp$	$\perp$
red	green	wait	bridge				
				red	green	$\perp$	leave
green	green	wait	away				
				green	green	$\perp$	arrive
red	green	bridge	wait				
				red	green	$\perp$	$\perp$
red	green	bridge	bridge				

Figure 3.7: An Execution of `RailRoadSystem1` That Violates `TrainSafety`

green. So if the east train returns *before* the west train has left the bridge, the west signal will turn red while admitting the east train onto the bridge, leading to a violation of the safety requirement.

A Second Attempt at the Design of the Railroad Controller

Figure 3.8 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 $mode_W$ 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. Consider the case when both signals are red and both trains issue `arrive`. Then both *near* variables are set to 1. In this case, 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

$$\texttt{RailRoadSystem2} \;=\; \texttt{Controller2} \parallel \texttt{Train}_W \parallel \texttt{Train}_E,$$

the property `TrainSafety` is indeed an invariant as desired.

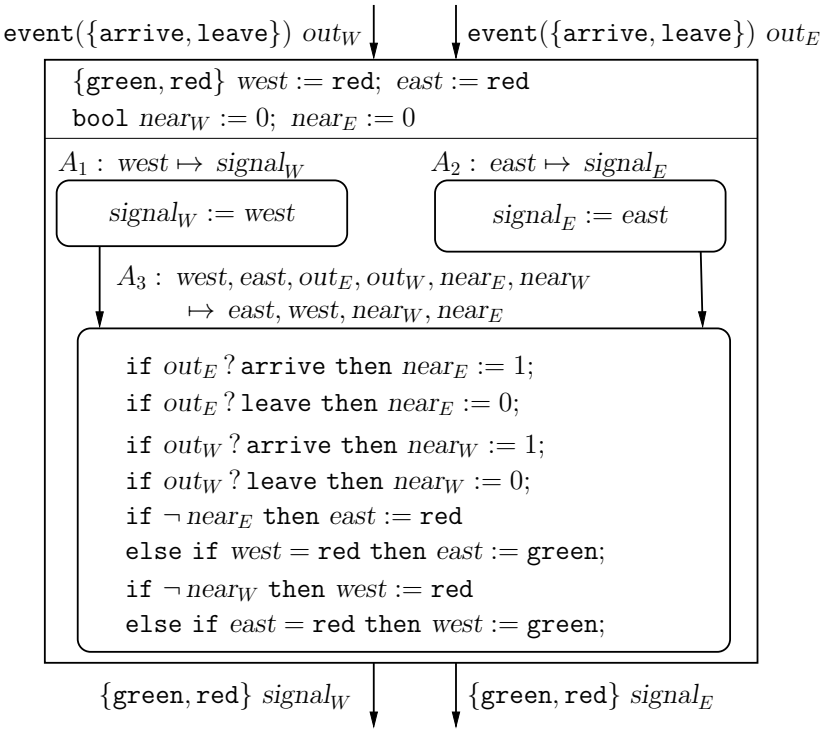


Figure 3.8: A Safe Controller for the Railroad Problem

Exercise 3.2: The composed system `RailRoadSystem1` has four state variables, *east* and *west*, each of which can take two values, and $mode_W$ and $mode_E$, each of which can take three values. Thus, `RailRoadSystem1` has 36 states. How many of these 36 states are reachable? ■

Exercise 3.3: The reaction description for the controller `Controller2` consists of three tasks as shown in figure 3.8. Split the task A_3 into four tasks, each of which writes exactly one of the state variables *east*, *west*, $near_W$, and $near_E$. Each task should be described by its read-set, write-set, and update code, along with the necessary precedence constraints. The revised description should have the same set of reactions as the original description. Does this splitting impact output/input await dependencies? If not, what would be the potential benefits and/or drawbacks of the revised description compared to the original description? ■

3.1.3 Safety Monitors

For our railroad crossing example, suppose we have an additional "fairness" requirement that if a train arrives at a bridge and as it waits for its signal to

turn green, the other train should not be allowed to enter the bridge repeatedly. More 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 along an execution of the railroad system, but it cannot be formulated as an invariant directly. It can, however, be stated as an invariant if we add another component, `WestFairMonitor`, shown in figure 3.9.

The monitor is described as an extended-state machine with four possible modes. The mode is initially 0. When the west train arrives, the mode changes to 1. If the east train leaves the bridge, then the mode changes to 2, and if the east train leaves the bridge again, then the mode changes to 3. In modes 1 and 2, if the west signal is turned green, then the mode is reset to 0. If there is an execution in which the monitor's mode gets updated to 3, then it demonstrates a violation of the desired fairness requirement with respect to the west train. The mode 3 of the monitor is marked as an *accepting mode* of the monitor (this is analogous to the final states of automata in theory of formal languages). An execution that reaches this accepting mode corresponds to a counterexample to the desired safety requirement. To check whether there is such a violation, we can determine whether the property $\texttt{WestFairMonitor}.mode = 3$ is reachable in the composite system $\texttt{RailRoadSystem} \parallel \texttt{WestFairMonitor}$. To ensure fairness with respect to the east train, we can compose the system with a symmetric version of the monitor, which can be defined by renaming the input variables of `WestFairMonitor`.

The definition of such monitors is summarized below.

Safety Monitor

A *safety monitor* for a reactive component C with input variables I and output variables O consists of a synchronous reactive component M such that:

- the set of input variables of M is a subset of the variables $I \cup O$,
- the set of output variables of M is disjoint from the variables $I \cup O$,
- and the reaction description of M is given as an extended-state machine, along with a subset F of the modes declared as accepting.

The component C satisfies the monitor specification if the property $M.mode \notin F$ is an invariant of the composed system $C \parallel M$.

The requirement that the input variables of the monitor M are the input/output variables of the component C means that M can *observe* the behavior of C in terms of its interaction with the other components. The requirement that the output variables of M are neither the inputs nor the outputs of C ensures that the behavior of C is not modified by the monitor M and also that it is compatible with M. We design the monitor so that it enters an error mode in the set F

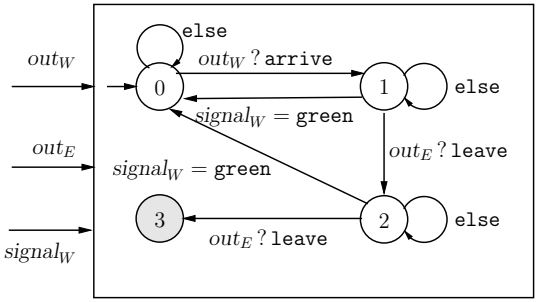


Figure 3.9: Fairness Monitor `WestFairMonitor` for the West Train

when the observed sequence of inputs and corresponding outputs violates the desired safety requirement.

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` as well as the requirement expressed by the safety monitor of figure 3.9. 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 10, rounds of the execution cannot be considered a violation of the response requirement. Indeed, hypothetically, 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 5.

Exercise 3.4: Consider a component C with an output variable x of type `int`. Design a safety monitor to capture the requirement that the sequence of values output by the component C is strictly increasing (that is, the output in each round should be strictly greater than the output in the preceding round). ■

Exercise 3.5: Does the second attempt to design the railroad controller satisfy the fairness requirement captured by the monitor `WestFairMonitor`? That is, is the property `WestFairMonitor`.*mode* $\neq 3$ an invariant of the composite system `RailRoadSystem2` $\|$ `WestFairMonitor`? If not, show a counterexample execution. ■

3.2 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 counterexample that demonstrates the reachability of a state violating the property. We first describe a general purpose proof methodology for establishing invariants and then consider the challenge of developing automatic tools for solving the invariant verification problem.

3.2.1 Proving Invariants

Inductive Invariants

Consider a transition system T and a property φ. If we can establish that φ holds initially and is preserved during every transition, then by the principle of mathematical 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 initial state satisfies φ.

Showing that the property is preserved by every transition amounts to establishing that,

> if a state s satisfies φ, and (s, t) is a transition of T, then the state t satisfies φ.

Notice the similarity of these two conditions with the classical proofs by induction. To show that a property holds for all natural numbers n, we first show that the property holds for 0 (this is called the *base case*), and then, assuming that the property holds for a number k, we show that the property also holds for the number $k + 1$ (this is called the *inductive case*). In case of properties of reachable states of a transition system, the base case corresponds to proving the property for the initial states, and the inductive case corresponds to, assuming the property holds for an arbitrary state s, proving that the property holds for any state t that has a transition from the state s.

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 is an *inductive invariant* of T if:
>
> 1. every initial state s satisfies φ, and
> 2. if a state s satisfies φ and (s, t) is a transition, then the state t also satisfies φ.

Let us consider the program GCD of figure 3.1. For the transition system $\texttt{GCD}(m, n)$, consider the property φ_{gcd} given by $\texttt{gcd}(m, n) = \texttt{gcd}(x, y)$. To show

that this is an inductive invariant, let us first consider the requirement corresponding to initialization. In this case, there is only one initial state s with $s(x) = m$ and $s(y) = n$, and this initial state clearly satisfies the property. Now let us focus on the inductive case. Consider a state s that satisfies the desired property φ_{gcd}. Let $s(x) = a$ and $s(y) = b$. By assumption, $\texttt{gcd}(m, n) = \texttt{gcd}(a, b)$ holds. If $s(\mathit{mode}) = \texttt{stop}$, then the state s has no transition from it. Now suppose $s(\mathit{mode}) = \texttt{loop}$, and consider a transition (s, t) of the system $\texttt{GCD}(m, n)$. To show that the new state t satisfies the property φ_{gcd}, it suffices to show that $\texttt{gcd}(t(x), t(y))$ equals $\texttt{gcd}(a, b)$. First, let us consider the case when $a > b > 0$. Then the program decrements x by y, and in this case, $t(x) = a - b$ and $t(y) = b$. So we need to show that $\texttt{gcd}(a - b, b) = \texttt{gcd}(a, b)$. This follows from basic properties of arithmetic: when $a > b$, any number that is a divisor of both a and b must also be a divisor of $a - b$, and any number that is a divisor of both a and $a - b$ must also be a divisor of b. In the case when $a \leq b$ and $b > 0$, $t(x) = a$ and $t(y) = b - a$, and by a symmetric argument $\texttt{gcd}(a, b - a) = \texttt{gcd}(a, b)$. In the case when either $a = 0$ or $b = 0$, the program switches to the mode `stop`. In such a case, either $a > 0$ and $t(x) = a$ and $t(y) = b$ or $a = 0$ and $t(x) = b$ and $t(y) = b$. In both these cases, verify that $\texttt{gcd}(t(x), t(y))$ equals $\texttt{gcd}(a, b)$.

For the cruise-control example from section 2.4.2, the property $\texttt{minSpeed} \leq \texttt{SetSpeed}.s \leq \texttt{maxSpeed}$ is an inductive invariant: it holds in the initial state, and whenever there is a transition (s, t), it holds in state t since the component `SetSpeed` never updates the value of the state variable s without checking the bounds.

Strengthening Invariants

To illustrate that a property may be an invariant but not an inductive invariant, let us consider the transition system $\texttt{IncDec}(m)$, parameterized by a natural number m, shown in figure 3.10. The system uses two variables x and y, both of type `int`. Initially, x is 0 and y is m, and in each transition, the program increments the variable x and decrements the variable y as long as the value of x does not exceed m.

Consider the property $\varphi_x : 0 \leq x \leq m$, which states that the value of the variable x is always within the range from 0 to m. Let us examine whether the property φ_x is an inductive invariant of the transition system $\texttt{IncDec}(m)$. In the sole initial state of $\texttt{IncDec}(m)$, the value of x is 0, and thus the property φ_x holds initially. A state of $\texttt{IncDec}(m)$ is a valuation of the integer variables x and y and thus is a pair of integers, listing the value of x first. Consider an arbitrary state $s = (a, b)$. We want to show that if the state s satisfies the property φ_x and (s, t) is a transition of $\texttt{IncDec}(m)$, then the state t also satisfies φ_x. Assume that the state s satisfies the property φ_x, that is, $0 \leq a \leq m$. If $a < m$, then in one transition from state s, the program increments x and decrements y, and thus the updated state t is $(a + 1, b - 1)$, and in this case, we can conclude that $0 \leq a + 1 \leq m$, and thus the state t continues to satisfy the property φ_x. If the condition $a < m$ does not hold, then executing the update code does not change

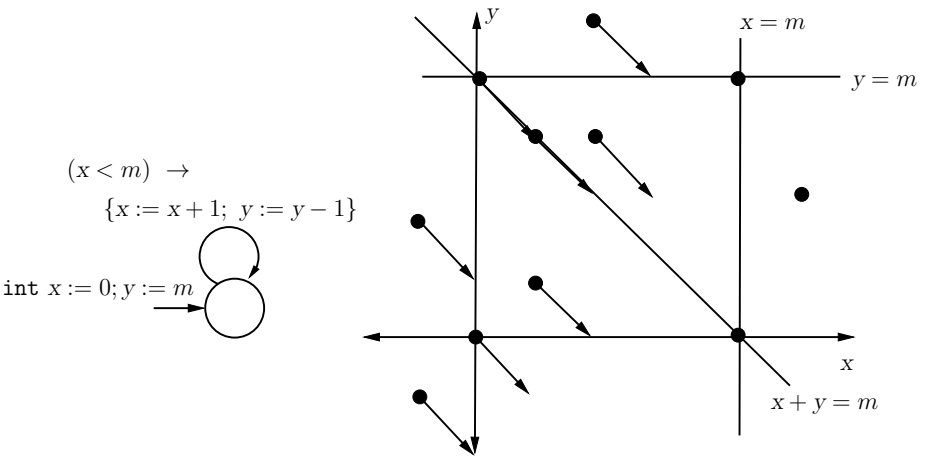


Figure 3.10: Transition System `IncDec`(m)

the state, and thus the property φ_x continues to hold. Hence, the property φ_x is an inductive invariant of the transition system `IncDec`(m).

Now let us examine whether the property $\varphi_y : 0 \leq y \leq m$ is an inductive invariant of the transition system `IncDec`(m). The property φ_y holds in the initial state $(0, m)$. Consider an arbitrary state $s = (a, b)$ of `IncDec`(m). If the only assumption about the state s is that it satisfies φ_y, that is, $0 \leq b \leq m$, then can we conclude that, after one transition of the program, the value of y will still be in the range $[0, m]$? The answer is negative. In particular, the state $(0, 0)$ satisfies φ_y, and executing one transition from the state $(0, 0)$ gives the state $(1, -1)$, which violates φ_y. In conclusion, the property φ_y is *not* an inductive invariant of the transition system `IncDec`(m).

Note, however, that the property φ_y is an invariant of `IncDec`(m): the reachable states of the transition system `IncDec`(m) are $(0, m), (1, m-1), \cdots (m, 0)$, all of which satisfy the property φ_y. That is, although the property φ_y is an invariant of the system, *it is not strong enough to be inductive.* The *inductive strengthening* of the property is φ_{xy}:

$$0 \leq y \leq m \ \wedge \ x + y = m.$$

The property φ_{xy} implies φ_y: if a state satisfies the property φ_{xy}, then clearly it also satisfies φ_y. The property φ_{xy} holds in the initial state $(0, m)$. Now consider a state $s = (a, b)$ that satisfies the property φ_{xy}, that is, assume that $a + b = m$ and $0 \leq b \leq m$. If the condition $a < m$ does not hold, then executing the update code does not change the state, and thus the property φ_{xy} continues to hold. Suppose $a < m$. Then there is a transition from the state s to the state $t = (a + 1, b - 1)$. Since $a + b = m$ and $a < m$, we can conclude that $b > 0$. It

follows that $0 \leq b - 1 \leq m$ and also $(a + 1) + (b - 1) = m$. Thus, the state t satisfies the property φ_{xy}. Hence, the property φ_{xy} is an inductive invariant of the transition system $\texttt{IncDec}(m)$.

Intuitively, since the program decrements y when the condition $x < m$ holds, it is not possible to show that the property φ_y is preserved by its transitions as this property asserts bounds on the values of y without relating it to the values of x. The stronger property φ_{xy} captures the relevant corelation between the values of the two variables and turns out to be inductive.

Figure 3.10 also shows states and transitions of the system visually and is useful to understand the concepts of reachable states, invariants, and inductive invariants. The reachable states of the system are the states on the line segment joining the points $(0, m)$ and $(m, 0)$. A property is an invariant as long as it includes this line segment. Thus, properties such as $0 \leq x \leq m$, $0 \leq y \leq m$, $x + y = m$, and $x \geq 0$ are all invariants, whereas the property $x < m$ is not an invariant. An inductive invariant is any set of states that contains the reachable line segment and has no transitions crossing its boundary from a state inside to a state outside it. Examples of such inductive invariants are $0 \leq x \leq m$, $x \geq 0$, $x \leq m$, $y \leq m$, $x + y = m$, and $x + y \leq m$. Examples of properties that are *not* inductive invariants include $y \geq 0$ and $0 \leq y \leq m \ \wedge \ 0 \leq x \leq m$. Given a property φ, its inductive strengthening is an inductive property that is a subset of φ. Such a strengthening need not be unique: for instance, while φ_{xy} is an inductive strengthening of the property φ_y, so is the property $0 \leq y \leq m \ \wedge \ x + y \geq m$.

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 property ψ implies the property φ (that is, a state satisfying ψ is guaranteed to satisfy φ).

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

The proof rule is also *complete* from a theoretical perspective. What this means is that if a property φ is indeed an invariant, then there does exist an inductive property ψ that implies φ, and thus the proof rule can always be used to establish

the desired invariant. In particular, let ψ_{reach} be the formula 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 thus ψ_{reach} implies φ. The property ψ_{reach} 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 writing formulas is expressive enough to 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. To establish that ψ is indeed inductive and implies φ, one can either use a "paper-and-pencil" argument or construct a formal proof using an automated theorem prover. Identifying such an inductive strengthening requires expertise, but note that this task can be much easier than understanding the precise set of reachable states of the system. For instance, let us modify the transition system $\texttt{IncDec}(m)$ of figure 3.10 by introducing an additional integer variable z, initialized to 0, and by modifying the self-loop to

$$(x < m) \;\rightarrow\; \{\, x := x + 1;\; y := y - 1;\; z := z + x\, y \,\}.$$

To prove that the property $0 \leq y \leq m$ is an invariant of the modified system, it still suffices to consider the strengthening $0 \leq y \leq m \wedge x + y = m$, which turns out to be an inductive invariant for the modified system also. Note that the property characterizing the set of reachable states of the modified system is a lot more complex as it needs to relate the current values of x and z. In other words, to prove that the variable y stays within the bounds of 0 and m using the technique of inductive invariants, it suffices to understand the relationship between x and y, and one can ignore the way the variable z gets updated.

Proof of the Synchronous Leader Election Protocol

To illustrate a more interesting proof, let us consider the central correctness argument for the flooding algorithm for leader election from section 2.4.3. 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 of all the node components $\texttt{SyncLENode}_n$, for each node $n \in P$, each executing the leader election protocol, and the combinational network component $\texttt{Network}_{P,E}$ that transfers the messages in a synchronous manner. The description of the transition system for `SyncLE` is summarized below.

The set of state variables contains the variables id_n and r_n for each node n. The type of all the state variables is `nat`. Initially, the value of each variable r_n is 1, and the value of each variable id_n equals n.

The transition description of `SyncLE` uses the variables in_n, out_n, and $status_n$ for each node n, which are the input/output variables of individual components that are used for communication. The update in each round involves the following sequence of steps:

1. The tasks A_1 of all the node components execute: for each node n, if the condition $r_n < N$ holds, then the value id_n is issued on out_n, and r_n is incremented. Otherwise out_n is absent and r_n stays unchanged.

2. The component `Network` executes updating all the variables in_n: for each node n, the value of in_n equals the set containing the values of the variables out_m, such that there is a network edge from node m to node n, and the event out_m is present.

3. The tasks A_2 of all the node components execute: for each node n, the value of id_n is updated to equal the maximum of its current value and the values contained in the set in_n.

Consider the following property φ_{leader} for the transition system `SyncLE`:

> for every node n, after N rounds, the value of the variable $\mathtt{id}_n$ equals the highest identifier in P: $r_n = N \;\rightarrow\; id_n = \mathtt{max}\ P$.

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

> for every node n, $id_n \;=\; \mathtt{max}\ \{m \mid \mathtt{dist}(m, n) < r_n\}$.

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 r_n equals N, the set $\{m \mid \mathtt{dist}(m, n) < r_n\}$ must equal the set P of all identifiers, and thus the value of id_n must equal the maximum identifier, $\mathtt{max}\ P$.

Let us check whether the property ψ is an inductive invariant of the system `SyncLE`. It holds initially that 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 ψ and a node n. Suppose the value of the round variable r_n equals j, where $j < N$. Then we know that the value of id_n is $\mathtt{max}\{m \,|\, \mathtt{dist}(m, n) < j\}$. In one round, the node n receives the current values of *id* variables of all its neighbors and updates id_n to the maximum of its current value and all the values received. Since the node n increments the variable r_n, we need to show that the updated value of id_n is $\mathtt{max}\{m \,|\, \mathtt{dist}(m, n) < j + 1\}$. 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 j in state s? We know, from the property ψ, that the value of id_{n_2} is the maximum of identifiers of all nodes at distance less than the value of the round variable r_{n_2}. We would be finished with the proof successfully if we could conclude that the value of r_{n_2} in state s is j. But this is not really captured in the property ψ, and the proof fails. The necessary strengthening must also assert 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}:

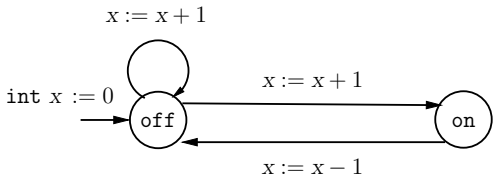


Figure 3.11: Exercise for Inductive Invariants

for every node n, $id_n = \texttt{max}\,\{m \mid \texttt{dist}(m,n) < r_n\}$, and for every pair of nodes m and n, $r_m = r_n$.

Exercise 3.6: Consider a transition system T with two integer variables x and y and a Boolean variable z. All the variables are initially 0. The transitions of the system correspond to executing the conditional statement

$$\texttt{if}\ (z = 0)\ \texttt{then}\ \{x := x + 1;\ z := 1\}\ \texttt{else}\ \{y := y + 1;\ z := 0\}.$$

Consider the property φ given by $(x = y) \vee (x = y + 1)$. Is φ an invariant of the transition system T? Is φ an inductive invariant of the transition system T? Find a formula ψ such that ψ is stronger than φ and is an inductive invariant of the transition system T. Justify your answers. ■

Exercise 3.7: Recall the transition system $\texttt{Mult}(m, n)$ from exercise 3.1. First, show that the invariant property $(mode = \texttt{stop}) \rightarrow (y = m \cdot n)$ is not an inductive invariant. Then find a stronger property that is an inductive invariant. Justify your answers. ■

Exercise 3.8: Consider the transition system specified by the extended-state machine of figure 3.11. Consider the property φ given by $x \geq 0$. Show that φ is *not* an inductive invariant of the system. Find a formula ψ such that ψ is stronger than φ and is an inductive invariant. Prove your answer. ■

Exercise 3.9: Consider the system `RailRoadSystem2` corresponding to the controller of figure 3.8. For each of the properties listed below, state whether the property is an invariant of the system and, if so, whether it is an inductive invariant. Justify your answers with an explanation.

1. The controller state variable $near_E$ is 0 when the east train is away and 1 otherwise: $(near_E = 0) \leftrightarrow (mode_E = \texttt{away})$.
2. When the east train is on the bridge, the controller state variable corresponding to the east signal is green: $mode_E = \texttt{bridge} \rightarrow east = \texttt{green}$.
3. The controller variables for the two signals cannot be green simultaneously: $\neg\,(east = \texttt{green} \wedge west = \texttt{green})$.

■

3.2.2 Automated Invariant Verification *

Before we proceed to consider some techniques for checking invariants of transition systems, let us consider invariant verification as a computational problem. Ideally, we would like to *automate* verification. The challenge then is to build a verification tool as shown in figure 3.12: the input to the verifier consists of the description of the transition system T and a property φ, and it decides whether φ is an invariant of T.

Undecidability

In general, the verification problem is *undecidable*. This means that we cannot hope to have a completely algorithmic solution to solve the invariant verification problem, and the ideal verifier of figure 3.12 does not exist. Intuitively, to check whether a property φ is an invariant of a given transition system T, the verification tool needs to check all reachable states of T. Given the description of T, the tool can systematically explore reachable states and check whether each such state satisfies the property φ. However, when the number of states of a system is unbounded, and this is the case for systems described using variables with unbounded types such as `nat`, the verification tool can potentially run forever exploring more and more states without ever being able to conclude that the property holds in all reachable states. Formally, the undecidability of the verification problem follows from the results in computability theory. In fact, if we limit ourselves to transition systems that have only *counter-variables*, then the invariant verification problem remains 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.

Verification of Finite-State Systems

Now let us restrict our focus on systems whose variables have finite types. For such finite-state systems, the number of all possible states is bounded. In this special case, the invariant verification problem can be automated. The input to the verifier is a description of the transition system in the source modeling language and, thus, is described in a compact manner. In particular, for synchronous reactive components, the set of states is described by listing the names and types of all the variables, and the set of transitions is described by a sequence of assignments and conditional statements. If the transition system T has k state variables, and each variable can take at most m different values, then the bound on the number of states is m^k. The total number of states grows exponentially with the number k of state variables, and as a result, the solution based on examining all states does not lead to a scalable analysis tool. This exponential complexity, however, is intrinsic to the problem.

The precise computational complexity of the invariant verification problem depends on the details of the modeling language used to specify transition systems. As an illustrative case, let us consider transition systems described as sequential

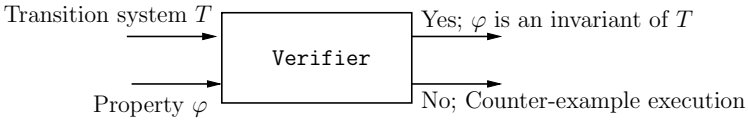


Figure 3.12: Ideal Automated Verification Tool

circuits discussed in section 2.4.1. A sequential circuit is specified using block diagrams connecting Not gates, And gates, and latches. All the variables in such a circuit are of type `bool`. If the circuit has k latches, then the number of possible states is 2^k. Suppose the property φ is specified as a Boolean expression over the state variables corresponding to the latches. In this case, the invariant verification problem is a canonical problem for the complexity class PSPACE of computational problems. PSPACE stands for problems solvable using space polynomial in the size of the input description. The upper bound of PSPACE means that there is an algorithm to solve the invariant verification problem for sequential circuits that requires memory polynomial in the number of latches and gates used in the circuit description. Furthermore, the invariant verification problem is computationally hard for the class PSPACE, implying that every other problem in this class can be reformulated as an invariant verification for sequential circuits.

Theorem 3.1 [Complexity of Finite-state Invariant Verification] *Given a sequential circuit C represented as a block diagram over Not gates, And gates, and latches, and a property φ specified as a Boolean formula over the state variables corresponding to the latches, the computational problem of checking whether the property φ is an invariant of the transition system corresponding to C is* PSPACE*-complete.* ■

Understanding the PSPACE complexity bounds in a rigorous manner would require a detour into computational complexity theory and is beyond the scope of this textbook. We will just note some facts useful in our context. First, the characterization of the complexity of the invariant verification problem is robust: the invariant verification problem is PSPACE-complete for finite-state systems for all typical choices of modeling languages used in practice. The complexity stays the same whether systems are deterministic or nondeterministic. Second, the class PSPACE of problems is a superset of the class NP of problems, which stands for problems that can be solved by *nondeterministic* algorithms in time polynomial in the size of the input. The canonical problem for the class NP is analysis of *combinational* circuits. Consider a combinational circuit C represented as a block diagram over Not and And gates, such that C has m Boolean input variables $x_1, \ldots x_m$ and a Boolean output variable y. Suppose we want to check whether there is an input, that is, an assignment of 0/1 values to the m input variables, so that the output of C is 1. Given a specific input, computing the corresponding output is easy, and we can develop an efficient algorithm for

this purpose that would require time linear in the size of the circuit (that is, the number of gates). However, to check whether there exists *some* input for which the corresponding output is 1, the most obvious strategy to try all inputs would lead to an exponential-time algorithm since there are 2^m possible inputs. This structure makes this analysis problem for combinational circuits belong to the class NP. Furthermore, it is a canonical representative of all NP problems: if a combinational circuit analysis problem can be solved efficiently, then so can every problem in the class NP. Thus, the combinational circuit analysis problem is an NP-complete problem. Whether such a problem can be solved by a polynomial-time algorithm remains a long-standing open problem in computer science. For all practical purposes, we can assume that efficient algorithms do not exist for NP-complete problems and, thus, also not for PSPACE-complete problems, such as the invariant verification problem for finite-state systems. Finally, note that the combinational circuit analysis problem is representative of computing reactions from a given state. For a finite-state system corresponding to a synchronous reactive component, given two states s and t, determining whether there is a transition from state s to state t is typically NP-complete as it would require examining all possible input combinations, but for a specific input i, determining whether state s on input i gets updated to state t can be determined efficiently by executing the update code.

In summary, for finite-state reactive components, analyzing what happens within a round is computationally hard (NP-complete) due to the multitude of combinations of values of input variables, and analyzing what happens in an execution, across multiple rounds, has additional computational difficulty (PSPACE-complete) due to the multitude of combinations of values of state variables.

3.2.3 Simulation-Based Analysis

The most commonly used industrial technique for analyzing systems is exploration 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 whether 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 as well as nondeterministic choices within the reaction description. In such a case, the simulator must resolve the choice in some way, for instance, by using randomization.

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 data structures and operations:

```
Input: A transition system T, a property φ, and an integer k > 0;
Output: If a state violating φ is encountered, return a counterex-
    ample;

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

if s = null then return;
exec[j] := s;
if Satisfies(s, φ) = 0 then return exec;
for j = 1 to k do {
  s := ChooseSuccState(s, T);
  if s = null then return;
  exec[j] := s;
  if Satisfies(s, φ) = 0 then return exec;
  }.
```

Figure 3.13: Simulation-based Invariant Falsification

- States of the transition system are of the type `state`. The constant `null` specifies a dummy state.
- Given a transition system T, *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, *ChooseSuccState*(s, T) returns *some* successor state of s, that is, some state t such that (s, t) is a transition of T, and the dummy state `null` if s has no outgoing transitions.
- Given a property φ and a state s, *Satisfies*(s, φ) returns 1 if the state s satisfies the property φ, and 0 otherwise.

The simulation algorithm is presented in figure 3.13. The variable *exec* is an array of states, and the algorithm fills its entries one by one for the specified number of steps. At each step, the function *Satisfies* is used to check whether the current state violates the desired invariant. Note that if no violation is found, the algorithm cannot make any conclusions about whether the invariant holds. In such a case, we can run the algorithm repeatedly to gain more confidence in the correctness of the system. However, such analysis cannot prove that the system indeed satisfies the invariant. This form of analysis, which can conclusively only disprove the claim that the system satisfies the specified correctness requirement, is called *falsification*.

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

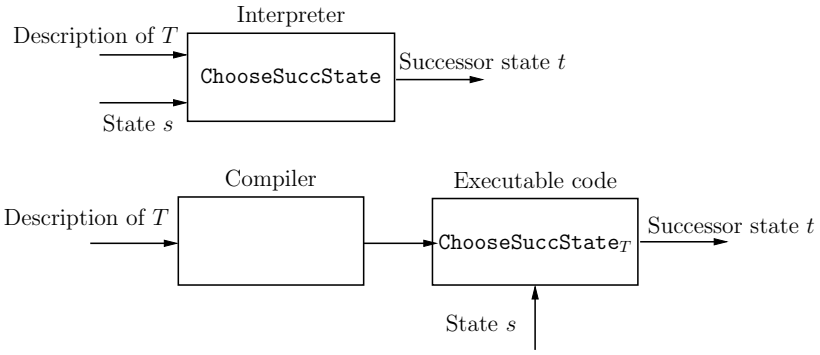


Figure 3.14: Interpretation vs. Compilation for Simulation-based Analysis

representation of the transition system. One possible representation for the transition system, then, is the original description of the system in the source modeling language. For transition systems corresponding to synchronous reactive components, a system description is typically a parallel composition of compatible components, where each component consists of one or more tasks, each with an update description given as straight-line code, along with precedence constraints among tasks. With this choice, the implementation of the function *ChooseSuccState* can be as follows. Given a state, it first picks arbitrary values for the input variables. It then orders all the tasks of all the components in a manner consistent with all the precedence constraints. Then the update specifications of the tasks are executed in the chosen order. If there is a nondeterministic assignment (using the `choose` command) in any update specification, an arbitrary choice is made. This style of implementation is called *model interpretation*: the different constructs in the model description are interpreted as needed during the execution of *ChooseSuccState*.

An alternative strategy is *model compilation*: the original description of the system in the source modeling language is compiled to executable functions for operations such as *ChooseSuccState* that are specialized for this particular transition system. That is, given the description of the transition system T, the model compiler generates an executable function for *ChooseSuccState*$_T$. The input to *ChooseSuccState*$_T$ is a state of T, and its output will be a successor-state of the input state. The two approaches of interpretation and compilation are illustrated in figure 3.14. In the context of synchronous reactive components, the compiled version of *ChooseSuccState*$_T$ contains a fixed order of execution of all the tasks: this order needs to be consistent with all the precedence constraints but is chosen just once when the code for *ChooseSuccState*$_T$ is generated. Nondeterministic constructs such as `choose` in the update specification of individual tasks are replaced with suitably chosen calls to random-number generator routines in the target language. The main benefit of the model-compilation approach is performance: individual calls to *ChooseSuccState*$_T$ do not need to

process the internal representation of the source model and thus execute faster.

State Compaction

The memory used by an analysis algorithm that stores states is obviously affected by how an individual state is represented and stored. For the simulation algorithm of figure 3.13, 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 that are sufficient to encode all possible values. A Boolean variable needs just one bit, a variable ranging over an enumerated type with three values needs two bits, and for a variable storing speed of a car, in miles per hour up to one decimal point, 10 bits should suffice. The state then is encoded as a sequence of bits.

3.3 Enumerative Search *

Given a transition system T and a property, to check whether the property is an invariant of the transition system, we can check whether the negation of the property is reachable. 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 by listing the state variables, their initialization, and the update code for the transitions and may not be finite. Hence, 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. To check whether a property is an invariant, it suffices to examine only the reachable subgraph.

Let us revisit the controller `Controller2` for the railroad crossing example of figure 3.8. For the system `RailRoadSystem2`, there are six state variables: $mode_W$ and $mode_E$, each of which ranges over $\{\texttt{away}, \texttt{wait}, \texttt{bridge}\}$; *west* and *east*, each of which ranges over $\{\texttt{green}, \texttt{red}\}$; and Boolean variables $near_W$ and $near_E$. As a result, the system has 144 possible states. It turns out that few of these states are reachable. Figure 3.15 shows the *reachable* portion of the graph. Each state is denoted by listing the values of the variables $mode_W$, $mode_E$, $near_W$, $near_E$, *west*, and *east*, in that order. We use a, w, b, g, and r as abbreviations for the values `away`, `wait`, `bridge`, `green`, and `red`, respectively. The initial state then

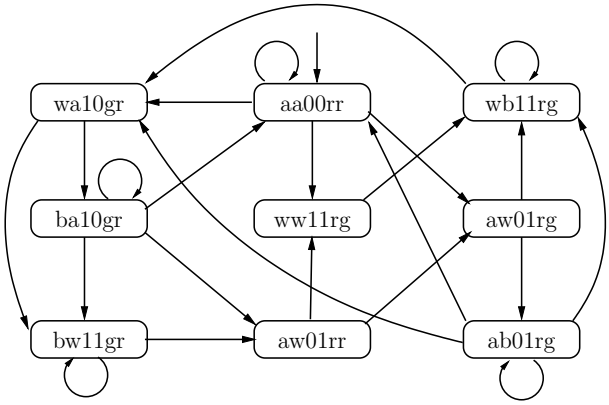


Figure 3.15: Reachable Subgraph of `RailRoadSystem2`

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, both, or neither of the two trains arriving. One of the transitions is a self-loop to $aa00rr$, and the other three lead to states $wa10gr$, $aw01rg$, and $ww11rg$. We then systematically consider all the possible transitions from these three new states and continue until no new states are discovered. Only 9 out of 144 states are found to be reachable.

As another example, consider the synchronous leader election of section 2.4.3. Suppose we choose values for the set P of node identifiers 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 $\{1, 2, \ldots N\}$, where N is the upper bound on the number of nodes in the network, 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 the case of 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 given a state s, there is a way to systematically enumerate all the successors of s, that is, states t such

that (s, t) is a transition. As in the case of simulation-based exploration, states of the transition system are of type `state`. We will use the following functions that access the representation of transition systems:

- Given a transition system T, *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, *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, *FirstSuccState*(s, T) returns the first successor state of s, according to the chosen enumeration of the set of states t such that (s, t) is a transition of T and the dummy state `null` if s has no outgoing transitions.
- Given a transition system T and states s and t, *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 no such state exists.

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 that can be enumerated as $10, 11, 12, \ldots$. 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. 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. If T has a state variable x of type `real` and the initialization specifies that $0 \leq x \leq 1$, then there are uncountably many initial states, one for every real number in the interval $[0, 1]$. This transition system is not countably branching. For such a system, it would not be possible to apply a systematic search exploring states one by one. Note that the simulation-based exploration is still possible in such a case, as one can implement *ChooseInitState* to return a randomly chosen real number up to a given precision of the floating-point representation of numbers.

The classical depth-first-search algorithm for on-the-fly exploration of (countably branching) transition systems is depicted in figure 3.16. It relies on the following data structures:

- The variable *Reach*, of type `set(state)`, stores the set of reachable states. The operations used on this set data structure are: (1) an initialization constant `EmptySet` that corresponds to the empty set; (2) a membership test `Contains` that takes a set and a state and returns 1 if the input state belongs to the input set and 0 otherwise; and (3) an insertion procedure `Insert` that takes a state and a set and updates the input set by adding the input state to it.

```
Input: A transition system T and property φ;
Output: If φ is reachable in T, return a witness, else return 0;

set(state) Reach := EmptySet;
stack(state) Pending := EmptyStack;
state s := FirstInitState(T);

while s ≠ null do {
  if Contains(Reach, s) = 0 then
    if DFS(s) = 1 then return Reverse(Pending);
  s := NextInitState(s, T);
  };
return 0.

bool function DFS(state s)
  Insert(s, Reach);
  Push(s, Pending);
  if Satisfies(s, φ) = 1 then return 1;
  state t := FirstSuccState(s, T);
  while t ≠ null do {
    if Contains(Reach, t) = 0 then
      if DFS(t) = 1 then return 1;
    t := NextSuccState(s, t, T);
    };
  Pop(Pending);
  return 0.
```

Figure 3.16: On-the-fly Depth-first Search Algorithm for Reachability

- 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: (1) an initialization constant `EmptyStack` that corresponds to the empty stack; (2) a procedure `Push` that takes a state and a stack and updates the input stack by adding the input state at its top; (3) a procedure `Pop` that takes a stack and updates it by removing the top element, if any; and (4) the function `Reverse` that takes a stack and returns the sequence of states it contains from bottom to top.

The algorithm maintains the set *Reach* of 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 satisfies the property φ, then the algorithm has discovered φ to be reachable, and all pending invocations of *DFS* terminate. The stack contains an execution of the transition system that demonstrates reachability of a state satisfying the property φ, and thus the reversed stack can be output as a witness execution. If the input state s does not satisfy the property φ, 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, then the algorithm has visited all the reachable states without encountering a state satisfying the property φ, and it returns 0.

Illustrative Example

Figure 3.17 shows a possible execution of the depth-first search algorithm on the transition system corresponding to the component `RailRoadSystem2` (see figure 3.15 for the reachable graph).

Initially, the algorithm invokes *DFS* with input state $aa00rr$ with *Reach* equal to the empty set and *Pending* equal to the empty stack. In figure 3.17, each state is assigned a unique number that indicates the order in which states are discovered and added to *Reach*. Thus, the state $aa00rr$ has number 1. The ordering of the successor states from a given state is depicted left to right in figure 3.17. That is, the first successor of $aa00rr$ is $aa00rr$. Since this state is already visited, no new call to *DFS* is initiated. Then the next successor of $aa00rr$, the state $wa10gr$, is examined. At this point, *Reach* contains only the state $aa00rr$, the stack also contains just this state, and the function *DFS* is invoked again with input state $wa10gr$, whose DFS discovery number is 2. Figure 3.17 shows the values of the set *Reach* of visited states and the stack *Pending* when a new call to *DFS* is made (for brevity, states are represented by their DFS discovery numbers). Note that when the call *DFS*($wa10gr$) returns, the set *Reach* contains all the states; as a result, examining the remaining two successors of $aa00rr$, states $aw01rg$ and $ww11rg$ in that order, do not initiate new calls to *DFS*.

When *DFS*($aw01rr$) first examines the successor $aw01rg$, it calls *DFS*($aw01rg$). This call explores all states that can be reached from $aw01rg$ and then returns. The value of *Reach* contains all these states. Since the next successor of $aw01rr$ is $ww11rg$ and not yet visited, *DFS*($ww11rg$) gets called. For two calls such as *DFS*($aw01rg$) and *DFS*($ww11rg$), which share the immediate parent calling context (*DFS*($aw01rr$) in this case), at the time the call, the value of the stack *Pending* is the same (and captures an execution leading to the input state for the call), but the value of the variable *Reach* has changed (the one on the right is guaranteed to be a superset of the one on the left).

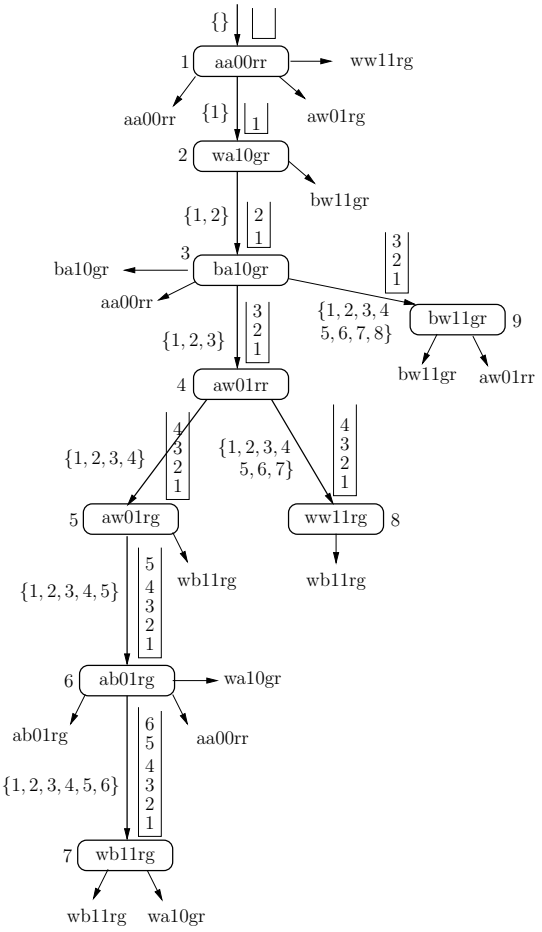


Figure 3.17: Sample Execution of Depth-first Search on `RailRoadSystem2`

Analysis of the Depth-First Search

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 to the number of reachable states and transitions. Even when the number of reachable states is not finite, the algorithm may encounter a state satisfying φ and terminate with a witness execution. However, if the property φ is not reachable and the number of reachable states is not finite, then the algorithm will just keep on examining more and more states until it runs out of available memory. These properties are summarized in the following theorem:

Theorem 3.2 [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 3.16 has the following guarantees:*

1. *If the algorithm returns 0, then the property φ is not reachable in T.*
2. *If the algorithm returns a sequence of states, then its output is a witness execution demonstrating the reachability of the property φ.*
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 the 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.

Bit-State 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, whose each entry is a list of states. Initially, all lists are empty. 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 jth entry of the hash-table. To check whether 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 jth entry is scanned to check whether it contains s. If the hashing function is chosen appropriately, then the number of states mapped to the same index is small; 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 jth entry is a single bit. Initially all bits are 0. The insertion of a state s, which is mapped to an integer j by the hash function, is implemented by setting the jth bit of the hash-table to 1. The membership test for a state s returns the value of the bit stored at the index corresponding to s. This scheme does not handle hash collisions correctly. Suppose that two states s and t are mapped to the same index j, and state s is inserted in the hash-table first. When the state t is encountered, as the jth bit of the hash-table is already set to 1, the membership test `Contains` incorrectly

returns a positive answer for the state t. 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. Thus, when the algorithm returns 0, we cannot be sure that the property φ is not reachable, but when it returns a sequence of states, it is guaranteed to be a witness to the reachability of φ. What fraction of the reachable states is explored 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: to insert a state, the corresponding bits in both the hash-tables are set to 1, and the membership test checks whether the bits corresponding to the input state in both the hash-tables are 1.

Exercise 3.10: Consider the reactive component `Switch` of figure 2.2. How many reachable states does it have? Draw the reachable subgraph for the corresponding transition system. ■

Exercise 3.11: 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 3.16 by removing the set *Reach*, and when a state is encountered, there is no test to check whether 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 3.2 continue to hold. How is the termination (claim (3) of theorem 3.2) affected? ■

Exercise 3.12*: 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. Such a representation 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 reachable in T, and state precisely its correctness and termination guarantees. ■

3.4 Symbolic Search

As the invariant-verification problem is computationally hard, we cannot hope to find an efficient scalable 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 explicitly processing 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 $20 \leq x \leq 99$ concisely represents the set $\{20, 21, \ldots, 99\}$ 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.

3.4.1 Symbolic Transition Systems

For symbolic analysis, the initial states and transitions can be described by formulas, that is, Boolean-valued expressions, over state variables that capture constraints on initialization and update.

Initialization and Transition Formulas

Consider a transition system T with state variables S. The initialization can be expressed by a Boolean expression φ_I over the variables S. The set of initial states then contains the states satisfying this expression. Let us revisit the transition system $\texttt{GCD}(m, n)$ (see figure 3.1) corresponding to the program that computes the `gcd` function. It has three state variables x, y, and *mode*. The initialization formula is $x = m \wedge y = n \wedge \mathit{mode} = \texttt{loop}$.

To express the transition description *Trans* as a formula, we need to relate the values of state variables *before* the transition to the values of the state variables *after* the transition. For a variable v, we use v' to denote a *primed* version of v with the same type as v, and this primed copy is used to capture the value after the transition. With this convention, the transition description is expressed by a Boolean expression φ_T over the state variables S and the corresponding set S' of primed state variables.

Let us develop the transition formula for the GCD program step by step. The extended-state machine of figure 3.1 has two mode-switches, and a transition corresponds to executing one of these switches. As a result, the transition formula is a disjunction of two formulas, with one disjunct each contributed by each mode-switch. Let us first focus on the self-loop.

Consider the assignment $x := x - y$. This translates to the logical formula $x' = x - y$ that relates the updated value of x to the old values of x and y. The assumption that "a state variable that is not updated explicitly stays unchanged" needs to be captured by explicit constraints in the logical formula. Thus, the formula corresponding to the decrement of x by y, with y implicitly unchanged, is $(x' = x - y) \wedge (y' = y)$. The conditional statement "`if` $(x > y)$ `then` $x := x - y$" is captured by the formula

$$(x > y) \wedge (x' = x - y) \wedge (y' = y).$$

Such a formula is evaluated over a pair of states of the system, where the first state is used to obtain values for the variables x and y, and the second state is used to obtain values for the variables x' and y'. A pair of states (s, t) satisfies this transition formula precisely when $s(x) > s(y)$ and $t(x) = s(x) - s(y)$ and $t(y) = s(y)$. Notice that if we simply omit the conjunct $y' = y$, the formula

would have the unintended meaning that the variable y can be updated to any arbitrary value when the condition $x > y$ holds.

The if-then-else statement "`if` $(x > y)$ `then` $x := x - y$ `else` $y := y - x$" corresponds to the formula ψ that is the disjunction of the two cases:

$$[(x > y) \wedge (x' = x - y) \wedge (y' = y)] \vee [\neg(x > y) \wedge (y' = y - x) \wedge (x' = x)].$$

The above pattern for mapping conditional statements to transition formulas is typical: the formula is a disjunction of cases, where each case is a conjunction of the condition and the update corresponding to this condition.

For the self-loop of the GCD program, the guard condition is $(x > 0 \wedge y > 0)$, and this mode-switch is from the mode `loop` to itself. Thus, the contribution of this self-loop to the transition formula is the formula ψ_1:

$$[(x > 0) \wedge (y > 0) \wedge (mode = \texttt{loop}) \wedge \psi \wedge (mode' = \texttt{loop})].$$

The contribution of the mode-switch from `loop` to `stop` can be computed in a similar manner. In particular, the update code "`if` $(x = 0)$ `then` $x := y$" translates to the formula ψ':

$$[(x = 0) \wedge (x' = y) \wedge (y' = y)] \vee [\neg(x = 0) \wedge (x' = x) \wedge (y' = y)],$$

and the contribution of this mode-switch to the transition formula is the formula ψ_2:

$$[\neg (x > 0 \wedge y > 0) \wedge (mode = \texttt{loop}) \wedge \psi' \wedge (mode' = \texttt{stop})].$$

The desired transition formula φ_T of the transition system $\texttt{GCD}(m, n)$ is: $\psi_1 \vee \psi_2$.

This style of specification of the transition relation is called *declarative*. Unlike the more familiar *operational* style that prescribes the sequence of statements to be executed, the declarative specifications only capture the constraints on the relationship between the old and new values of variables. Unlike an assignment, which is an operational and executable description, equality, which is a declarative and logical specification, does not have a meaningful distinction between the left-hand side and right-hand side. In particular, the expressions $x' = x - y$ and $x - y = x'$ are logically equivalent and express exactly the same constraint. Similarly, since logical conjunction is commutative, the formula $(x' = x - y) \wedge (y' = y)$ expresses exactly the same constraint as $(y' = y) \wedge (x' = x - y)$.

Reaction Formulas

For transition systems described by synchronous reactive components, the symbolic description of the transition system can be obtained by the analogous symbolic description of the corresponding reactive component. For a synchronous reactive component, the initialization can be captured by a formula φ_I over the state variables, and its reactions can be specified by a formula φ_R over (unprimed) state variables (denoting the state at the beginning of a round), input

and output variables, and primed state variables (denoting the state at the end of a round).

Let us revisit our first reactive component, `Delay`, of figure 2.1. The initialization formula φ_I for the component `Delay` is $x = 0$. Note that if we want to specify that the initial value of x may be either 0 or 1, the corresponding initial assignment $x :=$ `choose` $\{0, 1\}$ is captured by the formula 1 (every state satisfies the constant 1). Indeed, the constraint-based or declarative style can be more convenient to specify nondeterminism.

For the `Delay` component, the reaction formula φ_R is

$$(out = x) \ \wedge \ (x' = in)$$

that captures the relationship among old state, input, output, and updated state. In general, the reaction formula φ_R for a component C with state variables S, input variables I, and output variables O is a formula over the variables $S \cup I \cup O \cup S'$. For states s, t, input i, and output o of C, $s \xrightarrow{i/o} t$ is a reaction of C precisely when the formula φ_R is satisfied when we use state s to assign values to variables in S, input i to assign values to variables in I, output o to assign values to variables in O, and state t to assign values to the corresponding primed variables in S'.

Given a symbolic description of a synchronous reactive component C, the symbolic description of the corresponding transition system T can be obtained easily. The initialization formula for T is the same as the initialization formula for C. Recall that, for states s and t, (s, t) is a transition of T precisely when there *exists some* input i and output o such that $s \xrightarrow{i/o} t$ is a reaction of C. This relationship between transitions of T and reactions of C can be naturally expressed using the operation of *existential quantification* for logical formulas.

If f is a Boolean formula over a set V of variables and x is a variable in V, then $\exists\, x.\, f$ is a Boolean formula over $V \setminus \{x\}$. A valuation q over $V \setminus \{x\}$ satisfies the quantified formula $\exists\, x.\, f$ if q can be extended by assigning some value to x to satisfy f, that is, there exists a valuation s over V such that $s(f) = 1$ and $s(y) = q(y)$ for each variable y in $V \setminus \{x\}$. For example, if x and y are Boolean variables, then the formula $\exists\, x\, .(x \wedge y)$ expresses a constraint only over the variable y and is equivalent to the formula y (that is, $\exists\, x\, .(x \wedge y)$ evaluates to 1 exactly when y is assigned 1). Similarly, the formula $\exists\, x\, .(x \vee y)$ is equivalent to 1 (that is, this formula evaluates to 1 independent of the value of y).

The transition formula for the transition system corresponding to `Delay` is

$$\exists\, in.\, \exists\, out.\, [\,(out = x) \ \wedge \ (x' = in\,]).$$

This formula simplifies to the logical constant 1 since the formula is satisfied no matter what the values of x and x' are. This corresponds to the fact that for this transition system, there is a transition between every pair of states.

More generally, if φ_R is the reaction formula for the component C, then the transition formula φ_T for the corresponding transition system is obtained by existentially quantifying all the input and output variables: $\exists I. \exists O. \varphi_R$.

As another example, consider the component `TriggeredCopy` of figure 2.5. The only state variable is x of type `nat`, the initialization formula is $x = 0$, and the reaction formula is

$$(in? \wedge out = in \wedge x' = x + 1) \ \vee \ (\neg in? \wedge out = \bot \wedge x' = x).$$

The assumption that when the input event is absent, the component leaves the state unchanged and the output is absent, is captured explicitly in the reaction formula as a separate case. The corresponding transition formula is obtained by existentially quantifying the input and output variable:

$$\exists in, out. \ [\,(in? \wedge out = in \wedge x' = x + 1) \ \vee \ (\neg in? \wedge out = \bot \wedge x' = x)\,].$$

This transition formula can be simplified to a logically equivalent formula

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

Composing Symbolic Representations

In section 2.3, we studied how complex components can be combined using the operations of input/output variable renaming, parallel composition, and output hiding. The symbolic description of the resulting component can be obtained naturally from the symbolic descriptions of the original components. We will use the block diagram for `DoubleDelay` from figure 2.15 to illustrate this.

Renaming of variables is useful to create instances of components so that there are no name conflicts among state variables of different components, and common names for input/output variables indicate input/output connections. Given the initialization and reaction formulas for the original component, the corresponding formulas for the instantiated component can be obtained by textual substitution. For instance, the component `Delay1` is obtained from the component `Delay` by renaming the state variable x to x_1 and the output *out* to *temp*. The initialization formula for `Delay1` then is $x_1 = 0$, and the reaction formula is $(temp = x_1) \wedge (x_1' = in)$. Similarly, the initialization formula for `Delay2` is $x_2 = 0$, and the reaction formula is $(out = x_2) \wedge (x_2' = temp)$.

Consider two compatible components C_1 and C_2. If φ_I^1 and φ_I^2 are the respective initialization formulas for C_1 and C_2, then the initialization formula for the product $C_1 \| C_2$ is simply the conjunction $\varphi_I^1 \wedge \varphi_I^2$. This captures the fact that the formula φ_I^1 constrains the initial values of the state variables of C_1, and the formula φ_I^2 constrains the initial values of the state variables of C_2. Similarly, if φ_R^1 and φ_R^2 are the respective reaction formulas for C_1 and C_2, then the reaction formula for the product $C_1 \| C_2$ is the conjunction $\varphi_R^1 \wedge \varphi_R^2$. This again captures the intuition that in synchronous composition, state s of the composite,

on input i, can react with output o updating the state to t if the values assigned by s, i, o, and t are consistent with the reaction descriptions of the two original components.

In our running example, the composition of `Delay1` and `Delay2` gives the component with state variables $\{x_1, x_2\}$, output variables $\{temp, out\}$, input variables $\{in\}$, initialization formula $x_1 = 0 \wedge x_2 = 0$, and reaction formula

$$temp = x_1 \ \wedge \ x_1' = in \ \wedge \ out = x_2 \ \wedge \ x_2' = temp.$$

If y is an output variable of a component C, then we can use hiding to ensure that y is no longer an output that is observable outside. The initialization formula of the resulting component $C \setminus y$ is the same as the initialization formula for C, and if φ_R is the reaction formula for C, then the reaction formula for the result is $\exists y.\, \varphi_R$. In our example, if we hide the intermediate output *temp* of `Delay1` $\|$ `Delay2`, then we get the desired composite component `DoubleDelay`. Its initialization formula is

$$x_1 = 0 \ \wedge \ x_2 = 0,$$

and the reaction formula is

$$\exists\, temp\,.\ (\ temp = x_1 \ \wedge \ x_1' = in \ \wedge \ out = x_2 \ \wedge \ x_2' = temp\ ),$$

which can be simplified to an equivalent formula

$$x_1' = in \ \wedge \ out = x_2 \ \wedge \ x_2' = x_1.$$

To obtain the transition formula for the transition system corresponding to the component `DoubleDelay`, we can existentially quantify the variables *in* and *out* from the above formula. The resulting transition formula is equivalent to $x_2' = x_1$.

Exercise 3.13: Consider the transition system `Mult`(m, n) described in exercise 3.1. Describe this transition system symbolically using initialization and transition formulas. ■

Exercise 3.14: Consider the description of the component `Switch` given as an extended-state machine in figure 2.2. Give the initialization and reaction formulas corresponding to `Switch`. Obtain the transition formula for the corresponding transition system in as simplified form as possible. ■

Exercise 3.15*: Let C_1 and C_2 be two compatible reactive components with reaction formulas are φ_R^1 and φ_R^2, respectively. We have argued that the reaction formula for the product $C_1 \| C_2$ is $\varphi_R^1 \wedge \varphi_R^2$. Let φ_T^1 and φ_T^2 be the transition formulas for the transition systems corresponding to C_1 and C_2, respectively. Can we conclude that the transition formula for the transition system corresponding to the product $C_1 \| C_2$ is $\varphi_T^1 \wedge \varphi_T^2$? Justify your answer. ■

3.4.2 Symbolic Breadth-First Search

Before developing the symbolic algorithm for invariant verification, 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 over S, and the transitions are represented by a region over $S \cup S'$. We have already considered a specific instance of such a representation, namely, Boolean formulas over the variables. While representation using formulas is useful for understanding the symbolic algorithm, the algorithm works for any choice as long as the primitives discussed below are implemented.

The data type `reg` for regions supports the following operations:

- Given regions A and B, $\texttt{Disj}(A, B)$ returns the region that contains those states that are in either A or B.
- Given regions A and B, $\texttt{Conj}(A, B)$ returns the region that contains those states that are in both regions A and B.
- Given regions A and B, $\texttt{Diff}(A, B)$ returns the region that contains those states that are in A but not in B.
- Given a region A, $\texttt{IsEmpty}(A)$ returns 1 if the region A contains no states and 0 otherwise.
- Given a region A over V and a set $X \subseteq V$ of variables, $\texttt{Exists}(A, X)$ returns the region A projected onto over the variables $V \setminus X$. The result contains a valuation s over $V \setminus X$ precisely when there exists some valuation t over X such that the valuation over V obtained by combining s and t is in A.
- Given a region A over variables V, a list of variables $X = \{x_1, \ldots x_n\}$ in V and a list of variables $Y = \{y_1, \ldots y_n\}$ not in V, such that each variable y_j is of the same type as the corresponding variable x_j, $\texttt{Rename}(A, X, Y)$ returns the region obtained by renaming each variable x_j to y_j. Thus, the result contains a valuation t over the variables $(V \cup Y) \setminus X$ exactly when there exists some valuation s in A such that $t(y_j) = s(x_j)$, for $j = 1, \ldots n$, and $t(z) = s(z)$ for all variables z in $V \setminus X$.

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 the operations 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 $\texttt{Conj}(A, \mathit{Trans})$ 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 $\texttt{Post}(A)$.

Symbolic Image Computation

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

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

is a region over S that contains precisely those states t for which there is a transition (s, t) for some state s in A.

Examples of Image Computation

As an example, suppose the system has a single variable x of type `real`, and the transition region is given by the formula $x' = 2x+1$ (this corresponds to the assignment $x := 2x+1$). Consider the region A given by the formula $0 \leq x \leq 10$. 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 $1 \leq x \leq 21$. Verify that the constraint $1 \leq x \leq 21$ indeed describes all possible values of x after executing the assignment $x := 2x + 1$, given that $0 \leq x \leq 10$ describes all possible values of x before executing the assignment.

As a second example, consider a transition system with two integer variables x and y. Suppose the transitions of the system correspond to executing the statement:

$$\texttt{if}\ (y > 0)\ \texttt{then}\ x := x + 1\ \texttt{else}\ y := y - 1.$$

This statement translates to the following transition formula:

$$[\,(y > 0) \wedge (x' = x + 1) \wedge (y' = y)\,] \;\vee\; [\,(y \leq 0) \wedge (x' = x) \wedge (y' = y - 1)\,].$$

Consider a region A of this transition system described by the formula $2 \leq x - y \leq 5$. To compute the post-image of this region, we take its conjunction with the transition formula, and this gives:

$$\begin{array}{ll} & [\,(y > 0) \wedge (x' = x + 1) \wedge (y' = y) \wedge (2 \leq x - y \leq 5)\,] \\ \vee & [\,(y \leq 0) \wedge (x' = x) \wedge (y' = y - 1) \wedge (2 \leq x - y \leq 5)\,]. \end{array}$$

If we existentially quantify the variable x from this formula, then we get

$$\begin{array}{ll} & [\,(y > 0) \wedge (y' = y) \wedge (2 \leq x' - 1 - y \leq 5)\,] \\ \vee & [\,(y \leq 0) \wedge (y' = y - 1) \wedge (2 \leq x' - y \leq 5)\,]. \end{array}$$

Now let us existentially quantify the variable y from this formula, and we get

$$[\,(y' > 0) \wedge (2 \leq x' - 1 - y' \leq 5)\,] \;\vee\; [\,(y' + 1 \leq 0) \wedge (2 \leq x' - y' - 1 \leq 5)\,].$$

By renaming x' and y' to x and y, respectively, and some simplification, we obtain

$$(3 \leq x - y \leq 6) \;\wedge\; (y \neq 0)$$

as the post-image of the region A.

Consider the synchronous reactive component `3BitCounter` of section 2.4.1 (see figure 2.27). The component has two input variables, *inc* and *start*, and three output variables, out_0, out_1, and out_2. Let the state variables of the three latches corresponding to the three bits be x_0, x_1, and x_2. The reaction formula φ_R for the component is equivalent to:

$$out_0 = x_0 \;\wedge\; out_1 = x_1 \;\wedge\; out_2 = x_2 \;\wedge$$

$$\left[\begin{array}{l} (start = 1 \;\wedge\; x_0' = 0 \;\wedge\; x_1' = 0 \;\wedge\; x_2' = 0) \vee \\ (start = 0 \;\wedge\; inc = 0 \;\wedge\; x_0' = x_0 \;\wedge\; x_1' = x_1 \;\wedge\; x_2' = x_2) \vee \\ (start = 0 \;\wedge\; inc = 1 \;\wedge\; x_0' = \neg x_0 \;\wedge\; x_1' = x_0 \oplus x_1 \wedge\; x_2' = (x_0 \wedge x_1) \oplus x_2) \end{array}\right]$$

This formula expresses the constraint that each output bit out_j is the same as the (old) state x_j, and either (1) the input *start* is high, and all the updated state bits are 0, or (2) both the inputs are low, and the state stays unchanged, or (3) the input *start* is low and *inc* is high, and counter increments. The increment condition for the counter means that the low-order bit out_0 flips, the new value of middle bit out_1 is the exclusive-or (denoted by the operator $\oplus$) of the two low-order bits, and the new value of the high-order bit out_2 is the exclusive-or of the old value of out_2 and the conjunction of the other two bits. For the corresponding transition system, the transition formula φ_T is

$$\exists\, inc, start, out_0, out_1, out_2 \,.\, \varphi_R$$

which simplifies to

$$\begin{array}{l} (x_0' = 0 \;\wedge\; x_1' = 0 \;\wedge\; x_2' = 0) \vee \\ (x_0' = x_0 \;\wedge\; x_1' = x_1 \;\wedge\; x_2' = x_2) \vee \\ (x_0' = \neg x_0 \;\wedge\; x_1' = x_0 \oplus x_1 \;\wedge\; x_2' = (x_0 \wedge x_1) \oplus x_2) \end{array}$$

Now consider the region A given by the formula $x_0 = 0 \wedge x_1 = 0$; that is, the region A consists of states in which the two lower order bits are 0, but the higher order bit may be 0 or 1. To compute the image of this region, we first conjoin it with the transition formula φ_T, and this simplifies to

$$(x_0' = 0 \;\wedge\; x_1' = 0 \;\wedge\; x_2' = 0) \;\vee\; (x_1' = 0 \;\wedge\; x_2' = x_2)$$

Input: A transition system T given by regions *Init* for initial states and *Trans* for transitions, and a region φ for the property.
Output: If φ is reachable in T, return 1, else return 0.

```
reg Reach := Init;
reg New := Init;
while IsEmpty(New) = 0 do {
  if IsEmpty(Conj(New, φ)) = 0 then return 1;
  New := Diff(Post(New, Trans), Reach);
  Reach := Disj(Reach, New);
  };
return 0.
```

Figure 3.18: Symbolic Breadth-first Search Algorithm for Reachability

Existentially quantifying the old state variable x_2 and simplifying the result give us $x_1' = 0$. Finally, renaming the primed variables to unprimed ones gives the region $x_1 = 0$. This correctly captures the effect of executing one round of the three-bit counter: the set of successors of the region $\{000, 100\}$ is $\{000, 001, 100, 101\}$.

Iterative Image Computation

Now we are ready to describe the symbolic breadth-first search algorithm. The algorithm of figure 3.18 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* satisfies the property φ, that is, the intersection of the regions *New* and φ is non-empty, then the algorithm stops. If the region *New* is empty, then the algorithm can terminate reporting that no reachable state satisfies the property φ (that is, the negated property $\neg\varphi$ is an 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 3.19.

To illustrate how the symbolic breadth-first search algorithm works, let us revisit the transition system corresponding to the component `RailRoadSystem2` (see figure 3.15). Initially, $Reach = New = Init = \{aa00rr\}$. After one iteration of

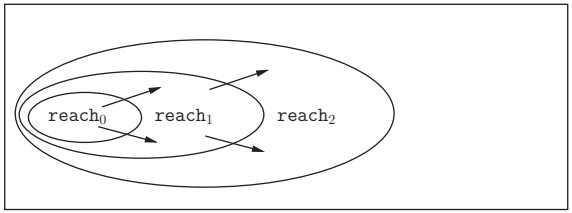


Figure 3.19: Symbolic Breadth-First Computation of Reachable States

the loop,

$$\begin{aligned} \mathit{Reach} &= \{\, aa00rr,\, wa10gr,\, ww11rg,\, aw01rg \,\}; \\ \mathit{New} &= \{\, wa10gr,\, ww11rg,\, aw01rg \,\}. \end{aligned}$$

After the second iteration of the loop,

$$\mathit{New} = \{\, ba10gr,\, bw11gr,\, ab01rg,\, wb11rg \,\}.$$

After the third iteration of the loop, *Reach* contains all the nine reachable states, and *New* equals $\{\, aw01rr \,\}$. After the fourth iteration, *New* becomes empty, and *Reach* stays unchanged. As a result, the algorithm stops.

If the symbolic breadth-first search algorithm terminates, then its answer is correct. If the property φ is reachable and the shortest witness contains j transitions, then after j iterations of the while-loop, the algorithm terminates. If the property is not reachable, then the algorithm can terminate only if it discovers all the reachable states after a finite number of iterations. This is guaranteed if the number of reachable states of T is finite.

Theorem 3.3 [Symbolic Breadth-first Search for Reachability] *Given a symbolic representation of a transition system T and a property φ, the symbolic breadth-first search algorithm of figure 3.18 has the following guarantees:*

1. *If the algorithm terminates, then the returned value correctly indicates whether the property φ is reachable in T.*

2. *If the property φ is reachable in the transition system T, then the algorithm terminates after j iterations of the while-loop, where j is the length of the shortest witness to the reachability of φ.*

3. *If there exists a natural number j, such that every reachable state of T is reachable by an execution with at most j transitions, then the algorithm terminates after at most j iterations of the while-loop.*

■

A natural choice for a symbolic representation of regions is formulas. In particular, for transition systems with only Boolean variables, we can use formulas over Boolean variables 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 $\texttt{Exists}(A, x)$, where x is a Boolean variable, corresponds to $A_0 \vee A_1$, where the formula A_0 is obtained from A by replacing each occurrence of the variable x with the constant 0, and the formula A_1 is obtained from A by replacing each occurrence of x with 1. All these operations individually can be implemented efficiently over formulas. However, the check $\texttt{IsEmpty}(A)$ corresponds to checking satisfiability of the formula A, that is, whether there *exists* a 0/1 assignment to Boolean variables so that the formula A evaluates to 1. This test is computationally expensive. When A has Boolean variables and logical connectives of negation, conjunction, and disjunction but no existential quantification, this is exactly the canonical NP-complete problem known as *propositional satisfiability* or `SAT`. Furthermore, in the context of the iterative breadth-first search of our symbolic algorithm of figure 3.18, the main drawback of representing the regions *Reach* and *New* as formulas is that there is no guaranteed way to simplify the formula representing the reachable states at each step. The formulas will get more and more complex due to the operations applied in each iteration of the while-loop, and their size grows exponentially with the number of iterations. The data structure of ordered binary decision diagrams, to be discussed in the next section, offers a possible remedy.

Exercise 3.16: Consider the symbolic image computation for a transition system with two real-valued variables x and y and transition description given by the formula $x' = x + 1 \wedge y' = x$. Suppose the region A is described by the formula $0 \leq x \leq 4 \wedge y \leq 7$. Compute the formula describing the post-image of A. ■

Exercise 3.17: Consider a transition system T with two integer variables x and y. The transitions of the system correspond to executing the statement:

$$\texttt{if } (x < y) \texttt{ then } x := x + y \texttt{ else } y := y + 1.$$

Write the transition formula over the variables x, y, x', and y' that captures the transition relation of the system. Consider a region A of the above transition system described by the formula $0 \leq x \leq 5$. Compute the formula describing the post-image of A. ■

Exercise 3.18*: The symbolic breadth-first search algorithm of figure 3.18 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` using the symbolic operations such as `Conj`, `Rename`, and `Exists` so that given a region A, $\texttt{Pre}(A, \mathit{Trans})$ is the region over S that contains precisely those states s for which there is a transition (s, 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`. ■

Exercise 3.19 *: Suppose we want to modify the symbolic breadth-first search algorithm of figure 3.18 so that when it finds the property φ to be reachable, it outputs a witness execution. Which additional operations on regions will be needed for this purpose? Using these operations, modify the algorithm so that it outputs a witness execution. ■

3.4.3 Reduced Ordered Binary Decision Diagrams *

Reduced ordered binary decision diagrams (ROBDDs) provide a compact and canonical representation for formulas over Boolean variables (or, equivalently, for Boolean functions).

Ordered Binary Decision Diagrams

Let V be a set containing k Boolean variables. A Boolean formula f over V represents a function from $\texttt{bool}^k$ to `bool`. For a variable x in V, the following equivalence, called the *Shannon expansion* of f around the variable x, holds:

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

Here, the formulas $f[x \mapsto 0]$ and $f[x \mapsto 1]$ are obtained from f by substituting the variable x by the constants 0 and 1, respectively. The formulas $f[x \mapsto 0]$ and $f[x \mapsto 1]$ do not refer to the variable x and are thus Boolean functions with domain $\texttt{bool}^{k-1}$. As a result, the Shannon expansion can be used to recursively simplify a Boolean function. This suggests representing Boolean functions as decision diagrams.

A (binary) decision diagram is a directed acyclic graph with two types of vertices: *terminal* and *internal*. The terminal vertices have no outgoing edges, and are labeled with one of the Boolean constants, 0 or 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 $f(u)$. Given a valuation q for all the variables in V, the value of the Boolean function $f(u)$ is obtained by traversing a path starting from u as follows. Consider an internal vertex v labeled with x. If $q(x)$ is 0, then we choose the left successor of v; if $q(x)$ is 1, then we choose the right successor of v. If the path terminates in a terminal vertex labeled with 0, then the function $f(u)$ evaluates to 0 according to the assignment q; if the path terminates in a terminal vertex labeled with 1, then the function $f(u)$ evaluates to 1 according to q.

Ordered (binary) decision diagrams (OBDDs) are decision diagrams in which we choose a linear order $<$ over the variables V and require that the labels of internal vertices appear in an order that is consistent with $<$. 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 $<$. The semantics of OBDDs is defined by associating Boolean formulas with each of the vertices. The definition is formalized below.

ORDERED BINARY DECISION DIAGRAM

Let V be a finite set of Boolean variables and $<$ be a total order over V. An *ordered binary decision diagram* B over $(V, <)$ 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 $\mathit{left}(u)$, such that if $\mathit{left}(u)$ is an internal vertex, then $\mathit{label}(u) < \mathit{label}(\mathit{left}(u))$; and
5. *Right edges:* a right-child function *right* that maps each internal vertex u to a vertex $\mathit{right}(u)$, such that if $\mathit{right}(u)$ is an internal vertex, then $\mathit{label}(u) < \mathit{label}(\mathit{right}(u))$.

For such an OBDD, each vertex u has an associated Boolean formula over V: $f(u)$ equals $\mathit{label}(u)$ if u is a terminal vertex, and equals

$$[\,\neg\, \mathit{label}(u) \;\wedge\; f(\mathit{left}(u))\,] \;\vee\; [\,\mathit{label}(u) \;\wedge\; f(\mathit{right}(u))\,]$$

otherwise. The Boolean formula $f(B)$ associated with the OBDD B is the formula $f(u^0)$ associated with the root u^0.

A Boolean constant is represented by an OBDD that contains a single terminal vertex labeled with that constant. Figure 3.20 shows one possible OBDD for the formula $(x \wedge y) \vee (x' \wedge y')$ with the ordering $x < y < x' < y'$. A left edge is shown as an arrow ending with an empty circle, and a right edge is shown as an arrow ending with a filled circle. The OBDD of figure 3.20 is, in fact, a tree. Figure 3.21 shows a more compact OBDD for the same formula with the same ordering of variables.

Isomorphism and Equivalence

Two OBDDs B and C are *isomorphic* if the corresponding labeled graphs are isomorphic. Two OBDDs B and C are *equivalent* if the Boolean formulas $f(B)$

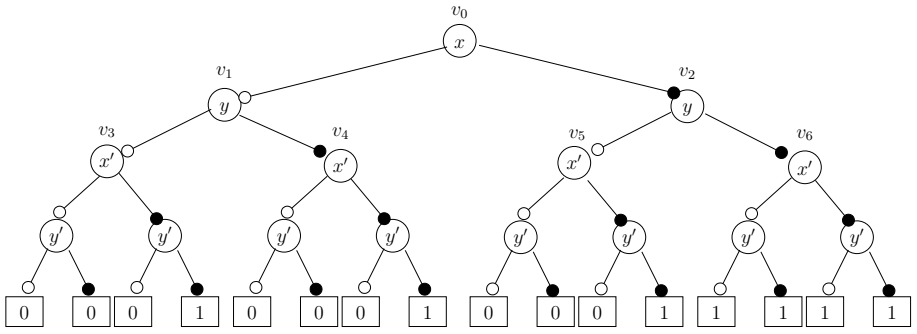


Figure 3.20: Ordered Binary Tree for $(x \wedge y) \vee (x' \wedge y')$

and $f(C)$ are equivalent. Thus, isomorphism means that the two OBDDs are structurally the same, and equivalence means that the two OBDDs are semantically the same. Clearly, isomorphic OBDDs represent the same formulas and are, thus, equivalent. The converse need not hold: the ordered binary decision diagrams of figures 3.20 and 3.21 are not isomorphic but are equivalent.

The notions of equivalence and isomorphism also extend to individual vertices. If B is an OBDD over $(V, <)$ and u is a vertex of B, then the subgraph rooted at u consisting of the vertices and edges that are reachable from u is also an OBDD over $(V, <)$. In figure 3.20, the subgraph rooted at vertex v_3 is an OBDD that represents the Boolean formula $x' \wedge y'$. Two vertices u and v of the OBDD 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. Since OBDDs are acyclic graphs, the notion of isomorphism can be defined by the following rules: (1) two terminal vertices u and v are isomorphic if $\mathit{label}(u) = \mathit{label}(v)$, and (2) two internal vertices u and v are isomorphic if $\mathit{label}(u) = \mathit{label}(v)$ and the left successors $\mathit{left}(u)$ and $\mathit{left}(v)$ are isomorphic and the right successors $\mathit{right}(u)$ and $\mathit{right}(v)$ are isomorphic. In figure 3.20, all terminal vertices with label 0 are isomorphic to one another. The subgraphs rooted at vertices v_3, v_4, and v_5 are isomorphic. In contrast, the vertices v_5 and v_6 are not isomorphic to each other.

Reduced Ordered Binary Decision Diagrams

A *reduced OBDD* (ROBDD) is obtained from an OBDD by repeatedly applying the following two steps:

1. Merge isomorphic vertices into one.
2. Eliminate internal vertices with identical left and right children.

Each step reduces the number of vertices while preserving equivalence. For instance, consider the OBDD of figure 3.20. Since vertices v_3 and v_4 are iso-

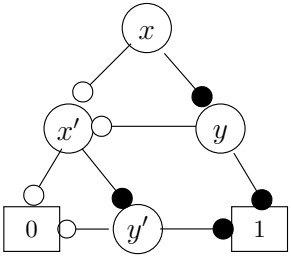


Figure 3.21: Reduced Ordered Binary Decision Diagram for $(x \wedge y) \vee (x' \wedge y')$

morphic, we can delete one of them, say v_4, along with the subtree rooted at 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 ROBDD of figure 3.21. It turns out that the above transformations are sufficient to obtain a *canonical* form: the final ROBDD is the same irrespective of the specific order in which the reductions are applied, and if we start with two non-isomorphic but equivalent OBDDs, we get the same final ROBDD.

Reduced Ordered Binary Decision Diagram

A ROBDD over a totally ordered set $(V, <)$ of Boolean variables is an ordered binary decision diagram $B = (U, u^0, \mathit{label}, \mathit{left}, \mathit{right})$ over $(V, <)$ such that:

1. there are no two distinct vertices u and v in U with $\mathit{label}(u) = \mathit{label}(v)$ and $\mathit{left}(u) = \mathit{left}(v)$ and $\mathit{right}(u) = \mathit{right}(v)$, and
2. for every internal vertex u, the two children $\mathit{left}(u)$ and $\mathit{right}(u)$ are distinct vertices.

Direct Construction of ROBDDs: An Example

Let us consider the Boolean formula

$$\varphi_0 : \ (x \vee \neg y) \ \wedge \ (y \vee z)$$

and suppose the variable ordering is $x < y < z$. Instead of first building an OBDD and then reducing it using the two reduction rules, let us try to build a ROBDD directly. The first step is illustrated in figure 3.22(a), which shows the root vertex v_0 labeled with the variable x. Note that the root vertex must be labeled with x because x is the first variable in the chosen ordering, and the value of the formula φ_0 depends on the value of x. The left successor of the vertex v_0 should be a vertex, say v_1, that represents the formula $\varphi_1 = \varphi_0[x \mapsto 0]$, which simplifies to the formula $\neg y \wedge z$, and the right successor of the vertex v_0

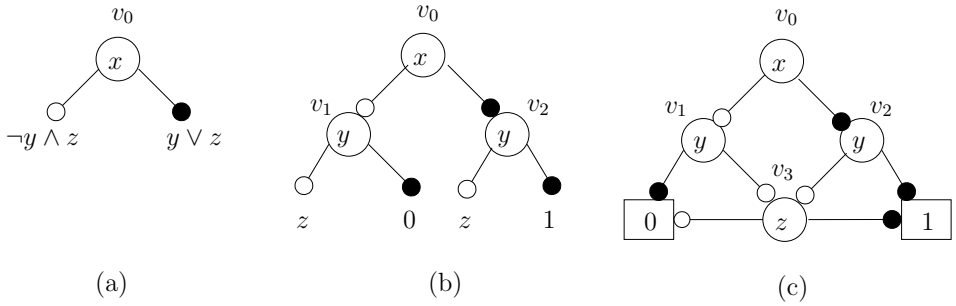


Figure 3.22: Building ROBDD for $(x \vee \neg y) \wedge (y \vee z)$

should be a vertex, say v_2, that represents the formula $\varphi_2 = \varphi_0[x \mapsto 1]$, which simplifies to the formula $y \vee z$.

The vertex v_1 corresponds to the formula $\varphi_1 : \neg y \wedge z$, and since the variable y is the next variable in the chosen ordering, the vertex v_1 is labeled with y. Its left successor should be a vertex, say v_3, that represents the formula $\varphi_1[y \mapsto 0]$, which is equivalent to the formula z, and the right successor of the vertex v_1 should be a vertex that represents the formula $\varphi_1[y \mapsto 1]$, which is equivalent to the constant 0, and hence must be a terminal vertex.

The ROBDD corresponding to the vertex v_2 that represents the formula $\varphi_2 : y \vee z$ is developed using a similar logic (see figure 3.22(b)). The vertex v_2 is labeled with the variable y. Its left successor should be a vertex that represents the formula $\varphi_2[y \mapsto 0]$, which is equivalent to the formula z. Since the vertex v_3 already represents the formula z and the reduction rules require as much sharing as possible, the left child of the vertex v_2 must be v_3. The right successor of the vertex v_2 should be a vertex that represents the formula $\varphi_2[y \mapsto 1]$, which is equivalent to the constant 1 and, hence, must be a terminal vertex.

Finally, the vertex v_3 corresponding to the formula z has label z, the terminal vertex 0 as its left successor, and the terminal vertex 1 as its right successor. This completes the desired ROBDD for the formula φ_0 as shown in figure 3.22(c).

Properties of ROBDDs

The next theorem asserts the basic facts about representing Boolean functions using ROBDDs. Every Boolean function has a unique, up to isomorphism, representation as a ROBDD. Furthermore, the ROBDD of a Boolean function has the least number of vertices among all OBDDs for the same function once we fix the variable ordering.

Theorem 3.4 [Existence, Uniqueness, and Minimality of ROBDDs] *Let V be a set of variables and $<$ be a total order over V.*

1. *If f is a Boolean formula over V, then there exists a ROBDD B over $(V, <)$ such that $f(B)$ and f are equivalent.*
2. *If B and C be two ROBDDs over $(V, <)$, then they are equivalent if and only if they are isomorphic.*
3. *If B is a ROBDD over $(V, <)$ and C is an OBDD over $(V, <)$, such that the two are equivalent, then C contains at least as many vertices as B.*

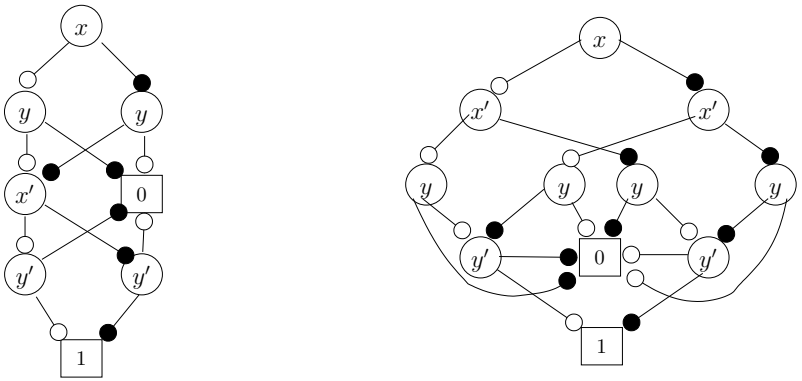
■

Checking equivalence of two ROBDDs 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 a ROBDD with a single terminal vertex labeled with 0, and the Boolean constant 1 is represented by a ROBDD with a single terminal vertex labeled with 1. A Boolean formula represented by a ROBDD B is satisfiable if and only if the root of B is not a terminal vertex labeled with 0. A Boolean formula represented by a ROBDD B is valid if and only if the root of B is a terminal vertex labeled with 1. Thus, checking satisfiability or validity of Boolean formulas is particularly easy if we use ROBDD representation.

The size of the ROBDD representation of a Boolean formula may be exponential in the number of variables. The size of the ROBDD representing a given formula depends on the choice of the ordering of variables. Consider the formula $(x = y) \wedge (x' = y')$. Figure 3.23 shows two ROBDDs for two different orderings of the variables. This example illustrates that the ordering can influence the size dramatically: one ordering may result in a ROBDD whose size is linear in the number of variables, whereas another ordering may result in a ROBDD whose size is exponential in the number of variables. While choosing an optimal ordering of variables can lead to exponential saving, computing the optimal ordering is a computationally hard problem.

There are Boolean functions whose ROBDD representation does not depend on the chosen ordering, and the ROBDD 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, whereas that 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, the ROBDD for the parity function contains $2k + 1$ vertices.
- **Multiplication.** Consider the set of variables $\{x_0, \ldots, x_{k-1}, y_0, \ldots, y_{k-1}\}$, and for $0 \leq j < 2k$, let $Mult_j$ denote the Boolean function that denotes the jth bit of the product of the two k-bit inputs, one encoded by the x-bits and another encoded by the y-bits. For every ordering $<$ of the variables, the total number of vertices in the ROBDDs representing all the functions



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

Figure 3.23: Two ROBDDs for $(x = y) \wedge (x' = y')$

$Mult_j$ is guaranteed to grow exponentially with k. More precisely, the following *lower bound* result has been established: there exists an index $0 \leq j < 2k$ such that the ROBDD for $Mult_j$ has at least $2^{k/8}$ vertices.

Shared Data Structure for ROBDDs

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

Let V be an ordered set of k Boolean variables. The type of ROBDDs is `bdd`, which is either a Boolean constant (denoting a terminal vertex) or a pointer to an entry in the global data structure *BDDPool*. The type of *BDDPool* is `set(bddnode)`, and it stores the (internal) vertices of ROBDDs. An internal vertex records a variable label, which ranges over the type $\texttt{nat}[1, k]$ containing the numbers $\{1, \ldots k\}$, and a left and a right pointer, each of which has type `bdd`. Thus, the vertices of ROBDDs have type `bddnode`, which equals $\texttt{nat}[1, k] \times \texttt{bdd} \times \texttt{bdd}$. The type `bddnode` supports the following operations:

- The operation $\texttt{Label}(u)$, for an internal vertex u, returns the first component of u, which is the number of the variable labeling u.
- The operation $\texttt{Left}(u)$, for an internal vertex u, returns the second component of u, which is either a Boolean constant or a pointer to the left

```
function AddVertex
Input: Variable label j in nat[1, k], ROBDDs B_0, B_1 of type bdd.
Output: ROBDD B such that f(B) is equivalent to (¬ x_j ∧ f(B_0)) ∨
    (x_j ∧ f(B_1)).

if B_0 = B_1 then return B_0;
if Contains((j, B_0, B_1), BDDPool) = 0 then
  Insert((j, B_0, B_1), BDDPool);
return Index((j, B_0, B_1)).
```

Figure 3.24: Creating ROBDD Vertices

successor of u.

- The operation `Right`(u), for an internal vertex u, returns the third component of u, which is either a Boolean constant or a pointer to the right successor of u.

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

- For an internal vertex u in *BDDPool*, `Index`(u) returns a pointer to u.
- For a pointer B, the operation *BDDPool*$[B]$ returns the vertex that B points to.

For such a representation, given a pointer B of type `bdd`, we write $f(B)$ to denote the Boolean function associated with the ROBDD that B points to. To avoid duplication of isomorphic nodes while manipulating ROBDDs, it is necessary that new vertices are created using the function *AddVertex* of figure 3.24. If no two vertices in the global set *BDDPool* are isomorphic before an invocation of the function *AddVertex*, then even after the invocation, no two vertices in *BDDPool* are isomorphic.

As an illustrative example, let us examine the snapshot of the global data structure *BDDPool* shown in figure 3.25. Each row shows an internal vertex stored in this data structure. For example, the ROBDD B_0 points to a vertex labeled with variable x_4 whose left successor is the terminal vertex 0 and right successor is the terminal vertex 1, and the ROBDD B_4 points to a vertex labeled with variable x_1 whose left successor is the terminal vertex 0 and right successor is the ROBDD B_2. The Boolean functions corresponding to each of the vertices are listed below:

$$\begin{aligned} f(B_0) &= x_4, \\ f(B_1) &= x_2 \vee x_4, \\ f(B_2) &= \neg x_3 \vee x_4, \\ f(B_3) &= x_1 \wedge (x_2 \vee x_4), \end{aligned}$$

Index	Label	Left	Right
B_0	4	0	1
B_1	2	B_0	1
B_2	3	1	B_0
B_3	1	0	B_1
B_4	1	0	B_2

Figure 3.25: Illustrative Snapshot of the Data Structure *BDDPool*

$$f(B_4) \quad = \quad x_1 \ \wedge \ (\neg\, x_3 \ \vee \ x_4).$$

Observe that the ROBDDs B_3 and B_4 share the ROBDD B_0.

Operations on ROBDDs

To construct a ROBDD representation of a given Boolean formula and implement the primitives of the symbolic reachability algorithm, we need a way to compute conjunctions and disjunctions of ROBDDs. We give a recursive algorithm for obtaining conjunction of ROBDDs. The algorithm is shown in figure 3.26.

Consider two ROBDDs, B and B', and suppose we wish to compute the conjunction $f(B) \wedge f(B')$. If one of them is a Boolean constant, then the result can be determined immediately. For instance, if B is the terminal constant 0, then the conjunction is also the terminal constant 0. If B is the terminal constant 1, then the conjunction is equivalent to $f(B')$, and thus the algorithm can return B'. Also, note that when both the ROBDDs are the same, we can use the fact $f \wedge f$ always equals f, and thus the result coincides with the input argument.

The interesting case is when both ROBDDs are pointers to distinct internal vertices, say u and u', respectively. Let j be the minimum of the indices labeling u and u'. Then x_j is the least variable that the function $f(u) \wedge f(u')$ can depend on. The label of the root of the conjunction is j, the left successor is the ROBDD for $(f(u) \wedge f(u'))[x_j \mapsto 0]$, and the right successor is the ROBDD for $(f(u) \wedge f(u'))[x_j \mapsto 1]$. Let us consider the left successor. Observe the equivalence

$$(f(u) \wedge f(u'))[x_j \mapsto 0] \ \equiv \ f(u)[x_j \mapsto 0] \ \wedge \ f(u')[x_j \mapsto 0].$$

If u is labeled with j, the ROBDD for $f(u)[x_j \mapsto 0]$ is the left successor of u. If the label of u exceeds j, then the function $f(u)$ does not depend on x_j, and the ROBDD for $f(u)[x_j \mapsto 0]$ is u itself. The ROBDD for $f(u')[x_j \mapsto 0]$ is computed similarly, and then the function *Conj* is applied recursively to compute the conjunction according to the expression above.

Let us apply this scheme to compute the conjunction of the ROBDDs B_3 and B_4 from figure 3.25. The vertex corresponding to B_3 has label x_1, left successor 0, and right successor B_1, while the vertex corresponding to B_4 has label x_1, left successor 0, and right successor B_2. As a result,

$$\mathit{Conj}(B_3, B_4) \;=\; \mathit{AddVertex}(1, \mathit{Conj}(0, 0), \mathit{Conj}(B_1, B_2)).$$

The call $\mathit{Conj}(0, 0)$ returns 0 using the rule for the constant ROBDDs. To compute the conjunction of B_1 and B_2, the algorithm examines the corresponding vertices: the vertex corresponding to B_1 has label x_2, left successor B_0, and right successor 1, while the vertex corresponding to B_2 has a higher label x_3. This leads to:

$$\mathit{Conj}(B_1, B_2) \;=\; \mathit{AddVertex}(2, \mathit{Conj}(B_0, B_2), \mathit{Conj}(1, B_2)).$$

This generates two recursive calls to *Conj* again: the second call $\mathit{Conj}(1, B_2)$ returns immediately with the answer B_2 using rules for simplification when one of the arguments is a constant. The first call $\mathit{Conj}(B_0, B_2)$ requires examination of the corresponding vertices: the vertex corresponding to B_0 has label x_4, while the vertex corresponding to B_2 has label x_3, left successor 1, and right successor B_0. This leads to:

$$\mathit{Conj}(B_0, B_2) \;=\; \mathit{AddVertex}(3, \mathit{Conj}(B_0, 1), \mathit{Conj}(B_0, B_0)).$$

In this case, both the recursive calls to `Conj` return immediately: $\mathit{Conj}(B_0, 1)$ returns B_0 and $\mathit{Conj}(B_0, B_0)$ also returns B_0 using the rule for the conjunction of identical ROBDDs. As a result, a call is made to $\mathit{AddVertex}(3, B_0, B_0)$. This does not create a new vertex since the reduction rules do not allow both left and right successors to be the same. The call $\mathit{AddVertex}(3, B_0, B_0)$ simply returns B_0:

$$\mathit{Conj}(B_0, B_2) \;=\; \mathit{AddVertex}(3, B_0, B_0) \;=\; B_0.$$

Now, $\mathit{Conj}(B_1, B_2)$ calls $\mathit{AddVertex}(2, B_0, B_2)$. The data structure *BDDPool* does not contain a vertex with label 2, left successor B_0, and right successor B_2. As a result, *AddVertex* will create a new entry $(2, B_0, B_2)$, with the index B_5:

$$\mathit{Conj}(B_1, B_2) \;=\; \mathit{AddVertex}(2, B_0, B_2) \;=\; B_5.$$

Finally, $\mathit{Conj}(B_3, B_4)$ calls $\mathit{AddVertex}(1, 0, B_5)$. Again, *BDDPool* does not contain such a vertex, so a new entry, indexed by B_6, is created and is the desired result, namely, the ROBDD representation of the conjunction of the functions represented by B_3 and B_4:

$$\mathit{Conj}(B_3, B_4) \;=\; \mathit{AddVertex}(1, 0, B_5) \;=\; B_6.$$

Avoiding Recomputation

The recursive algorithm described so far may call the function *Conj* repeatedly with the same two arguments. To avoid unnecessary computation, a table is

```
Input: bdd B, B'.
Output: bdd B'' such that f(B'') is equivalent to f(B) ∧ f(B').

table[(bdd × bdd) × bdd] Done = EmptyTable

return Conj(B, B').

bdd Conj(bdd B, B')
  bddnode u, u'; bdd B'', B_0, B_1, B'_0, B'_1; nat[1, k] j, j'

  if ( B = 0 ∨ B' = 1 ) then return B;
  if ( B = 1 ∨ B' = 0 ) then return B';
  if B = B' then return B;
  if Done[(B, B')] ≠⊥ then return Done[(B, B')];
  if Done[(B', B)] ≠⊥ then return Done[(B', B)];
  u := BDDPool[B]; u' := BDDPool[B'];
  j := Label(u); B_0 := Left(u); B_1 := Right(u);
  j' := Label(u'); B'_0 := Left(u'); B'_1 := Right(u');
  if j = j' then B'' := AddVertex(j, Conj(B_0, B'_0), Conj(B_1, B'_1));
  if j < j' then B'' := AddVertex (j,Conj(B_0,B'),Conj(B_1,B'));
  if j > j' then B'' := AddVertex (j',Conj(B,B'_0),Conj(B,B'_1));
  Done[(B, B')] := B'';
  return B''.
```

Figure 3.26: Algorithm for Taking Conjunction of ROBDDs

used to store the arguments and the corresponding result of each invocation of *Conj*. When *Conj* is invoked with input arguments B and B', it first consults the table to check whether the conjunction of $f(B)$ and $f(B')$ was previously computed. The actual recursive computation is performed only the first time, and the result is stored 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` × `value`]. This data type supports the retrieval and update operations like arrays: $D[k]$ is the value stored in the table D with the key k, and the assignment $D[k] := m$ updates the value stored in D for the key k. The constant table `EmptyTable` has the default value $\perp$ stored with every key. Tables can be implemented as arrays or hash-tables. The table used by the algorithm uses a pair of ROBDDs as a key and stores ROBDDs as values.

Let us analyze the time complexity of the algorithm of figure 3.26. Suppose the ROBDD pointed to by B has n vertices and the ROBDD pointed to by B' has n' 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 in 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 $O(n \cdot n')$.

Theorem 3.5 [ROBDD Conjunction] *Given two ROBDDs, B and B, the algorithm of figure 3.26 correctly computes the ROBDD for $f(B) \wedge f(B')$. If the ROBDD pointed to by B has n vertices and the ROBDD pointed to by B' has n' vertices, then the time complexity of the algorithm is $O(n \cdot n')$.* ■

Similar algorithms can be developed to implement other operations such as `Disj` (logical disjunction), `Diff` (set difference), and `Exists` (existential quantification).

Symbolic Search Using ROBDDs

We now have all the machinery to implement the symbolic search algorithm using ROBDDs as a representation for regions. We have already discussed how to construct symbolic representation of transition systems as initialization and transition formulas φ_I and φ_T. If all the variables in the source description are Boolean variables, then the formulas φ_I and φ_T are Boolean formulas built using logical connectives and existential quantification. We can build ROBDDs corresponding to these formulas using the operations we have discussed.

If the formula is an atomic formula of the form $x = 1$, then the corresponding ROBDD is obtained by calling the *AddVertex* function: if the position of the variable x in the variable ordering is j, then the desired ROBDD is simply $\mathit{AddVertex}(j, 1, 0)$. If the formula f is of the form $f_1 \wedge f_2$, then we first build ROBDDs B_1 and B_2 corresponding to the (simpler) formulas f_1 and f_2, respectively, and then the ROBDD for f is obtained by invoking $\mathit{Conj}(B_1, B_2)$. The ROBDD operations *Disj*, *Diff*, and *Exists* are used to process the corresponding operations of disjunction, negation, and existential quantification in the formulas. We can define the function *FormulaToBdd* that maps Boolean formulas to ROBDDs as follows:

$$
\begin{aligned}
\mathit{FormulaToBdd}(\mathrm{x}_j = 1) &= \mathit{AddVertex}(j, 0, 1) \\
\mathit{FormulaToBdd}(\mathrm{x}_j = 0) &= \mathit{AddVertex}(j, 1, 0) \\
\mathit{FormulaToBdd}(f_1 \wedge f_2) &= \mathit{Conj}(\mathit{FormulaToBdd}(f_1), \mathit{FormulaToBdd}(f_2)) \\
\mathit{FormulaToBdd}(f_1 \vee f_2) &= \mathit{Disj}(\mathit{FormulaToBdd}(f_1), \mathit{FormulaToBdd}(f_2)) \\
\mathit{FormulaToBdd}(\neg f) &= \mathit{Diff}(1, \mathit{FormulaToBdd}(f)) \\
\mathit{FormulaToBdd}(\exists X.\, f) &= \mathit{Exists}(\mathit{FormulaToBdd}(f), X)
\end{aligned}
$$

For symbolic invariant verification, given a property φ, we also need to build the ROBDD for the formula φ. Then we can use the algorithm of figure 3.18,

where every region is a pointer into the global data structure storing ROBDD vertices.

While ROBDDs can be used directly when all system variables are of type `bool`, they can also be used for analysis of finite-state systems with variables of enumerated or other finite types by encoding a finite type using a sequence of Boolean variables. For example, consider the state variable *mode* of the `Train` component that takes three possible values `away`, `wait`, and `bridge` (see figure 3.4). We can encode the variable *mode* using two Boolean variables $mode_0$ and $mode_1$, using values 00, 01, and 10 to encode the three possibilities for *mode*. Expressions of the form $mode = \texttt{away}$ are replaced by $mode_0 = 0 \wedge mode_1 = 0$; expressions of the form $mode = \texttt{wait}$ are replaced by $mode_0 = 0 \wedge mode_1 = 1$; and expressions of the form $mode = \texttt{bridge}$ are replaced by $mode_0 = 1 \wedge mode_1 = 0$.

The ROBDD representation of a Boolean formula can be exponential in the number of variables and is sensitive to the ordering of variables. Given a system with Boolean state variables S, to build the representation of the initialization and transition formulas, we need to choose an ordering $<$ of the variables in $S \cup S'$. Recall that one of the steps in the image computation is to rename all the primed variables to unprimed variables. This renaming step can be implemented by renaming the labels of the internal vertices of the ROBDD if the ordering of the primed variables is consistent with the ordering of the corresponding unprimed variables. This gives us our first rule for choosing $<$: for all variables $x, y \in S$, $x < y$ if and only if $x' < y'$. Another commonly used rule for choosing the ordering stipulates that a variable should appear only after all the variables it depends on. For example, when the update is specified using task graphs, if a task A writes a state variable x, then x' should appear after the variables read by the task A as well as the tasks preceding A according to the precedence constraints. Finally, the variables that are related to each other should be clustered together. In particular, instead of ordering all the primed variables after all the unprimed variables, we can try to minimize the distance between a primed variable and the unprimed variables it depends on.

The practical tools for analyzing systems using ROBDDs employ a wide array of techniques to combat the growth in the ROBDD size with the number of variables. As a result, symbolic invariant verification using ROBDDs has had significant success in analyzing industrial-scale hardware designs and embedded controllers. Yet it is not a *magic bullet*, and the performance of ROBDD-based tools remains unpredictable: sometimes they can reveal hitherto unknown bugs in complex designs, and sometimes the breadth-first search algorithm can finish only a few iterations before the number of ROBDD vertices created becomes too large compared to the available memory.

Exercise 3.20: Consider the Boolean formula

$$(x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z).$$

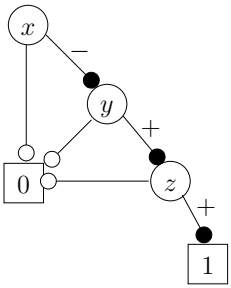


Figure 3.27: A Decision Graph with Complement Edges

Draw the ROBDD for this formula with respect to the variable ordering $x < y < z$. ■

Exercise 3.21: Consider the Boolean formula

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

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

Exercise 3.22: 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 ROBDD 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 $y_1 y_0$. ■

Exercise 3.23*: Give an algorithm for computing the existential quantification for ROBDDs: given a ROBDD B and a set X of variables, $\mathit{Exists}(B, X)$ should return the ROBDD for the formula $\exists X.\, f(B)$. ■

Exercise 3.24*: An ordered binary decision diagram with *complement edges* (COBDD) is similar to an ordered binary decision diagram B with an additional component that classifies each right edge as positive $+$ or negative $-$. The function $f(u)$ for an internal vertex u is redefined so that $f(u)$ equals $(\neg \mathit{label}(u) \wedge f(\mathit{left}(u))) \vee (\mathit{label}(u) \wedge f(\mathit{right}(u)))$ if the right edge of u is positive and $(\neg \mathit{label}(u) \wedge f(\mathit{left}(u))) \vee (\mathit{label}(u) \wedge \neg f(\mathit{right}(u)))$ if the right edge of u is negative. Thus, when the right edge is negative, we negate the function associated with the right successor. For instance, in figure 3.27, the vertex labeled with y represents the function $y \wedge z$ while the root represents the function $(x \wedge \neg(y \wedge z))$.

(1) Is there a function with a COBDD representation that is smaller than its ROBDD representation for the same variable ordering? (2) Can we define a notion of *reduced* ordered binary decision diagrams with complement edges (RCOBDD) as a subclass of COBDDs such that every boolean function has a unique representation? ■

Bibliographic Notes

The concept of invariants and inductive invariants was introduced in 1960s in early papers to formalize the notion of program correctness [Hoa69]. For an introduction to principles and tools for program verification, we refer the reader to [Lam02] and [BM07]. There have been prominent successes of software verification in industrial projects recently; see [BLR11, BBC+10, IBG+11, Hol13] as illustrative examples.

Efficient on-the-fly enumerative search for invariant verification was developed in the 1980s and forms the core analysis engine in the model checker SPIN [Hol04, Hol97] (see also the model checker MURPHI [Dil96]).

Bryant introduced ROBDDs as an efficient representation on Boolean functions [Bry86]. Symbolic search using ROBDDs was first introduced in the model checker SMV and was instrumental in the success of verification tools in analyzing hardware protocols [BCD+92, McM93] (see also the model checkers VIS and NUSMV and the associated optimized implementations of ROBDD operations [BHSV+96, CCGR00]).

Many of the illustrative examples in this chapter are borrowed from the draft textbook on *Computer Aided Verification* [AH99a].

We have briefly mentioned concepts in computability theory such as decidability and NP-completeness. For a comprehensive introduction to these topics, see [Sip13].