Contents

Chapter 2

Invariant Verification

In this chapter, we study the formulation and verification of the simplest but most important kind of system requirements, called invariants. An invariant classifies the states of a reactive module into safe and unsafe, and asserts that during the execution of the module, no unsafe state can be encountered.

2.1 Transition Graphs

The information about a module which is necessary for checking invariants is captured by the transition graph of the module. Consequently, invariant verification is performed on transition graphs.

2.1.1 Definition of Transition Graphs

At every point during the execution of a system, the information that is necessary to continue the execution is called the state of the system. The state of a discrete system changes in a sequence of update rounds. Every possible state change is called a transition of the system. The behaviors of a discrete system can thus be captured by a directed graph whose vertices represent the system states and whose edges represent the system transitions. Such a graph is called a transition graph.

> Transition graph
>
> A transition graph G consists of (1) a set Σ of vertices, (2) a subset $\sigma^I \subseteq \Sigma$ of the vertices, and (3) a binary edge relation $\rightarrow \subseteq \Sigma^2$ on the vertices. The vertices in Σ are called states, the vertices in σ^I are called initial states, and the edges in $\rightarrow$ are called transitions. We refer to the set Σ of states as the state space of G. Every subset of states from Σ is called a region; in particular, σ^I is the initial region of G. Every binary relation on Σ is called an action; in particular, $\rightarrow$ is the transition action of G.

Properties of transition graphs

The following properties of transition graphs are important. First, the transition action of every deadlock-free system is serial: for every state s, there is at least one successor state t with $s \rightarrow t$. Second, the mathematical analysis of a system is often simplified if the transition action is finitely branching: for every state s, there are at most finitely many successor states t with $s \rightarrow t$. Third, if the system may decide, in every update round, to leave the state unchanged, then the transition action is reflexive: every state s is its own successor state; that is, $s \rightarrow s$. Last, systems are amenable to analysis by graph algorithms if the state space is finite.

> Serial, finitely branching, reflexive, finite transition graph
>
> The transition graph $G = (\Sigma, \sigma^I, \rightarrow)$ is serial if (1) the initial region σ^I is nonempty and (2) the transition action $\rightarrow$ is serial. The transition graph G is finitely branching if (1) the initial region σ^I is finite and (2) the transition action $\rightarrow$ is finitely branching. The transition graph G is reflexive if the transition action $\rightarrow$ is reflexive. The transition graph G is finite if the state space Σ is finite.

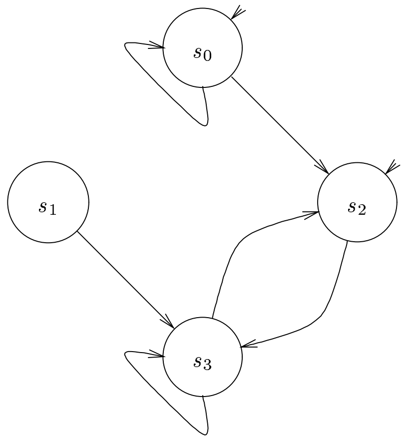

Figure 2.1: The transition graph $\hat{G}$

Remark 2.1 [Finite implies finite branching] Every finite transition graph is finitely branching. ■

Trajectories of transition graphs

The execution of a discrete system follows a path in the corresponding transition graph. Such a path starts in an initial state and proceeds through successive transitions. We are interested only in the states that are encountered within a finite number of transitions. The resulting finite paths are called initialized trajectories.

> Trajectory of transition graph
>
> Let $G = (\Sigma, \sigma^I, \rightarrow)$ be a transition graph. A trajectory of G is a nonempty word $\overline{s}_{1..m}$ over the alphabet Σ of states such that $s_i \rightarrow s_{i+1}$ for all $1 \leq i < m$. The first state s_1 is the source, the last state s_m is the sink, and the number m of states is the length of the trajectory $\overline{s}_{1..m}$. The trajectory $\overline{s}_{1..m}$ is an initialized trajectory of G if the source s_1 is an initial state of G. The set of initialized trajectories of G, denoted L_G, is called the state language of the transition graph G.

Remark 2.2 [Seriality implies trajectories of arbitrary length] Let G be a serial transition graph, and let s be a state of G. For every positive integer i, there is at least one trajectory of G with source s and length i. In particular, for every positive integer i, there is at least one initialized trajectory of G with length i. It follows that for serial transition graphs G, the state language L_G is infinite. ■

Example 2.1 [Transition graph] Figure 2.1 shows a finite transition graph with four states (s_0, s_1, s_2, and s_3). The two states s_0 and s_2 are initial, as is indicated by the short arrows without source state. The transition graph $\hat{G}$ has a total of six transitions. Since every state has at least one outgoing transition, $\hat{G}$ is serial. The infinite state language $L_{\hat{G}}$ includes the following four initialized trajectories:

s_0
$s_0 s_0 s_0 s_0$
$s_0 s_0 s_2 s_3$
$s_2 s_3 s_2 s_3 s_3 s_3 s_2$

The state language $L_{\hat{G}}$ is the regular set $s_0^+ \cup (s_0^* s_2 (s_3^+ s_2)^* s_3^*)$. ■

Remark 2.3 [Languages defined by transition graphs] For a transition graph G with state space Σ, the state language L_G is a language over the alphabet Σ. We say that G defines the language L_G. If G is finite, then L_G is a regular language. But not every subset of Σ^* is definable by a transition graph with state space Σ, and not every regular language is definable by a finite transition graph. This is shown in the following exercise. ■

Exercise 2.1 {T2} [Languages defined by transition graphs] Let A be an alphabet, and let $L \subseteq A^*$ be a language over A. (a) Prove that the language L is definable by a transition graph iff L is prefix-closed and fusion-closed. (Fusion closure captures the fact that the system state determines the possible future behaviors of the system.) (b) Prove that the language L is definable by a transition graph with the initial region A iff L is prefix-closed, fusion-closed, and suffix-closed. (c) Prove that the language L is definable by a reflexive transition graph iff L is prefix-closed, fusion-closed, and stutter-closed. ■

2.1.2 From Reactive Modules to Transition Graphs

We associate with every reactive module a serial transition graph that captures the behaviors of the module.

The states of a module

The state of a module between two rounds is determined by the values of all module variables. This is because the possible outcomes of the next and all future update rounds are determined solely by the current values of the module variables, and do not depend on any previous values.

> State space of a module
>
> Let P be a reactive module. A state of P is a valuation for the set X_P of module variables. We write Σ_P for the set of states of P.

Remark 2.4 [Existence of states] Every module has at least one state. The empty module, with the empty set of module variables, has exactly one state. ■

The state s of a module is initial if after the initial round, all module variables may have the values indicated by s. Consider a variable x. If x is external, then s can map x to any value of the appropriate type. If x is controlled by an atom U, then all variables in $\mathsf{await}X_U$ are initialized before x. In this case, the initial value $s(x)$ depends on the initial values of the awaited variables of U. The dependence is specified by the initial command init_U, which defines a relation between the valuations for the primed awaited variables $\mathsf{await}X'_U$ and the valuations for the primed controlled variables $\mathsf{ctr}X'_U$. In the following, if s is a valuation for a set X of unprimed variables, we write $prime(s)$ for the valuation for the set X' of corresponding primed variables such that $prime(s)(x') = s(x)$ for all variables $x \in X$.

Initial region of a module

Let P be a reactive module, let s be a state of P, and let $s' = prime(s)$. The state s is an initial state of P if for every atom U of P,

$$(s'[\mathsf{await}X'_U], s'[\mathsf{ctr}X'_U]) \in [\![\mathsf{init}_U]\!].$$

We write σ^I_P for the set of initial states of P.

Example 2.2 [Mutual exclusion] Recall Peterson's solution to the asynchronous mutual-exclusion problem from Figure 1.23. The module $Pete$ has $3\times2\times3\times2 = 36$ states. Four of the states —s_1, s_2, s_3, and s_4— are initial:

$$\begin{array}{l}
s_1(pc_1) = outC,\ s_1(x_1) = true,\ s_1(pc_2) = outC,\ s_1(x_2) = true;\\
s_2(pc_1) = outC,\ s_2(x_1) = true,\ s_2(pc_2) = outC,\ s_2(x_2) = false;\\
s_3(pc_1) = outC,\ s_3(x_1) = false,\ s_3(pc_2) = outC,\ s_3(x_2) = true;\\
s_4(pc_1) = outC,\ s_4(x_1) = false,\ s_4(pc_2) = outC,\ s_4(x_2) = false.
\end{array}$$

It follows that the initial value of pc_1 and pc_2 is $outC$, and the initial values of x_1 and x_2 are unspecified. ■

Lemma 2.1 [Existence of initial states] Every module has an initial state.

Proof. Consider a module P. We prove the stronger claim that for every valuation s_e for the external variables of P, there is an initial state s of P such that $s[\mathsf{extl}X_P] = s_e$. Consider a valuation s_e for $\mathsf{extl}X_P$ and an execution order $U_1, \ldots, U_n$ for the atoms of P. We construct a sequence $s_0, s_1, \ldots, s_n$ of valuations for X_P as follows (let $s'_i = prime(s_i)$ for all $0 \le i \le n$): first, choose s_0 so that $s_0[\mathsf{extl}X_P] = s_e$; then, for all $1 \le i \le n$ and $Y_i = X_P \backslash \mathsf{ctr}X_{U_i}$, let $s_i[Y_i] = s_{i-1}[Y_i]$ and choose $s_i[\mathsf{ctr}X_{U_i}]$ so that $(s'_{i-1}[\mathsf{await}X'_{U_i}], s'_i[\mathsf{ctr}X'_{U_i}]) \in$

$[\![\mathsf{init}_{U_i}]\!]$. At each step, at least one choice is possible because the binary relation $[\![\mathsf{init}_{U_i}]\!]$ is serial. The construction ensures that for all module variables x of P, if x is an external variable, then $s_n(x) = s_e(x)$, and if x is a controlled variable of the atom U_i, then $s_n(x) = s_i(x)$. It follows that s_n is an initial state of P with $s_n[\mathsf{extl}X_P] = s_e$. ■

The transitions of a module

Consider two states s and t of a module. If the state s indicates the current values of the module variables at the beginning of an update round, and the state t indicates possible next values of the module variables at the end of the update round, then the state pair (s,t) is a transition of the module. For a formal definition of transitions, consider a variable x. If x is external, then t can map x to any value of the appropriate type. If x is controlled by an atom U, then all variables in $\mathsf{await}X_U$ are updated before x. In this case, the next value $t(x)$ depends on the current values of the read variables of U and on the next values of the awaited variables. The dependence is specified by the update command update_U, which defines a relation between the valuations for the unprimed read variables $\mathsf{read}X_U$ and the primed awaited variables $\mathsf{await}X'_U$ on one hand, and the valuations for the primed controlled variables $\mathsf{ctr}X'_U$ on the other hand.

Transition action of a module

Let P be a reactive module, let s and t be two states of P, and let $t' = \mathit{prime}(t)$. The state pair (s,t) is a transition of P if for every atom U of P,

$$(s[\mathsf{read}X_U] \cup t'[\mathsf{await}X'_U], t'[\mathsf{ctr}X'_U]) \in [\![\mathsf{update}_U]\!].$$

We write $\rightarrow_P$ for the set of transitions of P.

Example 2.3 [Mutual exclusion] Consider the state s_1 of the module *Pete* from Example 2.2. There are four transitions —(s_1,s_1), (s_1,s_5), (s_1,s_6), and (s_1,s_7)— whose first component is the initial state s_1:

$$\begin{array}{l} s_5(pc_1) = reqC,\ s_5(x_1) = true,\ s_5(pc_2) = outC,\ s_5(x_2) = true; \\ s_6(pc_1) = outC,\ s_6(x_1) = true,\ s_6(pc_2) = reqC,\ s_6(x_2) = false; \\ s_7(pc_1) = reqC,\ s_7(x_1) = true,\ s_7(pc_2) = reqC,\ s_7(x_2) = false. \end{array}$$

The transition (s_1,s_1) corresponds to an update round in which both processes sleep; the transition (s_1,s_5) corresponds to an update round in which the first process proceeds and the second process sleeps; the transition (s_1,s_6) corresponds to an update round in which the first process sleeps and the second process proceeds; and the transition (s_1,s_7) corresponds to an update round in which both processes proceed. ■

Lemma 2.2 [Existence of transitions] Let P be a module. For every state s of P, there is a state t of P such that $s \rightarrow_P t$.

Exercise 2.2 {T1} [Proof of Lemma 2.2] Let P be a module. Prove that for every state s of P, and every valuation t_e for the external variables of P, there is a state t of P such that (1) $s \rightarrow_P t$ and (2) $t[\mathsf{extl}X_P] = t_e$. Lemma 2.2 follows. ■

The transition graph of a module

We can now collect together the state space, initial region, and transition action of a module, thus obtaining a transition graph.

> Transition graph of a module
>
> Given a reactive module P, the transition graph underlying P is $G_P = (\Sigma_P, \sigma_P^I, \rightarrow_P)$.

Terminology. From now on, we freely attribute derivatives of the transition graph G_P to the module P. For example, each trajectory of G_P is called a trajectory of P; the state language L_{G_P} is called the state language of P, and denoted L_P. ■

Example 2.4 [Mutual exclusion] Figure 2.2 shows the transition graph G_{Pete} for Peterson's mutual-exclusion protocol. The label $o1i0$ denotes the state s with $s(pc_1) = outC$, $s(x_1) = true$, $s(pc_2) = inC$, and $s(x_2) = false$, etc. Each state s has a reflexive transition of the form $s \rightarrow s$, and these transitions are omitted from the figure. Note that some states at the left and right borders of the figure are identical, so as to avoid a large number of crossing edges in the figure. It can be checked that the state sequence shown in Figure 1.24 is indeed an initialized trajectory of G_{Pete}. ■

Proposition 2.1 [Seriality of transition graphs that underlie modules] For every module P, the transition graph G_P is serial.

Proof. Proposition 2.1 follows from Lemmas 2.1 and 2.2. ■

Remark 2.5 [Transition graphs for finite, closed, deterministic, and passive modules] If P is a finite module, then G_P is a finite transition graph. If all external variables of P have finite types, then G_P is a finitely branching transition graph. In particular, for every closed module, the underlying transition graph is finitely branching, and for every closed deterministic module P, the underlying transition graph has branching degree 1: there is exactly one initial state, and for each state s, there is exactly one successor state t with $s \rightarrow_P t$. If P is a passive module, then the transition action $\rightarrow_P$ is reflexive. ■

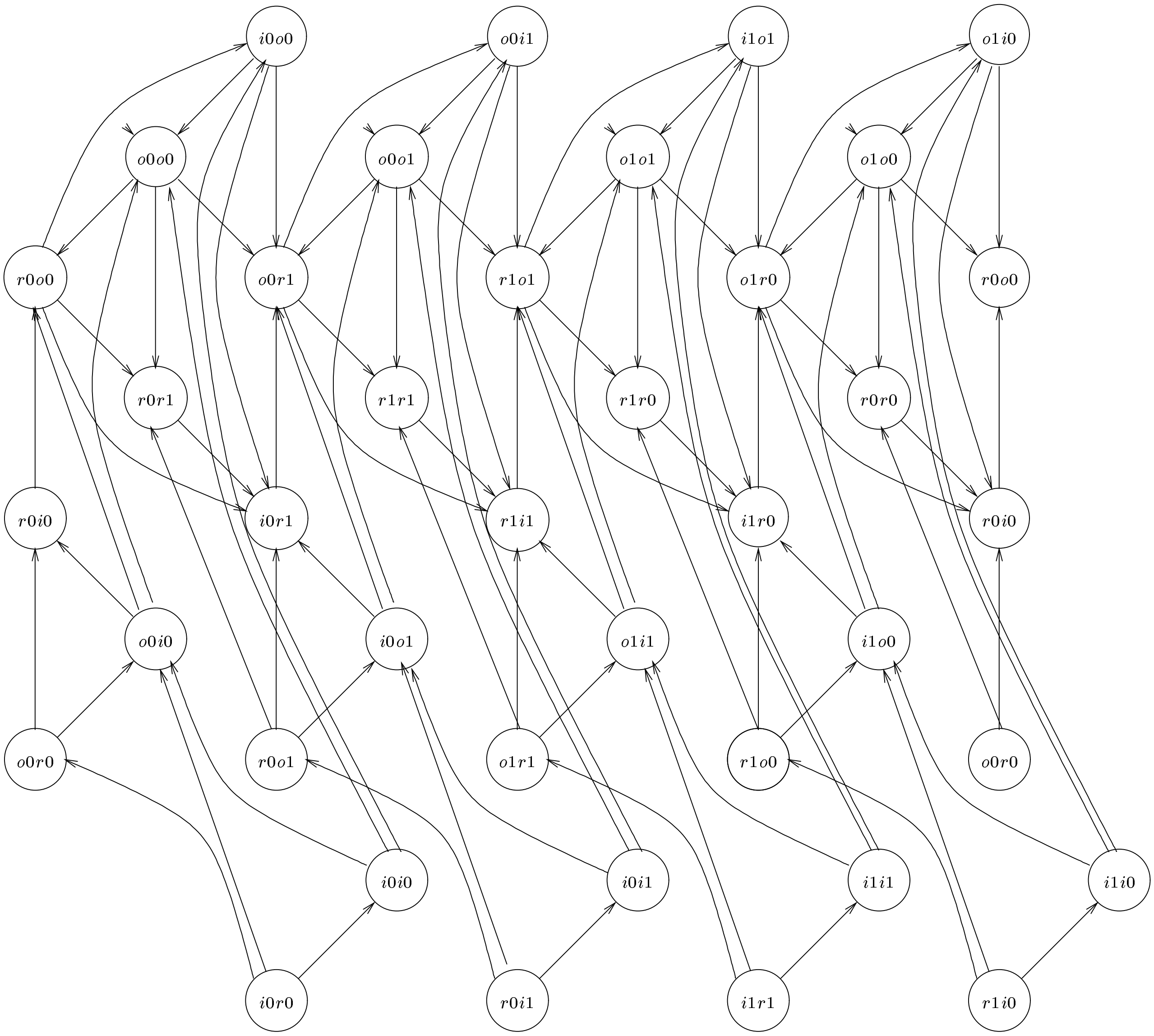

Figure 2.2: The transition graph G_{Pete}

The transition graph G_P captures only the behaviors of the module P, and not its interface structure. First, transition graphs do not distinguish between controlled and external variables. Hence there is no composition operation on transition graphs. Second, transition graphs do not distinguish between private and observable variables. Hence asynchronicity does not correspond to a property of transition graphs, and there is no hiding operation on transition graphs (the hiding of variables does not alter the transition graph of a module).

Exercise 2.3 {T2} [Transition graph of compound modules] Consider two compatible modules P and Q. (a) Assume that the two modules have no private variables ($\mathsf{priv}X_P = \mathsf{priv}X_Q = \emptyset$), and that the interface variables of one module are the external variables of the other module ($\mathsf{intf}X_P = \mathsf{extl}X_Q$ and $\mathsf{extl}X_P = \mathsf{intf}X_Q$). Then the two modules and the compound module have the same state space: $\Sigma_P = \Sigma_Q = \Sigma_{P\|Q}$. Prove that the transition action of the compound module $P\|Q$ is the intersection of the transition actions of the two component modules P and Q; that is, $s \rightarrow_{P\|Q} t$ iff $s \rightarrow_P t$ and $s \rightarrow_Q t$. What can you say about the initial states of the compound module $P\|Q$? (b) Assume that the two modules have no variables in common ($X_P \cap X_Q = \emptyset$). Then $\Sigma_{P\|Q} = \{s_1 \cup s_2 \mid s_1 \in \Sigma_P \text{ and } s_2 \in \Sigma_Q\}$. Prove that the transition action of the compound module is the cartesian product of the transition actions of the two component modules; that is, $(s_1 \cup s_2) \rightarrow_{P\|Q} (t_1 \cup t_2)$ iff $s_1 \rightarrow_P t_1$ and $s_2 \rightarrow_Q t_2$. What can you say about the initial states of $P\|Q$? (c) Now consider the general case. Consider two states s and t of the compound module $P\|Q$. Prove that (1) $s \in \sigma^I_{P\|Q} t$ iff $s[X_P] \in \sigma^I_P$ and $s[X_Q] \in \sigma^I_Q$, and (2) $s \rightarrow_{P\|Q} t$ iff $s[X_P] \rightarrow_P t[X_P]$ and $s[X_Q] \rightarrow_Q t[X_Q]$. ■

The following proposition asserts that the (initialized) trajectories of a compound module are determined by the (initialized) trajectories of the component modules. In particular, for two compatible modules P and Q, if the two modules have the same state space, then $L_{P\|Q} = L_P \cap L_Q$.

Proposition 2.2 [Trajectories of compound modules] For every pair P and Q of compatible modules, a sequence $\overline{s}$ of states in $\Sigma_{P\|Q}$ is an (initialized) trajectory of the compound module $P\|Q$ iff $\overline{s}[X_P]$ is an (initialized) trajectory of P and $\overline{s}[X_Q]$ is an (initialized) trajectory of Q.

Proof. Proposition 2.2 follows from part (c) of Exercise 2.3. ■

Exercise 2.4 {T1} [Least constraining environments] The module Q is a least constraining environment for the module P if (1) P and Q are compatible, (2) the compound module $P\|Q$ is closed, and (3) $G_{P\|Q} = G_P$. Prove that for every module P, if all external variables of P have finite types, then there exists a least constraining environment for P. Can a module have more than one least constraining environment? ■

An interpreter for reactive modules

Following the informal execution model of Chapter 1, we are now equipped to build an interpreter for reactive modules. The execution of a module for a finite number of rounds yields an initialized trajectory of the module. Therefore, if the interpreter receives as input the module P, it returns as output an initialized trajectory of P. Since P may have many initialized trajectories, the output of the interpreter is nondeterministic. Indeed, every initialized trajectory of P must be a possible output of the interpreter.

The interpreter, Algorithm 2.1, proceeds in three phases. The first phase computes an execution order for the atoms of P. The second phase simulates the initial round, by executing the initial commands of all atoms in the chosen execution order. The third phase simulates a finite number of update rounds, by iteratively executing the update commands of all atoms in the chosen execution order. Algorithm 2.1 uses the following notation. The function $\mathit{Execute}(\Gamma, s)$, shown below, computes the result of executing the guarded command Γ on the valuation s. If Γ is a guarded command from X to Y, then s must be a valuation for X, and the function *Execute* returns a valuation t for Y such that $(s, t) \in [\![\Gamma]\!]$:

function $\mathit{Execute}(\Gamma, s)$
  Assume Γ is a guarded command from X to Y;
  Choose a guarded assignment γ of Γ such that $s(p_\gamma) = \mathit{true}$;
  Let t be the valuation for the empty set of variables;
  foreach y in Y do $t := t[y \mapsto s(e^x_\gamma)]$ od;
  return t.

If s is a valuation for a set X' of primed variables, we write $\mathit{unprime}(s)$ for the valuation for the set X of corresponding unprimed variables such that $\mathit{unprime}(s)(x) = s(x')$ for all variables $x \in X$.

2.1.3 The Reachability Problem

For a transition graph G that captures the behaviors of a system, we are interested only in the states of G that occur on initialized trajectories. These states are called reachable. By deleting the unreachable states from G, we obtain the reachable subgraph of G. The reachable subgraph can often be significantly smaller than the complete transition graph.

Algorithm 2.1 [Module Execution] (schema)

Input: a reactive module P.
Output: an initialized trajectory $\overline{s}_{1..m}$ of P.

Preparation.
Topologically sort the atoms of P with respect to the precedence relation $\prec\!\!\prec_P$, and store the result as $(U_1, \ldots, U_n)$;

Initial round.
Choose an arbitrary valuation s for $\mathsf{extl}X'_P$;
for $j := 1$ to n do $s := s \cup \mathit{Execute}(\mathsf{init}_{U_j}, s)$ od;
$s_1 := \mathit{unprime}(s)$;

Update rounds.
Choose an arbitrary positive integer m;
for $i := 2$ to m do
Choose an arbitrary valuation s for $\mathsf{extl}X'_P$;
for $j := 1$ to n do $s := s \cup \mathit{Execute}(\mathsf{update}_{U_j}, s_{i-1} \cup s)$ od;
$s_i := \mathit{unprime}(s)$
od.

Reachability

Let $G = (\Sigma, \sigma^I, \rightarrow)$ be a transition graph, and let s be a state of G. The state s is reachable in i transitions, for a nonnegative integer i, if the transition graph G has an initialized trajectory with sink s and length $i + 1$. The state s is a reachable state of G if there is a nonnegative integer i such that s is reachable in i transitions. The transition graph G is finitely reaching if there is a nonnegative integer i such that every reachable state of G is reachable in at most i transitions. The reachable region of G is the set σ^R of reachable states of G. The reachable subgraph of G is the transition graph $G^R = (\sigma^R, \sigma^I, \rightarrow^R)$, where $\rightarrow^R = \rightarrow [\sigma^R]$ is the restriction of the transition action $\rightarrow$ to the reachable region σ^R. The transitions of G^R are called the reachable transitions of G.

Example 2.5 [Reachable subgraph] In the simple transition graph $\hat{G}$ from Example 2.1, the state s_1 is unreachable, and so is the transition from s_1 to s_3. The reachable subgraph of $\hat{G}$ is shown in Figure 2.3. ■

Example 2.6 [Mutual exclusion] Figure 2.4 shows the reachable subgraph of the transition graph G_{Pete} from Example 2.4 (reflexive transitions are suppressed). It has four initial states, 20 reachable states, and 64 reachable transitions. In other words, 16 of the states in Figure 2.2 are unreachable. ■

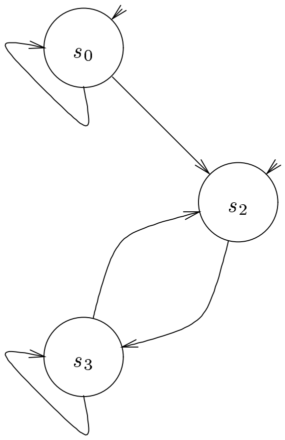

Figure 2.3: The reachable subgraph of $\hat{G}$

Remark 2.6 [Finite vs. finitely branching vs. finitely reaching] Every finite transition graph is finitely reaching. If a transition graph G is both finitely branching and finitely reaching, then the reachable subgraph G^R is finite. ■

The most important questions in computer-aided verification can be phrased as reachability questions. A reachability question asks if any state from a given region is reachable in a given transition graph. If the reachable subgraph is finite, then the reachability question can be solved using graph-traversal algorithms (see Section 2.3).

> Reachability problem
>
> An instance (G, σ^T) of the reachability problem consists of (1) a transition graph G and (2) a region σ^T of G, which is called the target region. The answer to the reachability question (G, σ^T) is Yes if a state in the target region σ^T is reachable, and otherwise No. A witness for a Yes-instance (G, σ^T) of the reachability problem is an initialized trajectory of G whose sink is in σ^T.

Remark 2.7 [Emptiness problem for finite automata] The reachability problem is equivalent to the one-letter emptiness problem for finite automata, which asks if a given finite automaton with a singleton input alphabet accepts any input word. To see this, view the target region as an accepting region. ■

2.2 Invariants

To an observer, only the values of the interface and external variables of a module are visible. We therefore specify properties of module states by constraints on

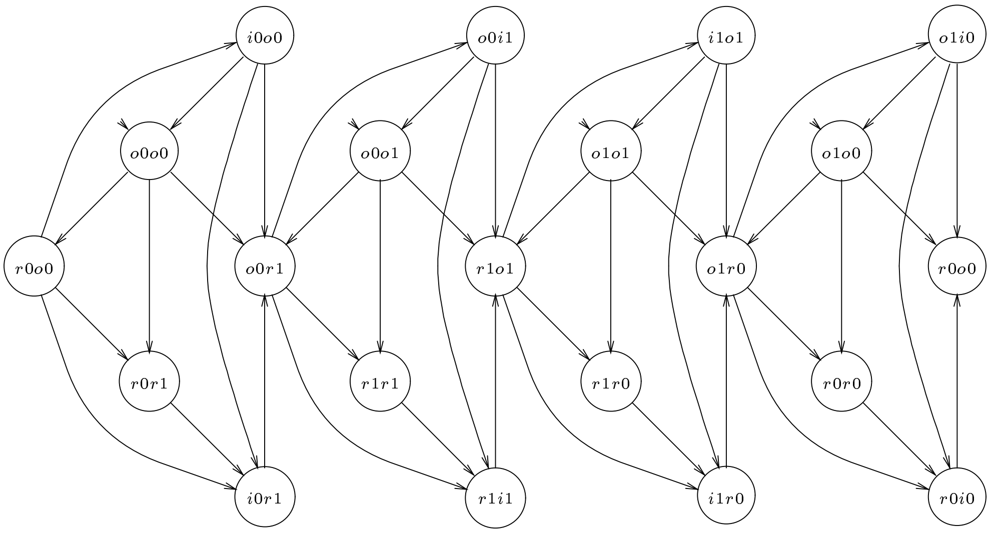

Figure 2.4: The reachable subgraph of G_{Pete}

the values of the observable variables. For example, if x is an observable integer variable, then the constraint $x > 5$ is satisfied by the states that map x to a value greater than 5, and the constraint is violated by the states that map x to a value less than or equal to 5. A constraint r on the observable variables of a module P is an invariant of P if all reachable states of P satisfy r. If r is an invariant of P, then it cannot happen that within a finite number of rounds, the module P moves into a state that violates r. Many important requirements on the behavior of reactive modules can be expressed as invariants.

Example 2.7 [Mutual exclusion] Peterson's protocol meets the mutual-exclusion requirement iff the constraint

$$r^{mutex}: \quad \neg(pc_1 = inC \wedge pc_2 = inC)$$

is an invariant of the module *Pete*. This constraint asserts that at most one of the two processes is inside its critical section. Note that the status of each process can be observed, because both pc_1 and pc_2 are interface variables. It is evident from inspecting the reachable subgraph of G_{Pete} (see Example 2.6) that every reachable state of *Pete* satisfies the constraint r^{mutex}. It follows that r^{mutex} is an invariant of the module *Pete*. ■

2.2.1 The Invariant-Verification Problem

If P is a reactive module, then a constraint on the values of module variables is called a state predicate for P. We do not allow the occurrence of event variables in state predicates, because the value of an event variable in any given state is immaterial. A state predicate that constrains only the values of observable variables is called an observation predicate. In particular, observation predicates cannot constrain the values of private variables.

> State predicate
>
> Let P be a reactive module. A state predicate for P is a boolean expression over the set $X_P \backslash \mathsf{event} X_P$ of module variables that are not event variables. The state predicate q is an observation predicate if all free variables of q are observable variables of P. The observation predicate q is an interface predicate if all free variables of q are interface variables, and q is an external predicate if all free variables of q are external variables. Given a state predicate q for P, we write $[\![q]\!]_P$ for the set of states of P that satisfy q.

Remark 2.8 [Regions defined by state predicates] Let P be a module, and let q and r be two state predicates for P. We say that the state predicate p defines the region $[\![q]\!]_P = \{s \in \Sigma_P \mid s \models q\}$ of P. The regions of P that are definable by state predicates form a boolean algebra:

$[\![\mathit{true}]\!]_P = \Sigma_P$ and $[\![\mathit{false}]\!]_P = \emptyset$;
$[\![q \wedge r]\!]_P = [\![q]\!]_P \cap [\![r]\!]_P$ and $[\![q \vee r]\!]_P = [\![q]\!]_P \cup [\![r]\!]_P$;
$[\![\neg q]\!]_P = \Sigma_P \backslash [\![q]\!]_P$.

■

> Invariant
>
> Let P be a reactive module, and let r be an observation predicate for P. The predicate r is an invariant of P if all reachable states of P satisfy r.

In other words, given a module P with the reachable region σ^R, the observation predicate r is an invariant of P iff $\sigma^R \subseteq [\![r]\!]_P$.

Remark 2.9 [Monotonicity of invariants] Let P be a module, and let q and r be two observation predicates for P. (1) The observation predicate *true* is an invariant of P. If q is an invariant of P, and $q \rightarrow r$ is valid, then r is also an invariant of P. It follows that every valid observation predicate for P is an invariant of P. (2) If both q and r are invariants of P, then $q \wedge r$ is also an invariant of P. ■

> Invariant-verification problem
>
> An instance (P, r) of the invariant-verification problem consists of (1) a reactive module P and (2) an observation predicate r for P. The answer to the invariant-verification question (P, r) is Yes if r is an invariant of P, and otherwise No. An error trajectory for a No-instance (P, r) of the invariant-verification problem is an initialized trajectory of P whose sink violates r.

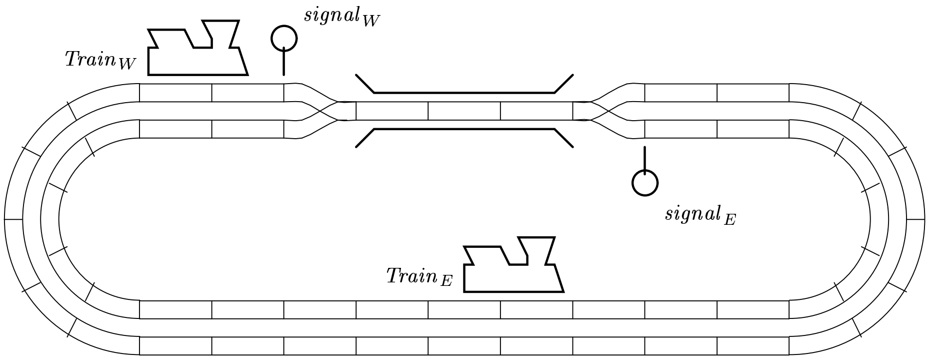

Figure 2.5: Railroad example

If the observation predicate r is not an invariant of the module P, then error trajectories present evidence to the designer of P as to how the module can end up in a state that violates r. Error trajectories thus provide valuable debugging information on top of the answer No to an invariant-verification question.

Example 2.8 [Railroad control] Figure 2.5 shows two circular railroad tracks, one for trains that travel clockwise, and the other for trains that travel counterclockwise. At one place in the circle, there is a bridge which is not wide enough to accommodate both tracks. The two tracks merge on the bridge, and for controlling the access to the bridge, there is a signal at either entrance. If the signal at the western entrance is green, then a train coming from the west may enter the bridge; if the signal is red, the train must wait. The signal at the eastern entrance to the bridge controls trains coming from the east in the same fashion.

A train is modeled by the asynchronous and passive module *Train* shown in Figure 2.6. When the train approaches the bridge, it sends an *arrive* event to the railroad controller and checks the signal at the entrance to the bridge ($pc = wait$). When the signal is red, the train stops and keeps checking the signal. When the signal is green, the train proceeds onto the bridge ($pc = bridge$). When the train exits from the bridge, it sends a *leave* event to the controller and travels around the circular track ($pc = away$). The traveling around the circular track, the checking of the signal, and the traveling time across the bridge each take an unknown number of rounds. There are two trains, one traveling clockwise and the other traveling counterclockwise. The first train, which arrives at the western entrance of the bridge, is represented by the module

module $Train_W$ **is**
$Train[pc, arrive, signal, leave := pc_W, arrive_W, signal_W, leave_W]$,

module *Train* **is**
  interface $pc\colon \{away, wait, bridge\};\ \ arrive, leave\colon \mathbb{E}$
  external $signal\colon \{green, red\}$

  lazy atom controls *arrive* **reads** *pc*
    update
      $[\!]\ pc = away \rightarrow arrive!$

  lazy atom controls *leave* **reads** *pc*
    update
      $[\!]\ pc = bridge \rightarrow leave!$

  lazy atom controls *pc* **reads** $pc, arrive, leave, signal$ **awaits** $arrive, leave$
    init
      $[\!]\ true \rightarrow pc' := away$
    update
      $[\!]\ pc = away \wedge arrive? \rightarrow pc' := wait$
      $[\!]\ pc = wait \wedge signal = green \rightarrow pc' := bridge$
      $[\!]\ pc = bridge \wedge leave? \rightarrow pc' := away$

Figure 2.6: Train

and the second train, which arrives at the eastern entrance, is represented by the module

module $Train_E$ **is**
  $Train[pc, arrive, signal, leave := pc_E, arrive_E, signal_E, leave_E].$

We are asked to design a passive controller module *Controller* that prevents collisions between the two trains by ensuring the train-safety requirement that in all rounds, at most one train is on the bridge. The module *Controller* enforces the train-safety requirement iff the observation predicate

$$r^{safe}\colon\quad \neg(pc_W = bridge \wedge pc_E = bridge)$$

is an invariant of the compound module

module *RailroadSystem* **is**
  hide $arrive_W, arrive_E, leave_W, leave_E$ **in**
    $\|\ Train_W$
    $\|\ Train_E$
    $\|\ Controller.$

The external variables of the module *Controller* should be $arrive_W$, $arrive_E$, $leave_W$, and $leave_E$.

module *Controller1* **is**

  interface $signal_W, signal_E : \{green, red\}$

  external $arrive_W, arrive_E, leave_W, leave_E : \mathbb{E}$

  passive atom controls $signal_W, signal_E$

    reads $signal_W, signal_E, arrive_W, arrive_E, leave_W, leave_E$

    awaits $arrive_W, arrive_E, leave_W, leave_E$

    init

      $[\!]\ true \rightarrow signal'_W := green;\ signal'_E := green$

    update

      $[\!]\ arrive_W? \rightarrow signal'_E := red$

      $[\!]\ arrive_E? \rightarrow signal'_W := red$

      $[\!]\ leave_W? \rightarrow signal'_E := green$

      $[\!]\ leave_E? \rightarrow signal'_W := green$

Figure 2.7: First attempt at railroad control

pc_W	*away*	*wait*	*bridge*	*away*	*away*	*wait*	*bridge*
$arrive_W$							
$leave_W$							
pc_E	*away*	*wait*	*wait*	*wait*	*bridge*	*bridge*	*bridge*
$arrive_W$							
$leave_W$							
$signal_W$	*green*	*green*	*green*	*green*	*green*	*green*	*green*
$signal_E$	*green*	*red*	*red*	*green*	*green*	*red*	*red*

Figure 2.8: An error trajectory that violates train safety

```
module Controller2 is
  private near_W, near_E : 𝔹
  interface signal_W, signal_E : {green, red}
  external arrive_W, arrive_E, leave_W, leave_E : 𝔼
  passive atom controls near_W
    reads near_W, arrive_W, leave_W
    awaits arrive_W, leave_W
    init
      [] true → near'_W := false
    update
      [] arrive_W? → near'_W := true
      [] leave_W?  → near'_W := false
  passive atom controls near_E
    reads near_E, arrive_E, leave_E
    awaits arrive_E, leave_E
    init
      [] true → near'_E := false
    update
      [] arrive_E? → near'_E := true
      [] leave_E?  → near'_E := false
  lazy atom controls signal_W, signal_E
    reads near_W, near_E, signal_W, signal_E
    init
      [] true → signal'_W := red; signal'_E := red
    update
      [] near_W ∧ signal_E = red → signal'_W := green
      [] near_E ∧ signal_W = red → signal'_E := green
      [] ¬near_W                 → signal'_W := red
      [] ¬near_E                 → signal'_E := red
```

Figure 2.9: Second attempt at railroad control

Figure 2.7 shows a first attempt at designing the railroad controller. Initially, both signals are green. A signal turns red whenever a train approaches the opposite entrance to the bridge, and it turns back to green whenever that train exits from the bridge. If both trains approach the bridge in the same round, then only one of the two signals turns red (the one that turns red is chosen nondeterministically), and the other train is admitted to the bridge. Unfortunately, the resulting railroad system does not have the invariant r^{safe}. This is evidenced by the error trajectory shown in Figure 2.8, which leads to a state with both trains on the bridge. If both trains approach the bridge simultaneously, then one is admitted to the bridge. When that train exits from the bridge, the other train is admitted to the bridge. At that point both signals are green. So when the first train returns while the second train is still on the bridge, the two trains will collide. It can be checked that the state sequence shown in Figure 2.8 is in fact the shortest initialized trajectory whose sink violates r^{safe}. ■

Exercise 2.5 {P2} [Railroad control] Figure 2.9 shows a second attempt at designing a railroad controller for Example 2.8. How many states does the module $\mathit{Train}_W \parallel \mathit{Train}_E \parallel \mathit{Controller2}$ have? How many of these states are reachable? Is there a reachable state with both trains on the bridge? To answer the latter two questions, draw the reachable subgraph of the transition graph. ■

2.2.2 From Invariant Verification to Reachability

Given a reactive module P, the execution of P generates a single initialized trajectory of P. By contrast, for solving an invariant-verification question about P, we must systematically explore all initialized trajectories of P. This can be done by solving a reachability question about the underlying transition graph G_P.

Proposition 2.3 [Reduction from invariant verification to reachability] The answer to an instance (P, r) of the invariant-verification problem is Yes iff the answer to the instance $(G_P, [\![\neg r]\!]_P)$ of the reachability problem is No. Furthermore, if (P, r) is a No-instance of the invariant-verification problem, then every witness for the reachability question $(G_P, [\![\neg r]\!]_P)$ is an error trajectory for the invariant-verification question (P, r).

It follows that we can answer the question if an observation predicate r is an invariant of a reactive module P if we can solve the reachability question $(G_P, [\![\neg r]\!]_P)$. Furthermore, if r is not an invariant of P, then we can provide an error trajectory by generating a witness for the Yes-instance $(G_P, [\![\neg r]\!]_P)$ of the reachability problem. We will discuss several algorithms for solving and generating witnesses for reachability questions. Yet it is important to clearly distinguish between the two problems: the input to the invariant-verification problem is a module and an observation predicate; the input to the reachability problem is a transition graph and a region. While the former can be reduced to the latter, this reduction typically requires exponential amount of work: indeed, as we shall see, invariant verification is inherently harder —for finite state spaces, by an exponential factor— than reachability.

Remark 2.10 [State predicates as invariants] Our formulation of the invariant-verification problem allows us to check whether r is an invariant of a module P when r refers only to the observable variables of P. The prohibition of requirements that refer to private variables is a good specification discipline, which can be exploited by reduction techniques such as minimization (see Chapter 5). However, it should be evident that one can check whether all reachable states of a module P satisfy a state predicate r by solving the reachability question $(G_P, [\![\neg r]\!]_P)$ even when r refers to private variables of P. ■

Exercise 2.6 {T1} [Transition invariants] Invariants cannot be used to directly specify module requirements that involve events, because observation predicates are interpreted over individual states. It is possible, however, to generalize invariants from observation predicates to transition predicates, which are interpreted over individual transitions and therefore can refer to the presence and absence of events. Let P be a reactive module. A transition predicate for P is a boolean expression over the set $X_P \cup X'_P$ of unprimed and primed module variables, with the restriction that event variables can occur only in subexpressions of the form $x?$ (which stands for $x' \neq x$). A pair (s, t) of states of P satisfies the transition predicate r' if $(s \cup t) \models r'$. It follows that every transition predicate r' defines an action $[\![r']\!]_P \subseteq \Sigma_P^2$, which contains all pairs of states of P that satisfy r'. The transition predicate r' is observable if no private variables (unprimed or primed) of P occur in r'. The observable transition predicate r' is a transition invariant of P if all reachable transitions of P satisfy r'; that is, if $\rightarrow_P^R \subseteq [\![r']\!]_P$. An instance (P, r') of the transition-invariant verification problem consists of (1) a reactive module P and (2) an observable transition predicate r' for P. The answer to the transition-invariant question (P, r') is Yes if r' is a transition invariant of P, and otherwise No.

Define a notion of error trajectories for the transition-invariant problem and reduce the problem, including the generation of error trajectories, to the following transition-reachability problem. An instance (G, α^T) of the transition-reachability problem consists of (1) a transition graph G and (2) an action α^T of G, which is called the target action. The answer to the transition-reachability question (G, α^T) is Yes if a transition in the target action α^T is reachable, and otherwise No. The answer Yes can be witnessed by an initialized trajectory of G of the form $\overline{s}_{1..m}$ with $m \geq 2$ and $(s_{m-1}, s_m) \in \alpha^T$. ■

2.2.3 Monitors

Invariants can distinguish between two trajectories only if one of the trajectories contains a state that does not occur on the other trajectory. Hence there are

$$
\begin{array}{l}
\textbf{module } \mathit{MonMonitor} \textbf{ is} \\
\quad \textbf{interface } \mathit{alert} \colon \{0,1\} \\
\quad \textbf{external } x \colon \mathbb{N} \\
\quad \textbf{passive atom controls } \mathit{alert} \textbf{ reads } x \textbf{ awaits } x \\
\quad\quad \textbf{init} \\
\quad\quad\quad [\!] \; \mathit{true} \to \mathit{alert}' := 0 \\
\quad\quad \textbf{update} \\
\quad\quad\quad [\!] \; x' \geq x \to \mathit{alert}' := 0 \\
\quad\quad\quad [\!] \; x' < x \to \mathit{alert}' := 1
\end{array}
$$

Figure 2.10: Monitoring monotonicity

requirements on the behavior of a reactive module P that cannot be expressed as invariants of P. However, many such requirements can be expressed as invariants of the compound module $P \| M$, for a monitor M of P. The module M is a monitor of P if (1) M is compatible with P and (2) $\mathsf{intf}X_M \cap \mathsf{extl}X_P = \emptyset$. If M is a monitor of P, then in every round, M may record the values of the observable variables of P, but M cannot control any external variables of P. Thus the monitor M can watch but not interfere with the behavior of P. In particular, the monitor M may check if P meets a requirement, and it may signal every violation of the requirement by sounding an observable alarm. The module P then meets the given requirement iff the compound module $P \| M$ has the invariant that no alarm is sounded by the monitor M.

Consider, for example, a module P with an interface variable x that ranges over the nonnegative integers. Assume that, during every update round, it is ok for P to increase the value of x, or to leave it unchanged, but it is not ok for P to decrease the value of x. This monotonicity requirement cannot be expressed as an invariant of P. However, we can design a monitor *MonMonitor* of P so that the monotonicity requirement can be expressed as an invariant of the compound module $P \,\|\, \mathit{MonMonitor}$. The monitor *MonMonitor*, shown in Figure 2.10, has but one variable, *alert*, which is an interface variable and ranges over the set $\{0,1\}$ of two alertness levels. The monitor *MonMonitor* watches for changes in the value of x. In every update round, if the value of x does not decrease, then the new value of *alert* is 0, which indicates that there is no reason for concern; if the value of x decreases, then the new value of *alert* is 1, which sounds an alarm. The module P then meets the monotonicity requirement iff the observation predicate $\mathit{alert} \neq 1$ is an invariant of the module $P \,\|\, \mathit{MonMonitor}$.

For a slightly more involved variation of the previous example, assume that it is ok for P to decrease the value of x occasionally, but it is not ok to decrease the value of x twice in a row, during two consecutive update rounds. Figure 2.11

module *AltMonitor* **is**
  interface $alert\colon \{0,1,2\}$
  external $x\colon \mathbb{N}$
  atom controls $alert$ **reads** $alert, x$ **awaits** x
    init
      $[\!]\ true \to alert' := 0$
    update
      $[\!]\ x' \geq x \to alert' := 0$
      $[\!]\ alert = 0 \wedge x' < x \to alert' := 1$
      $[\!]\ alert = 1 \wedge x' < x \to alert' := 2$

Figure 2.11: Monitoring alternation

module *EqOppMonitor* **is**
  interface $alert\colon \{0,1,2,3\}$
  external $pc\colon \{away, wait, bridge\};\ signal_1, signal_2\colon \{green, red\}$
  passive atom controls $alert$ **reads** $alert, pc, signal_1, signal_2$
    init
      $[\!]\ true \to alert' := 0$
    update
      $[\!]\ alert = 0 \wedge pc = wait \wedge signal_1 = red \wedge signal_2 = green \to alert' := 1$
      $[\!]\ alert = 1 \wedge signal_1 = green \to alert' := 0$
      $[\!]\ alert = 1 \wedge signal_1 = red \wedge signal_2 = red \to alert' := 2$
      $[\!]\ alert = 2 \wedge signal_1 = green \to alert' := 0$
      $[\!]\ alert = 2 \wedge signal_1 = red \wedge signal_2 = green \to alert' := 3$

Figure 2.12: Monitoring equal opportunity

shows a monitor of P that checks this alternation requirement. The interface variable *alert* of the monitor *AltMonitor* ranges over the set $\{0, 1, 2\}$ of three alertness levels. If the value of x does not decrease during an update round, then $alert = 0$, which indicates that there is no immediate danger of P violating the alternation requirement; if $alert = 0$ and the value of x decreases during an update round, then $alert = 1$, which indicates that there is an immediate danger of P violating the alternation requirement; if $alert = 1$ and the value of x decreases during an update round, then $alert = 2$, which indicates that P has violated the alternation requirement. The module P then meets the alternation requirement iff the observation predicate $alert \neq 2$ is an invariant of the module $P \,\|\, AltMonitor$.

Example 2.9 [Railroad control] This is a continuation of Example 2.8. Fig-

pc_W	*away*	*wait*	*wait*	*wait*	*wait*	*wait*	*wait*	*wait*	*wait*
$arrive_W$									
$leave_W$									
pc_E	*away*	*away*	*wait*	*wait*	*bridge*	*away*	*wait*	*wait*	*bridge*
$arrive_W$									
$leave_W$									
$signal_W$	*red*	*red*	*red*	*red*	*red*	*red*	*red*	*red*	*red*
$signal_E$	*red*	*red*	*red*	*green*	*green*	*red*	*red*	*green*	*green*
$alert_W$	0	0	0	0	1	1	2	2	3
$alert_E$	0	0	0	0	0	0	0	0	0

Figure 2.13: An error trajectory that violates equal opportunity

ure 2.9 presents an asynchronous railroad controller that enforces the train-safety requirement. Yet the module *Controller2* is not a satisfactory railroad controller, because it may keep a train waiting at a red signal while the other train is allowed to cross the bridge repeatedly. In particular, the resulting railroad system does not meet the equal-opportunity requirement that, while a train is waiting at a red signal, it is not possible that the signal at the opposite entrance to the bridge turns from green to red and back to green. Since the equal-opportunity requirement is violated by trajectories, and not by individual states, we need to employ monitors. The module

$$\textbf{module } \mathit{EqOppMonitor}_W \textbf{ is}$$
$$\mathit{EqOppMonitor}[\mathit{alert}, \mathit{pc}, \mathit{signal}_1, \mathit{signal}_2 := \mathit{alert}_W, \mathit{pc}_W, \mathit{signal}_W, \mathit{signal}_E]$$

monitors the equal-opportunity requirement for the train that travels clockwise, where *EqOppMonitor* is shown in Figure 2.12. The monitor has four levels of alertness. The alertness level is 0 as long as the train is not waiting at a red signal while the other signal is green, in which case the alertness level rises to 1. The alertness level rises to 2 when the other signal turns red, and to 3, when the other signal turns green again, while the train is still waiting at a red signal. An alertness level of 3 sounds an alarm that indicates a violation of the equal-opportunity requirement for the train that travels clockwise. The equal-opportunity requirement for the train that travels counterclockwise is monitored by the module

$$\textbf{module } \mathit{EqOppMonitor}_E \textbf{ is}$$
$$\mathit{EqOppMonitor}[\mathit{alert}, \mathit{pc}, \mathit{signal}_1, \mathit{signal}_2 := \mathit{alert}_E, \mathit{pc}_E, \mathit{signal}_E, \mathit{signal}_W]$$

in the same manner. The module *RailroadSystem* then meets the equal-opportunity requirement iff the observation predicate

$$\neg(\mathit{alert}_W = 3 \vee \mathit{alert}_E = 3)$$

is an invariant of the compound module

$$\mathit{RailroadSystem} \parallel \mathit{EqOppMonitor}_W \parallel \mathit{EqOppMonitor}_E.$$

The error trajectory of Figure 2.13 shows that this is not the case. ■

Exercise 2.7 {P3} [Mutual exclusion] The first-request-first-in requirement for mutual-exclusion protocols asserts that the first process to request admission to the critical section (meaning: $pc = reqC$) is the first process with an opportunity to enter the critical section (meaning: the guard is true for some guarded command that updates pc from $reqC$ to inC). (If both processes request to enter simultaneously, no order is specified.) (a) Write a monitor that checks the first-request-first-in requirement for mutual-exclusion protocols, and reduce the question of whether a mutual-exclusion protocol meets the first-request-first-in requirement to a invariant-verification question. (b) Does Peterson's mutual-exclusion protocol (Figure 1.23) meet the first-request-first-in requirement? What about the synchronous mutual-exclusion protocol from Figure 1.22? (c) How does the first-request-first-in requirement relate to the accessibility requirement specified in Chapter 1? (Does one imply the other?)

The equal-opportunity requirement for mutual-exclusion protocols asserts that, while a process is requesting to enter its critical section, it is not possible that the other process enters its critical section more than once. Equal opportunity, then, is a weaker requirement than first-request-first-in. Repeat parts (a)–(c) for the equal-opportunity requirement. ■

2.3 Graph Traversal

The reachability problem, and therefore the invariant-verification problem, can be solved by classical graph-search algorithms.

2.3.1 Reachability Checking

Graph-search algorithms traverse a graph one edge at a time, moving from a given vertex to its successor (or predecessor) vertices. It is useful to view these algorithms in terms of the following notions.

Predecessor and successor regions

Let $G = (\Sigma, \sigma^I, \to)$ be a transition graph, and let s be a state of G. The state t of G is a predecessor of s if $t \to s$, and t is a successor of s if $s \to t$. The predecessor region $pre_G(s)$ of s is the set of predecessors of s, and the successor region $post_G(s)$ of s is the set of successors of s. We write $pre^*_G(s)$ for the so-called source region $(\cup\, i \in \mathbb{N} \mid pre^i_G(s))$ of s, and $post^*_G(s)$ for the sink region $(\cup\, i \in \mathbb{N} \mid post^i_G(s))$.

Algorithm 2.2 [Enumerative Graph Search] (schema)

Input: a transition graph $G = (\Sigma, \sigma^I, \rightarrow)$.
Output: the reachable region σ^R of G.
Local: a multiset τ of states from Σ.

```
Initialize σ^R to ∅;
Initialize τ to σ^I;
while τ ≠ ∅ do
  Choose a state s in τ, and remove s from τ;
  if s ∉ σ^R then
    Add s to σ^R;
    Add all states in post_G(s) to τ
    fi
  od.
```

In other words, given a transition graph G and a state s of G, the source region $pre_G^*(s)$ contains the sources of all trajectories of G with sink s, and the sink region $post_G^*(s)$ contains the sinks of all trajectories of G with source s.

Terminology. The functions pre_G, $post_G$, pre_G^*, and $post_G^*$ are extended to regions in the natural way: for a region σ of the transition graph G, let $pre_G(\sigma) = (\cup\ s \in \sigma \mid pre_G(s))$ —i.e., the region $pre_G(\sigma)$ contains the predecessors of all states in σ— let $post_G(\sigma) = (\cup\ s \in \sigma \mid post_G(s))$ —i.e., the region $post_G(\sigma)$ contains the successors of all states in σ— etc. As usual, if the transition graph underlies a module P, we write pre_P instead of pre_{G_P}, etc. If the transition graph is understood, we suppress the subscript altogether. ■

Remark 2.11 [Reachability] Let G be a transition graph with the initial region σ^I, let s be a state of G, and let σ^T be a region of G. The state s is reachable in i transitions iff $s \in post^i(\sigma^I)$, and s is reachable iff $s \in post^*(\sigma^I)$; that is, $post^*(\sigma^I)$ is the reachable region of G. The transition graph G is finitely reaching iff there is a nonnegative integer i such that $post^*(\sigma^I) = (\cup\ j \leq i \mid post^j(\sigma^I))$. The answer to the reachability question (G, σ^T) is Yes iff $post^*(\sigma^I) \cap \sigma^T \neq \emptyset$ or, equivalently, iff $\sigma^I \cap pre^*(\sigma^T) \neq \emptyset$. ■

Enumerative graph search

Algorithm 2.2 shows a generic schema for graph search. As the algorithm finds new reachable states, they are explored by traversing all transitions to successor states. Throughout the algorithm, the multiset τ, which is called the frontier, contains the states that have been found but not yet explored; the set σ^R always contains the states that have been both found and explored. Algorithm 2.2 is said to be enumerative, because the states in the frontier τ are processed one state at a time. Therefore the multiset τ is best implemented by an enumeration of its members. If τ is implemented as a queue (when choosing a state from τ, always choose the state that was inserted least recently), then we obtain breadth-first search. If τ is implemented as a stack (always choose the state that was inserted most recently), then we obtain depth-first search. Algorithm 2.2 terminates iff the reachable subgraph G^R of the input graph G is finite. Consider a state s of G with m_s^R reachable incoming transitions; that is, $m_s^R = |pre(s) \cap \sigma^R|$. If s is reachable but not initial, then s is added to the frontier τ exactly m_s^R times; if s is initial, then s is added to τ exactly $1 + m_s^R$ times; if s is not reachable, then s is never added to τ. Every iteration of the while loop removes one state from τ. It follows that the while loop is executed $n^I + m^R$ times, where n^I is the number of initial states of G, and m^R is the number of reachable transitions.

Lemma 2.3 [Enumerative graph search] Let G be a transition graph with n^I initial states and m^R reachable transitions. Algorithm 2.2 computes the reachable region σ^R within $n^I + m^R$ iterations of the while loop.

Remark 2.12 [Backward search] Algorithm 2.2 performs a forward search of the input graph, starting from the initial region. Symmetrically, the graph may be searched backward from the target region, using the predecessor operation *pre* instead of the successor operation *post*. While forward search explores only reachable states, this is not necessarily the case for backward search. Hence the running time of backward search cannot be bounded by the number of reachable transitions. ■

Depth-first reachability checking

Algorithm 2.3 shows a recursive depth-first implementation of graph search for solving the reachability problem. The implementation differs from the schematic Algorithm 2.2 in three respects. First, for checking reachability, the graph search is aborted when a state in the target region is found. Second, the recursive implementation of depth-first search allows the construction of witnesses without bookkeeping. Third, the input graph is assumed to be finitely branching and the input region is assumed to be finite, so that the initial region, the successor region of each state, and the target region all can be represented as queues of states. More specifically, Algorithm 2.3 uses the following abstract types. Assuming a given type **state** for states, the type of a finitely branching transition graph is **enumgraph**, and the type of a finite region is **enumreg**. The abstract type **enumgraph** supports two operations:

InitQueue: **enumgraph** $\mapsto$ **queue of state**. The operation $InitQueue(G)$ returns a queue that contains the initial states of G, in some order.

Algorithm 2.3 [Depth-first Reachability Checking]

Input: a finitely branching transition graph G, and a finite region σ^T of G.
Output: Done, if the instance (G, σ^T) of the reachability problem has the answer No; a witness for the reachability question (G, σ^T), otherwise.

```
input G: enumgraph;  σ^T: enumreg;
local σ^R: enumreg;  τ: stack of state;  t: state;
begin
  σ^R := EmptySet;
  τ := EmptyStack;
  foreach t in InitQueue(G) do
    if DepthFirstSearch(t) then return Reverse(τ) fi
    od;
  return Done
  end.

function DepthFirstSearch(s): 𝔹
  local t: state;
  begin
    τ := Push(s, τ);
    if not IsMember(s, σ^R) then
      if IsMember(s, σ^T) then return true fi;
      σ^R := Insert(s, σ^R);
      foreach t in PostQueue(s, G) do
        if DepthFirstSearch(t) then return true fi;
        od
      fi;
    τ := Pop(τ);
    return false
    end.
```

PostQueue : **state** $\times$ **enumgraph** $\mapsto$ **queue of state**. The operation $PostQueue(s, G)$ returns a queue that contains the successors of s, in some order.

The abstract type **enumreg** supports three standard set operations:

EmptySet : **enumreg**. The operation *EmptySet* returns the empty region.

Insert : **state** $\times$ **enumreg** $\mapsto$ **enumreg**. The operation $Insert(s, \sigma)$ returns the region that results from adding the state s to the region σ.

IsMember : **state** $\times$ **enumreg** $\mapsto \mathbb{B}$. The operation $IsMember(s, \sigma)$ returns *true* if the region σ contains the state s, and otherwise returns *false*.

If all states in the target region are unreachable and the reachable subgraph of the input graph is finite, then the algorithm terminates once every reachable state is found; if some state in the target region is reachable in a finite number of transitions, then the algorithm may terminate even if the reachable subgraph is infinite. This is because as soon as a state in the target region σ^T is found, the search is aborted. At this point, the stack τ of unexplored states contains a witness for the given reachability question, in reverse order. To see this, observe that τ always contains an initialized trajectory of the input graph G, in reverse order.

Lemma 2.4 [Partial correctness of depth-first reachability checking] If Algorithm 2.3 terminates, then it solves the reachability question (G, σ^T) and returns a witness, if one exists.

2.3.2 Enumerative Graph and Region Representations

For the analysis of the time and space requirements of Algorithm 2.3, we need to agree on the representation of the abstract types **enumgraph** and **enumreg**. For this purpose, we restrict ourselves to finite input graphs. We distinguish between two cases, depending on whether or not the type **state** is atomic.

- In the state-level model, every variable of type **state** is stored in constant space, and constant time is required for every read or write access to a state. This is the standard model used in the analysis of graph algorithms. It is appropriate if the number of states is bounded. For example, for computers with 64-bit words, the state-level model is realistic if the number of states does not exceed 2^{64}; otherwise, the storage of a state requires more than a single word.
- The bit-level model is more detailed and makes no assumptions about the number of states. If the total number of states is n, then in the bit-level model, every variable of type **state** is stored in $\Theta(\log n)$ space, and $\Theta(\log n)$ time is required for every read or write access to a state. The bit-level model is of particular interest in computer-aided verification, where we regularly encounter very large state spaces.

In the following analysis, we consider a transition graph G with n states, n^I initial states, and m transitions, and we consider a region σ of G. We first discuss state-level data structures for representing G and σ, and then we move on to bit-level data structures.

State-level data structures

The finite transition graph G can be represented using adjacency lists, by a record $\{G\}_{se}$ with two components:

$$\textbf{enumgraph} \;=\; (\textbf{queue of state}) \times (\textbf{array}[\textbf{state}] \textbf{ of queue of state})$$

The first component of $\{G\}_{se}$ is a queue that contains the initial states of G. The second component of $\{G\}_{se}$ is an array, indexed by the states of G, which points, for each state s, to a queue that contains the successors of s. The record $\{G\}_{se}$ is called the state-enumerative representation of the transition graph G, because it is built from atomic components of the type **state** to facilitate the enumerative graph operations *InitQueue* and *PostQueue*. The state-level data structure $\{G\}_{se}$ requires $\Theta(n+m)$ space and supports the operations *InitQueue* and *PostQueue* in constant time. The state-enumerative representation of the region σ is a boolean array, denoted $\{\sigma\}_{se}$, which is indexed by the states of G, so that a state s is contained in σ iff $\{\sigma\}_{se}[s] = \mathit{true}$:

$$\textbf{enumreg} \;=\; \textbf{array}[\textbf{state}] \textbf{ of } \mathbb{B}$$

The state-level data structure $\{\sigma\}_{se}$ requires $\Theta(n)$ space and supports the enumerative region operations *EmptySet*, *Insert*, and *IsMember*: the first in $O(n)$ time, the second and third in constant time.

Remark 2.13 [Space-efficient state-level data structures] The state-enumerative graph and region representations $\{G\}_{se}$ and $\{\sigma\}_{se}$ optimize the running time, in the state model, of Algorithm 2.3. If we want to optimize, instead, the space requirements of the data structures, different choices are necessary. We can define an alternative state-enumerative graph representation $\{G\}_{se}^T$ which uses $\Theta(n^I + m)$ space, and an alternative state-enumerative region representation $\{\sigma\}_{se}^T$ which uses $\Theta(|\sigma|)$ space, both of which are optimal. In both cases, we replace the array indexed by states with a balanced binary search tree over states: the second component of the record $\{G\}_{se}^T$ is a tree whose nodes represent the states that have nonempty queues of successors; the nodes of the tree $\{\sigma\}_{se}^T$ represent the states that are members of the region σ. The search-tree implementations of the abstract data types **enumgraph** and **enumreg** support the operations *InitQueue* and *EmptySet* in constant time, the operation *PostQueue* in $O(\log n)$ time, and the operations *Insert* and *IsMember* in $O(\log |\sigma|)$ time. ■

Bit-level data structures

In the bit-level model, we cannot have arrays indexed by states, as the index elements are no longer representable by a fixed number of bits. Without loss of generality, we assume that each state of the transition graph G is identified by a unique sequence of $\lceil \log n \rceil$ bits. For example, the transition graph G_P of a module with k boolean variables has 2^k states, and each state can be represented by a sequence of k bits denoting the values of the module variables. The bit-enumerative representation $\{\sigma\}_{be}$ of the region σ is a binary tree whose paths represent the states of G that are members of σ. The height of the tree is $\lceil \log n \rceil$, the number of leaves is $|\sigma|$, and the total number of nodes is $\Theta(|\sigma| \cdot (1 + \log n - \log |\sigma|))$ or, less precisely, $\Theta(\min(|\sigma| \cdot \log n, n))$. In particular, if n is a power of 2 and σ contains all n states, then $\{\sigma\}_{be}$ is the complete binary tree with $2n$ nodes. The bit-enumerative representation $\{G\}_{be}$ of the transition graph G is, like the state-enumerative representation, a record whose first component is a queue of the initial states, and whose second component is an index structure over states which points to queues of successor states. The index structure is implemented as a binary tree whose paths represent the states of G that have nonempty queues of successors. A queue of pairwise distinct states can be implemented in a space-efficient way by sharing common suffixes of the bitvector representations of the states in the queue. This is achieved by a binary tree with child-to-parent pointers, whose leaf-to-root paths represent the states in the queue, and whose leaves are connected by pointers that represent the order of the states in the queue. If the queue contains ℓ states, then the bit-level queue representation requires $\Theta(\min(\ell \cdot \log n, n))$ bits. The following lemma completes the space and time analysis of the bit-enumerative data structures.

Lemma 2.5 [Bit-enumerative graph and region representations] Let G be a transition graph with n states, n^I initial states, and m transitions, and let s be a sequence of $\lceil \log n \rceil$ bits. The bit-enumerative graph representation $\{G\}_{be}$ uses $\Theta(\min(n^I \cdot \log n, n) + \min(m \cdot \log n, n^2))$ space. The operations *InitQueue*$(\{G\}_{be})$ and *PostQueue*$(s, \{G\}_{be})$ require $O(1)$ and $O(\log n)$ time, respectively. Let σ be a region of G. The bit-enumerative region representation $\{\sigma\}_{be}$ uses $\Theta(\min(|\sigma| \cdot \log n, n))$ space. The operation *EmptySet* requires $O(1)$ time; the operations *Insert*$(s, \{\sigma\}_{be})$ and *IsMember*$(s, \{\sigma\}_{be})$ each require $O(\log n)$ time.

Exercise 2.8 {`T2`} [Proof of Lemma 2.5] Let G be a transition graph whose states are the bitvectors of length k, and let σ be a region of G. Give formal definitions of the bit-enumerative graph and region representations $\{G\}_{be}$ and $\{\sigma\}_{be}$, and prove Lemma 2.5. ■

Time and space requirements of depth-first reachability checking

To determine the time complexity of Algorithm 2.3, let n be the total number of states of the input graph, let n^I be the number of initial states, and let m^R

be the number of reachable transitions. Recall the analysis of the schematic Algorithm 2.2. In particular, if s is an initial state with m_s^R reachable incoming transitions, then the function *DepthFirstSearch* is invoked with input state s at most $1+m_s^R$ times; if s is reachable but not initial, then *DepthFirstSearch* is invoked with input s at most m_s^R times; if s is unreachable, then *DepthFirstSearch* is never invoked with input s. The first time that *DepthFirstSearch* is invoked with input s, the function call performs $O(|post(s)|)$ state-level work, and $O(\log n + |post(s)|)$ bit-level work, in addition to invoking *DepthFirstSearch* for every successor of s. Each subsequent call of *DepthFirstSearch* with input state s terminates, after a single membership test $IsMember(s, \sigma^R)$, within constant state-level time and $O(\log n)$ bit-level time. It follows that the total time required by all invocations of *DepthFirstSearch* is, in the worst case, $O(n^I + m^R)$ time in the state-level model, and $O((n^I + m^R) \cdot \log n)$ time in the bit-level model. The worst case is obtained when no state in the target region is reachable. The initialization of the region σ^R requires $O(n)$ state-level time vs. constant bit-level time. The space complexity of Algorithm 2.3 is dominated by the space requirements of the input representations. The complete analysis is summarized in the following theorem.

Theorem 2.1 [Depth-first reachability checking] Let G be a finite transition graph with n states, of which n^I are initial, and m transitions, of which m^R are reachable. Let σ^T be a region of G. In the state-level model, given the input $\{G\}_{se}$ and $\{\sigma^T\}_{se}$, Algorithm 2.3 solves the reachability question (G, σ^T) and computes a witness, if one exists, in $O(n + m^R)$ time and $\Theta(n + m)$ space. In the bit-level model, given the input $\{G\}_{be}$ and $\{\sigma^T\}_{be}$, Algorithm 2.3 requires $O((n^I + m^R) \cdot \log n)$ time and $\Theta(\min(n^I \cdot \log n, n) + \min(m \cdot \log n, n^2) + \min(|\sigma^T| \cdot \log n, n))$ space.

Remark 2.14 [Time complexity of state-level reachability checking] In the state-level model, the running time of Algorithm 2.3 is proportional to the size n of the state space, no matter how quickly a state in the target region is found. This is caused by the initialization of the array $\{\sigma^R\}_{se}$ for representing the region of explored states. In practice, this behavior is undesirable, and alternative representations of the region σ^R are preferred. One such representation, based on search trees, is studied in Exercise 2.9; another one, based on hashing, and by far the most popular in practice, will be discussed in Section 2.3.3. ■

Exercise 2.9 {`T2`} [Space-efficient state-level data structures] Suppose that the input to Algorithm 2.3 is given by the alternative state-enumerative graph and region representations $\{G\}_{se}^T$ and $\{\sigma^T\}_{se}^T$ introduced in Remark 2.13, and that the region σ^R of explored states is stored in the same manner. What is the resulting time and space complexity of Algorithm 2.3 in the state-level model? Under which conditions on the input G and σ^T are the alternative data structures preferable in order to optimize running time? Under which conditions are they preferable in order to optimize memory space? ■

Algorithm 2.4 [Enumerative Invariant Verification] (schema)

> Input: a finite module P, and an observation predicate r for P.
> Output: Done, if the instance (P, r) of the invariant-verification problem has the answer Yes; an error trajectory for the invariant-verification question (P, r), otherwise.
>
> Construct the enumerative graph representation $\{G_P\}_e$;
> Construct the enumerative region representation $\{[\![\neg r]\!]_P\}_e$;
> Return the result of Algorithm 2.3 on the input $\{G_P\}_e$ and $\{[\![\neg r]\!]_P\}_e$.

Exercise 2.10 {P2} [Nonrecursive depth-first reachability checking] Write a non-recursive, state-level version of Algorithm 2.3 with the same time and space complexity, assuming that the input (G, σ^T) is given by the state-enumerative graph and region representations $\{G\}_{se}$ and $\{\sigma^T\}_{se}$. Be careful with the witness construction. ■

Exercise 2.11 {P3} [Breadth-first reachability checking] Algorithm 2.3 traverses the input graph in depth-first fashion. Write a breadth-first algorithm for reachability checking, including witness construction, assuming that the input (G, σ^T) is given, first, by the state-enumerative graph and region representations $\{G\}_{se}$ and $\{\sigma^T\}_{se}$, and second, by the bit-enumerative representations $\{G\}_{be}$ and $\{\sigma^T\}_{be}$. (Maintain the frontier τ of unexplored states as a queue.) Determine the time and space requirements of your algorithm in both the state-level and bit-level models. ■

Exercise 2.12 {P2} [Transition invariants] Modify Algorithm 2.3, without changing its state-level and bit-level time and space requirements, to solve the transition-reachability problem from Exercise 2.6. For this purpose, you must define state-enumerative and bit-enumerative representations for transition predicates. ■

2.3.3 Invariant Verification

For finite modules P, invariant-verification questions of the form (P, r) can be reduced to reachability checking, as is shown in Algorithm 2.4. In terms of the size of the input (P, r), the asymptotic amount of work for constructing and the asymptotic amount of space required for storing the enumerative representations of the transition graph G_P and the target region $[\![\neg r]\!]_P$ are independent of whether the state-level or bit-level model is used. Hence, as in Algorithm 2.4, if the argument G is a transition graph that underlies a module, we write $\{G\}_e$ as a place-holder for either the state-enumerative representation $\{G\}_{se}$ or the bit-enumerative representation $\{G\}_{be}$; similarly, we write $\{\sigma\}_e$ if the region σ is defined by a state predicate. The translation from the module P to the enumerative graph representation $\{G_P\}_e$, as well as the translation from the observation predicate r to the enumerative region representation $\{[\![\neg r]\!]_P\}_e$, may involve an exponential amount of work. To make these claims precise, we need to agree on a syntax for the legal expressions in the initial and update commands of reactive modules.

Propositional modules

The most basic type is the boolean type. In propositional modeling, we restrict ourselves to variables of this type, which are called propositions. If all variables of a module are propositions, then the module is said to be a propositional module. Every propositional module is finite, and dually, every finite module can be viewed propositionally —by replacing each variable of a finite type with k values by $\lceil k \rceil$ boolean variables. For the propositional modules, we now agree on a specific syntax. In particular, all expressions that occur in the textual description of a propositional module result from combining propositions using a standard set of logical connectives.

> Propositional module
>
> A proposition is a variable of type boolean. The propositional formulas are the boolean expressions generated by the grammar
>
> $$p \ ::= \ x \mid \mathit{true} \mid \mathit{false} \mid p_1 \wedge p_2 \mid p_1 \vee p_2 \mid \neg p_1 \mid p_1 \rightarrow p_2 \mid p_1 \leftrightarrow p_2,$$
>
> where x is a proposition, and p_1 and p_2 are propositional formulas. A propositional module is a reactive module P such that (1) all module variables of P are propositions, and (2) every expression that appears in the initial and update commands of P is a propositional formula.

Remark 2.15 [Transition graphs for propositional modules] If P is a propositional module with k variables, then the transition graph G_P has 2^k states and at most 4^k transitions. ■

From propositional modules to enumerative graph representations

Let P be a propositional module with k variables, and let $|P|$ be the number of symbols in the textual description of P. Note that $|P| > k$. In the first step of Algorithm 2.4, we need to construct the enumerative representation $\{G_P\}_e$ of the underlying transition graph. In the following analysis, it is immaterial whether or not each of the 2^k states can be stored and accessed atomically: in terms of the parameters $|P|$ and k, an asymptotically equal amount of work is required to construct, for sufficiently small k (say, $k \leq 64$), the state-level representation $\{G_P\}_{se}$ or, for arbitrary k, the bit-level representation $\{G_P\}_{be}$;

module *Nondet* **is**
  interface $x_1, \ldots, x_k : \mathbb{B}$
  atom controls x_1
    initupdate
      $[\!]$ $true \rightarrow x_1' := true$
      $[\!]$ $true \rightarrow x_1' := false$
  $\vdots$
  atom controls x_k
    initupdate
      $[\!]$ $true \rightarrow x_k' := true$
      $[\!]$ $true \rightarrow x_k' := false$

Figure 2.14: Unconstrained propositional module

we therefore use the notation $\{G_P\}_e$. To construct the queue of initial states of P, we generate each state s of P and check if s is an initial state of P. Similarly, for every state s, to construct the queue of successors of s, we generate each state t of P and check if (s,t) is a transition of P. Since each state of P is a bitvector of length k, we can generate all states in $O(2^k)$ time. The next lemma shows that each of the 2^k initiality checks and each of the 4^k transition checks can be performed in linear time. It follows that the construction of the enumerative graph representation $\{G_P\}_e$ can be completed in $O(4^k \cdot |P|)$ time.

Lemma 2.6 [Initial states and transitions for propositional modules] Given a propositional module P, and two states s and t of P, it can be checked in $O(|P|)$ time if s is an initial state of P and if (s,t) is a transition of P.

Exercise 2.13 {`T2`} [Proof of Lemma 2.6] Given a propositional module P, construct two propositional formulas q_P^I and q_P^T, whose lengths are linear in $|P|$: the initial predicate q_P^I is a boolean expression over the set X_P of module variables so that for every state s of P, the expression q_P^I evaluates to true in s iff s is an initial state of P; the transition predicate q_P^T is a boolean expression over the set $X_P \cup X_P'$ of unprimed and primed module variables so that for every pair (s,t) of states of P, the expression q_P^T evaluates to true in $s \cup t'$ iff (s,t) is a transition of P. Since each state of P is a bitvector of length k, where k is the number of propositions of P, and since $|P| > k$, Lemma 2.6 follows. ■

The construction time of the enumerative graph representation $\{G_P\}_e$ is exponential in the number k of variables. This exponential amount of work cannot be avoided, as the record $\{G_P\}_e$ may require exponentially more space than the textual description of the module P. Consider the propositional module *Nondet* with k boolean interface variables such that all initial values are arbitrary, and in every update round, the values of all variables can change arbitrarily. If every variable is controlled by a separate atom, then P can be specified using $\Theta(k)$ symbols, as shown in Figure 2.14. The transition graph G_{Nondet} is the complete graph with 2^k states, all of which are initial, and 4^k transitions. It follows that the enumerative graph representation $\{G_{Nondet}\}_e$ requires $\Theta(4^k)$ space, independent of whether we can use the state-level model or must resort to the bit-level model.

Propositional invariant verification

If we restrict our attention to propositional modules, then we obtain a special case of the invariant-verification problem.

> **Propositional invariant-verification problem**
>
> An instance (P, r) of the invariant-verification problem is *propositional* if P is a propositional module and r is a propositional formula. The instances of the *propositional invariant-verification problem* are the propositional instances of the invariant-verification problem. The propositional instance (P, r) *has* k *variables* if the module P has k module variables.

Let (P, r) be a propositional instance of the invariant-verification problem with k variables. The enumerative representation $\{[\![\neg r]\!]_P\}_e$ of the target region can be constructed in $O(2^k \cdot |r|)$ time, by generating each state s of P and checking if s satisfies the predicate r. The constructed data structure $\{[\![\neg r]\!]_P\}_e$ occupies $\Theta(2^k)$ space. Both construction time and memory space are independent of whether we use the state-enumerative region representation $\{[\![\neg r]\!]_P\}_{se}$ or the bit-enumerative region representation $\{[\![\neg r]\!]_P\}_{be}$. Together with Exercise 2.13 and Theorem 2.1, this gives exponential time and space bounds for solving the propositional invariant-verification problem which are independent of the state-level vs. bit-level issue.

Theorem 2.2 [Propositional invariant verification] Let (P, r) be a propositional instance of the invariant-verification problem with k variables. Algorithm 2.4 solves the invariant-verification question (P, r) and computes an error trajectory, if one exists, in $O(4^k \cdot (|P| + |r|))$ time and $\Theta(4^k)$ space.

Exercise 2.14 {T4} [Equational modules with interpreted constants] An *ic-equational term* is either a variable or an interpreted constant. Like variables, constants are typed. Each interpreted constant denotes a fixed value of its type. Examples of interpreted constants are the boolean constant *true* and the integer constant 19. In particular, for two interpreted constants, it is known if they denote equal or different values. The *ic-equational formulas* are the boolean expressions that are generated by the grammar

$$p ::= f_1 = f_2 \mid p_1 \wedge p_2 \mid \neg p_1,$$

where f_1 and f_2 are ic-equational terms of the same type, and p_1 and p_2 are ic-equational formulas. An ic-equational module is a reactive module P such that (1) every guard that appears in the initial and update commands of P is an ic-equational formula, and (2) every assignment that appears in the initial and update commands of P is an ic-equational term. An instance (P, r) of the invariant-verification problem is ic-equational if P is an ic-equational module and r is an ic-equational formula. The instance (P, r) is finite if the ic-equational module P is finite. Suppose we wish to use Algorithm 2.4 for solving the ic-equational instances of the invariant-verification problem. (a) Given an ic-equational instance (P, r) of the invariant-verification problem, find a finite propositional instance (P', r') that has the same answer as (P, r). (b) Write a preprocessor that, given the ic-equational module P and the ic-equational formula r, constructs the enumerative graph representation $\{G_{P'}\}_e$ and the enumerative region representation $\{[\![\neg r']\!]_{P'}\}_e$. What are the space requirements of your representations as a function of the size of the input (P, r)? What are the running times of your preprocessor and of Algorithm 2.4 in terms of the size of (P, r)? Does it make a difference if the state-level model or the bit-level model is used? ■

Exercise 2.15 {P3} [Backward search] Write a backward-search algorithm for reachability checking, and an invariant-verification algorithm that invokes your backward-search algorithm. In your algorithms, assume that the abstract type **enumgraph** supports the operation

$$\mathit{PreQueue}\colon \textbf{state} \times \textbf{enumgraph} \mapsto \textbf{queue of state}$$

which, given a transition graph G and a state s of G, returns a queue that contains the predecessors of s, in some order. Assuming that the state-level model is adequate, suggest a representation for transition graphs that supports the three operations *InitQueue*, *PostQueue*, and *PreQueue* in constant time, and write an algorithm that constructs your representation for the transition graphs of propositional modules. Give the running time required to construct and the memory space required to store your representation. ■

2.3.4 Three Space Optimizations

The exponential space requirements of Algorithm 2.4 (Theorem 2.2) are a serious obstacle to practical applications. The problem is caused by the enumerative graph representation $\{G_P\}_e$ for the input module P, and by the enumerative region representations for the sets σ^T of target states and σ^R of explored states, all of which require space at least proportional to the number of states of P. For many invariant-verification questions, the number of states is too large for the transition graph and its regions to be stored explicitly. The following three observations allow us to reduce the space requirements of invariant verification. First, on-the-fly methods avoid the enumerative representations of the input graph and target region. Second, state-hashing methods avoid the enumerative storage of the explored states. Third, latch-reduction methods reduce both the size and number of frontier states and explored states that need to be stored.

On-the-fly graph and region representations

Consider an instance (P, r) of the invariant-verification problem. Instead of constructing, at once, the space-intensive enumerative graph representation $\{G_P\}_e$ from the input module P, we can construct portions of the graph only as needed, whenever the operations *InitQueue* and *PostQueue* are invoked in Algorithm 2.3. Similarly, instead of constructing, at once, the space-intensive region representation $\{[\![\neg r]\!]_P\}_e$ from the input predicate r, we can answer each query in Algorithm 2.3 of the form $\mathit{IsMember}(s, \sigma^T)$ by evaluating the predicate $\neg r$ in the state s. Such an "on-the-fly" implementation of Algorithm 2.3 relies on the following data structures for representing the input graph and the target region.

- On-the-fly representations for graphs are restricted to transition graphs that underly modules. Given a module P, the on-the-fly representation $\{G_P\}_{of}$ for the transition graph G_P is a queue that contains the atoms of the module P in some execution order.

- On-the-fly representations for regions are restricted to regions that are defined by state predicates. Given a module P and a state predicate q for P, the on-the-fly representation $\{[\![q]\!]_P\}_{of}$ for the region $[\![q]\!]_P$ is the predicate q.

In an on-the-fly implementation of Algorithm 2.3, the input graph G is given by the queue $\{G_P\}_{of}$ and the input region σ^T is given by the predicate $\{[\![\neg r]\!]_P\}_{of}$ (the region σ^R of explored states, which is not defined by a state predicate, has no on-the-fly representation). Clearly, the space requirements of the on-the-fly graph and region representations $\{G_P\}_{of}$ and $\{[\![\neg r]\!]_P\}_{of}$ are linear in the input (P, r) to the invariant-verification problem. The overall time and space requirements of on-the-fly invariant verification for propositional modules are analyzed in the following exercise.

Exercise 2.16 {P3} [On-the-fly invariant verification for propositional modules] Consider a propositional module P with k variables, and an observation predicate r for P. (a) Write an algorithm that computes the on-the-fly representation $\{G_P\}_{of}$ of the underlying transition graph. What is the running time of your algorithm? (b) Write algorithms for computing the operations $\mathit{InitQueue}(\{G_P\}_{of})$, $\mathit{PostQueue}(s, \{G_P\}_{of})$, and $\mathit{IsMember}(s, \{[\![\neg r]\!]_P\}_{of})$, where s is a state of P. What are the running times of your algorithms? (c) Using the state-level array representation $\{\sigma^R\}_{se}$ for the region of explored states, solve the propositional instance (P, r) of the invariant-verification problem in

$O((2^k + m^R) \cdot (|P| + |r|))$ time and $\Theta(2^k + |P| + |r|)$ space, where m^R is the number of reachable transitions of the transition graph G_P. (d) Using the state-level search-tree representation $\{\sigma^R\}^T_{se}$ or the bit-level representation $\{\sigma^R\}_{be}$ for the region of explored states, solve the instance (P, r) of the invariant-verification problem in $O((n^I + m^R) \cdot (|P| + |r|))$ time and $\Theta(n^R + |P| + |r|)$ space, where n^I is the number of initial states and n^R is the number of reachable states of the transition graph G_P. ■

Remark 2.16 [On-the-fly invariant verification for finitely branching modules] The on-the-fly representation $\{G_P\}_{of}$ of the transition graph does not require that the input module P is finite. Neither does the on-the-fly representation $\{[\![\neg r]\!]_P\}_{of}$ of the target region require that the input predicate r evaluates to true in all but finitely many states of P. Rather, the on-the-fly implementation of Algorithm 2.3 can be applied to input modules with finitely branching transition graphs and infinite target regions. As observed earlier, Algorithm 2.3 is guaranteed to terminate if the reachable subgraph G_P^R of the input module P is finite. The algorithm may terminate even when G_P^R is infinite, if it visits a state that belongs to the target region. ■

Exercise 2.17 {P3} [Integer modules with addition] The integer terms with addition are the nonnegative integer expressions generated by the grammar

$$f \ ::= \ x \mid m \mid f_1 + f_2 \mid f_1 \dot{-} f_2,$$

where x is a variable of type $\mathbb{N}$, where m is a nonnegative integer (i.e., an interpreted constant of type $\mathbb{N}$), and f_1 and f_2 are integer terms with addition. The integer formulas with addition are the boolean expressions generated by the grammar

$$p \ ::= \ f_1 \leq f_2 \mid p \wedge p \mid \neg p,$$

where f_1 and f_2 are integer terms with addition. An integer module with addition is a reactive module P such that (1) all module variables of P are of type $\mathbb{N}$, (2) every guard that appears in the initial and update commands of P is an integer formula with addition, and (3) every assignment that appears in the initial and update commands of P is an integer term with addition. The instance (P, r) of the invariant-verification problem is an integer instance with addition if P is an integer module with addition and r is an integer formula with addition. (a) Suppose we wish to apply an on-the-fly implementation of Algorithm 2.3 to the integer instances with addition of the invariant-verification problem. In order to obtain a finitely branching transition graph, we need to restrict ourselves to integer modules with addition which are closed. Write algorithms that, given a closed integer module P with addition, computes the operations *InitQueue* and *PostQueue* for the transition graph G_P on the fly, and write an algorithm that, given an integer formula r with addition, computes the operation *IsMember* for the region $[\![\neg r]\!]_P$ on the fly. What are the running times of your algorithms? (b) Give an example of a closed, deterministic integer module P with addition and a state s of P so that the set $pre_P(s)$ is infinite. What are the ramifications for on-the-fly backward search for integer modules with addition? ■

Hashing of explored states

On-the-fly implementations of Algorithm 2.3 reduce the space required by the transition graph G and the target region σ^T, but they do not address the space required by the region σ^R of explored states. In particular, in the state-level model, neither the array representation $\{\sigma^R\}_{se}$ nor the search-tree representation $\{\sigma^R\}_{se}^T$ perform satisfactorily in practice: the array representation $\{\sigma^R\}_e$ is exponential in the number of input variables, even if the region σ^R, which is initially empty, remains small compared to the size of the state space; the search-tree representation $\{\sigma^R\}_{se}^T$ is space-optimal, but the rebalancing overhead involved in the frequent insertions adversely affects the verification time in practice. State hashing is a compromise which often offers the best practical performance. In state hashing, the region σ^R is represented by a hash table $\{\sigma^R\}_{se}^H$ that consists of (1) a hash function that maps each state s to an integer between 0 and N, for a suitably chosen nonnegative integer N, and (2) an array of length N whose i-th entry, for $1 \leq i \leq N$, points to a queue of states that are mapped to i by the hash function:

$$\textbf{enumreg} \;=\; (\textbf{state} \mapsto \{0..N\}) \times (\textbf{array}[0..N] \textbf{ of queue of state})$$

The choice of N is determined by the expected number of reachable states and by the word size of the computer on which the hash table is implemented; for example, $N = 2^{64} - 1$. The hash table $\{\sigma^R\}_{se}^H$ is a state-level data structure which uses $\Theta(N + |\sigma^R|)$ space. The running time of Algorithm 2.3 depends crucially on the complexity of the membership test for the hash table $\{\sigma^R\}_{se}^H$, which in turn depends on the choice of hash function and on the ratio of N to the number of explored states. A detailed analysis of hashing can be found in [Knuth:Vol.1].

Remark 2.17 [Bit-state hashing] While hashing is an effective technique to represent 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 i-th entry, for $1 \leq i \leq N$, is a single bit. The insertion of a state, which is mapped to an integer i between 0 and N by the hash function, is implemented by setting the i-th bit of the hash table to 1. All hash collisions are ignored. Suppose that two states s and t are mapped to the same integer i, and s is inserted in the hash table first. When the state t is encountered, as the i-th bit of the hash table is already set, the membership test returns a positive answer.

Consequently, Algorithm 2.3 does not explore the successors of t. Hence, only a fraction of the reachable region is explored. The algorithm may return false negatives (the false answer No for Yes-instances of the reachability problem), but no false positives (the false answer Yes for No-instances of the reachability problem). In particular, every error trajectory that is found indeed signals a violation of the invariant. More general approximation schemes will be discussed in detail in Chapter 5.

What fraction of the reachable region is visited by bit-state hashing depends on the choice of table size and hash function. The table size can be increased iteratively until either an error trajectory is found or all available memory space is used. The performance of bit-state hashing can be improved dramatically by using two bit-state hash tables that employ independent hash functions. Each explored state is stored in preferably both, but at least in one hash table, so that a collision occurs only if both table entries are already occupied. If N is the size of the hash tables, this strategy typically ensures that, if necessary, close to N reachable states are explored. ■

Latch reduction for event variables and history-free variables

If, for every state s of a module, the value $s(x)$ of the module variable x is not needed for determining the successors of s, then in Algorithm 2.2 the value of x does not have to be stored as part of the frontier τ of unexplored states nor as part of the region σ^R of explored states. This is the case for event variables, whose values in a given state are immaterial, and for variables whose values are never read, only awaited. The variables whose values are not read are called history-free; their values in a given state depend (possibly nondeterministically) on the values of other variables in the same state. The space savings that arise from not storing the values of event variables and history-free variables during graph search can be substantial. For example, in synchronous circuits, all variables that represent input and output wires of gates and latches are history-free, and only the variables that represent the internal states of the latches need to be stored. Motivated by this example, we refer to the variables that are neither event variables nor history-free as latched. The set of latched variables of a module can be computed easily from the module and atom declarations. The projection of the transition graph of a module to the latched variables is called the latch-reduced transition graph of the module.

Latch-reduced transition graph of a module

A variable x of the module P is latched if (1) x does not have the type $\mathbb{E}$, and (2) x is read by some atom of P. We write $latchX_P$ for the set of latched module variables of P. The latch-reduced transition graph of P is the transition graph $G_P^L = (\Sigma_P[latchX_P], \sigma_P^I[latchX_P], \to_P^L)$, where $s^L \to_P^L t^L$ iff there is a transition $s \to_P t$ such that $s^L = s[latchX_P]$ and $t^L = t[latchX_P]$.

Example 2.10 [Latch-reduced transition graph for three-bit counter] Recall the circuit from Figure 1.20 which realizes a three-bit binary counter. For the module *Sync3BitCounter*, all variables except the three output bits out_0, out_1, and out_2, are history-free. Consequently, the latch-reduced transition graph of *Sync3BitCounter* has only 8 states, which correspond to the possible values of the three output bits. Each state of the latch-reduced transition graph encodes a counter value, and there is a transition from state s to state t iff the value encoded by t is one greater (modulo 8) than the value encoded by s. ■

Exercise 2.18 {P1} [Latch-reduced transition graph for railroad control] Consider the module $Train_W \parallel Train_E \parallel Controller1$ of Example 2.8. Which variables are latched? Draw the latch-reduced transition graph of the module. ■

Remark 2.18 [Latch-reduced transition graphs] The latch-reduced transition graph G_P^L of a module P may be finite even if the transition graph G_P is not, and G_P^L may be finitely branching even if G_P is not. ■

The latched-reduced transition graph of a module can be used for invariant verification. Let (P, r) be an instance of the invariant verification problem. If the observation predicate r contains only latched variables —that is, $free(r) \subseteq latchX_P$— then the invariant-verification question (P, r) reduces to the reachability question $(G_P^L, [\![\neg r]\!]_P)$. If the observation predicate r refers to some history-free variables, then a transition-reachability question on the latch-reduced transition graph G_P^L needs to be answered. To see this, we make use of the following definitions.

Latch-satisfaction of a state predicate

Let P be a module, and let q be a state predicate for P. The initial state s^L of the latch-reduced transition graph G_P^L latch-satisfies q if there is an initial state s of P such that $s^L = s[latchX_P]$ and $s \models q$. The transition (s^L, t^L) of G_P^L latch-satisfies q if there is a transition (s, t) of P such that $s^L = s[latchX_P]$ and $t^L = t[latchX_P]$ and $t \models q$.

Exercise 2.19 {P2} [Latch-satisfaction] Consider a propositional module P and a propositional formula q which is a state predicate for P. Implement the following four functions. The function $LatchReducedInit(\{G_P\}_{of})$ returns a queue containing the initial states of the latch-reduced transition graph G_P^L. Given a state s^L of G_P^L, the function $LatchReducedPost(s^L, \{G_P\}_{of})$ returns a queue containing the successors of s^L in the latch-reduced transition graph G_P^L. Given an initial state s^L of G_P^L, the boolean function $InitLatchSat(s^L, \{G_P\}_{of}, q)$ checks if s^L latch-satisfies q. Given a transition (s^L, t^L) of G_P^L, the boolean function $TransLatchSat(s^L, t^L, \{G_P\}_{of}, q)$ checks if (s^L, t^L) latch-satisfies q. What are the running times of your algorithms in terms of the size of the input (P, q)? ■

Proposition 2.4 [Latch-reduced invariant verification] The answer to the invariant-verification question (P, r) is No iff there is an initialized trajectory $\overline{s}^L_{1..m}$ of the latch-reduced transition graph G_P^L such that either $m = 1$ and the initial state s_1^L latch-satisfies r, or $m > 1$ and the transition (s^L_{m-1}, s^L_m) latch-satisfies r.

Exercise 2.20 {T2} [Proof of Proposition 2.4] Consider a module P and three states s^L, t^L, and u^L of the latch-reduced transition graph G_P^L. Prove that if $s^L \rightarrow_P^L t^L \rightarrow_P^L u^L$, then there are three states s, t, and u of P such that $s^L = s[latchX_P]$ and $t^L = t[latchX_P]$ and $u^L = u[latchX_P]$ and $s \rightarrow_P t \rightarrow_P u$. Proposition 2.4 follows. ■

Proposition 2.4 gives a recipe for invariant verification using the latch-reduced transition graph instead of the full transition graph of a module: the invariant verifier Algorithm 2.4 can call the reachability checker Algorithm 2.3 on the latch-reduced transition graph G_P^L of the input module P, provided that the membership test $IsMember(s, \sigma^T)$ for the target region is replaced by appropriate applications of the boolean functions *InitLatchSat* and *TransLatchSat*. The space savings may be substantial, as the type **state** needs to store only the values for the latched variables of P.

Exercise 2.21 {P3} [On-the-fly, latch-reduced transition-invariant verification] Since the latch-satisfaction of a state predicate is based on transitions, rather than states, latch reduction lends itself naturally to checking transition invariants (cf. Exercise 2.6). Give a detailed algorithm for solving the propositional transition-invariant verification problem, using both on-the-fly and latch-reduction techniques. Use the functions *LatchReducedInit* and *LatchReducedPost* from Exercise 2.20, and modify *TransLatchSat* for transition invariants. Choose either the state-level or the bit-level model. In either case, for every propositional instance (P, r') of the transition-invariant problem, you should aim for the time complexity $O(4^k \cdot (|P| + |r'|))$ and the space complexity $O(2^k + |P| + |r'|)$, where k is the number of latched variables of the input module P. ■

2.4 State Explosion*

The exponential difference between reachability checking for transition graphs (Theorem 2.1) and invariant verification for reactive modules (Theorem 2.2) is intrinsic and, in general, cannot be avoided: in this section, we formally prove that the complexity class of the propositional invariant-verification problem is Pspace, and therefore, in absence of any major breakthroughs in complexity theory, the invariant-verification problem cannot be solved efficiently. The stark contrast to the complexity class of the reachability problem, Nlogspace, is caused by the fact that a module provides an exponentially more succinct description of a transition graph than an enumerative graph representation. This phenomenon is called state explosion. The source of state explosion is the number of module variables: for a module P, the number of states of the transition graph G_P grows exponentially with the number of variables of P. State explosion is the single most important obstacle to verification practice, for two reasons. First, state explosion does not arise from any peculiarities of our modeling framework —any discrete system with k boolean variables gives rise to 2^k states— and therefore is present in all modeling frameworks. Second, state explosion arises in invariant verification, which asks the simplest kind of global questions about the dynamics of a discrete system, and therefore is present for all verification questions. This prominence has thrust state explosion into the center of verification research. At the same time, the results of this section show that all approaches to alleviate the state-explosion problem are ultimately doomed to be heuristics that work well in certain limited cases. Several of the next chapters will present such heuristics which have proved useful in practice.

2.4.1 Hardness of Invariant Verification

We first prove that in the propositional case —where all variables are boolean— the invariant-verification problem is hard for Pspace, and then, that in the general case —specifically, in the presence of integer variables— the invariant-verification problem is undecidable. Both proofs are similar, in that reactive modules are used to simulate Turing machines: polynomial-space Turing machines in the boolean case; arbitrary Turing machines in the integer case.

Pspace-hardness of propositional invariant verification

The Pspace-hardness of propositional invariant verification follows from the fact that with a polynomial number of boolean variables, one can simulate the behavior of a Turing machine that visits a polynomial number of tape cells.

Theorem 2.3 [Hardness of propositional invariant verification] The propositional invariant-verification problem is Pspace-hard.

Proof. We polynomial-time reduce the acceptance problem for polynomial-space Turing machines to the propositional invariant-verification problem. We are given a deterministic Turing machine M that accepts or rejects every input in polynomial space, and we are given an input word $\overline{a}$ for M. We need to construct, in time polynomial in the specification of M and the length of $\overline{a}$, a propositional module $P_{M,\overline{a}}$ and an observation predicate $r_{M,\overline{a}}$ so that the invariant-verification problem $(P_{M,\overline{a}}, r_{M,\overline{a}})$ has the answer Yes iff the Turing machine M accepts the input $\overline{a}$. Since determining whether or not a polynomial-space Turing machine accepts an input is, by definition, Pspace-hard, it follows that the propositional invariant-verification problem is also Pspace-hard.

As the given Turing machine M uses only polynomial space, there is a polynomial function $p(\cdot)$ so that M accepts or rejects every input of length i by visiting at most $p(i)$ tape cells. Let A be the tape alphabet of M, containing the blank letter, and let Q be the set of control modes of M, containing the initial mode q_I, the accepting mode q_A, and the rejecting mode q_R. Let n be the length of the given input word $\overline{a}$. The Turing machine starts in the control mode q_I, its read head at the first tape cell, with the first n tape cells containing the input $\overline{a}$, and the remaining $p(n) - n$ tape cells containing blanks. The Turing machine accepts the input by entering the control mode q_A, and it rejects the input by entering the control mode q_R. Let t be the number of computation steps that M needs for accepting or rejecting the input $\overline{a}$.

We construct a finite, closed, deterministic module $P_{M,\overline{a}}$, which, for simplicity, is not necessarily propositional; the task of turning $P_{M,\overline{a}}$ into an appropriate propositional module is left to the reader (Exercise 2.22). The module $P_{M,\overline{a}}$ has $p(n)$ variables, $x_1, \ldots, x_{p(n)}$, each of the finite type A, and $p(n)$ variables, $y_1, \ldots, y_{p(n)}$, each of the finite type $Q \cup \{\perp\}$. The value of x_i indicates the contents of the i-th tape cell. If the read head of M is located at the i-th tape cell, then the value of y_i indicates the control mode of M; otherwise y_i has the value $\perp$. In this way, every state s of the module $P_{M,\overline{a}}$, such that $s(y_i)$ belongs to Q for precisely one i between 1 and $p(n)$, encodes a configuration of the Turing machine M. All variables are interface variables and are controlled by a single atom, $U_{M,\overline{a}}$. The initial and update commands of $U_{M,\overline{a}}$ ensure that the unique trajectory of $P_{M,\overline{a}}$ of length t encodes the computation of M on input $\overline{a}$. The initial command of $U_{M,\overline{a}}$ contains a single guard assignment, which assigns the input letter a_i to x_i for all $1 \leq i \leq n$, assigns the blank letter to x_i for all $n < i \leq p(n)$, assigns the initial mode q_I to y_1, and assigns $\perp$ to y_i for all $1 < i \leq p(n)$. Then, the unique initial state of $P_{M,\overline{a}}$ encodes the initial configuration of M. The Turing machine M is specified by a set of transition rules, which are tuples in $(Q \times A) \times (Q \times A \times \{\mathit{left}, \mathit{right}\})$. For example, the transition rule $((q, a), (q', a', \mathit{right}))$ specifies that "if the control mode is q and the tape letter at the read head is a, then switch the control mode to q', write letter a' onto the tape, and move the read head one tape cell to the right." For each transition rule of M, the update command of $U_{M,\overline{a}}$ contains $p(n) - 1$ guarded assignments, which simulate the effect of the rule. For example, for the transition rule $((q, a), (q', a', \mathit{right}))$, for each $1 \leq i < p(n)$, the update command contains the guarded assignment

$$x_i = a \wedge y_i = q \ \rightarrow \ x'_i := a';\ y'_i := \perp;\ y'_{i+1} := q'.$$

Then, for all $j \leq t$, the unique initialized trajectory of $P_{M,\overline{a}}$ of length j encodes the first j computation steps of M on input $\overline{a}$. Consequently, the observation predicate

$$r_{M,\overline{a}}: \quad (\wedge\ 1 \leq i \leq p(n) \mid y_i \neq q_R)$$

is an invariant of the module $P_{M,\overline{a}}$ iff the Turing machine M accepts the input $\overline{a}$. If the specification of M contains $|M|$ symbols, then the textual description of $P_{M,\overline{a}}$ has $O(|M| \cdot p(n))$ symbols, and the predicate $r_{M,\overline{a}}$ consists of $O(p(n))$ symbols; so both $P_{M,\overline{a}}$ and $r_{M,\overline{a}}$ can be constructed in time polynomial in the size of $(M, \overline{a})$. ■

Remark 2.19 [Hardness of propositional invariant verification] The module $P_{M,\overline{a}}$ constructed in the proof of Theorem 2.3 contains a single atom that controls a polynomial number of variables. Instead, we can construct a module that has polynomial number of atoms, each of which controls a single variable, reads a constant number of variables, awaits none, and has initial and update commands of constant size. Thus, the state explosion is independent of the number or complexity of atoms; it occurs when all module variables are controlled by a single atom, and when each atom controls a single variable. Also note that, in the proof of Theorem 2.3, nondeterminism plays no role in establishing Pspace-hardness. ■

Exercise 2.22 {`T3`} [Hardness of propositional invariant verification] (a) Complete the proof of Theorem 2.3 by turning the module $P_{M,\overline{a}}$ and the predicate $r_{M,\overline{a}}$, in polynomial time, into a propositional module and a propositional formula with the appropriate properties. (b) The module $P_{M,\overline{a}}$ used in the proof of Theorem 2.3 is synchronous. Prove that the invariant-verification problem is Pspace-hard even if the problem instances are restricted to propositional modules with (1) a single atom which is a speed-independent process, and (2) multiple atoms each of which is a speed-independent process controlling a single variable. ■

Undecidability of invariant verification with counters

For infinite modules, the invariant-verification question can be algorithmically undecidable. To see this, we define a class of reactive modules which make use of nonnegative integer variables in a very restricted way. A counter is a variable of type $\mathbb{N}$ which can be initialized to 0 or 1, tested for 0, incremented, and decremented. If all variables of a module are counters, then the module is said to be a counter module. Thus, the counter modules are a proper subset of the integer modules with addition from Exercise 2.17.

Counter module

A counter module is a reactive module P such that (1) all module variables of P are of type $\mathbb{N}$, (2) every guard that appears in the initial and update commands of P is a finite (possibly empty) conjunction of predicates of the form $x = 0$ and $x > 0$, (3) every assignment that appears in the initial commands of P is either 0 or 1, and (4) every assignment that appears in the update commands of P has the form $x + 1$ or $x \dot{-} 1$. The variables of a counter module are called counters. An instance (P, r) of the invariant-verification problem is a counter instance if P is a counter module and the predicate r has the form $x = 0$. The instances of the counter invariant-verification problem are the counter instances of the invariant-verification problem.

A classical counter machine is a discrete, deterministic system with a finite number of control modes and a finite number of counters. Since each control mode q can be replaced by a counter, whose value is 1 iff the control is in the mode q, and otherwise 0, every counter machine can be simulated by a closed, deterministic counter module. Since every Turing machine can, in turn, be simulated, by a counter machine —in fact, two counters suffice to encode the contents of an unbounded number of tape cells— it follows that the reachability problem for counter machines, which asks if a given counter machine ever enters a given control mode, and therefore also the counter invariant-verification problem, is undecidable.

Theorem 2.4 [Hardness of counter invariant verification] The counter invariant-verification problem is undecidable.

2.4.2 Complexity of Invariant Verification

The Pspace lower bound for propositional invariant verification (Theorem 2.3) can be tightly matched by an upper bound. However, Algorithm 2.3 uses $\Omega(2^k)$ space for solving an invariant-verification question with k variables, even if on-the-fly methods are employed, and independent of state-level vs. bit-level analysis, because the region σ^R of explored states may contain up to 2^k states. A different approach is needed if we wish to use only space polynomial in k. We now present such an algorithm and give a detailed, bit-level space analysis. (Bit-level analysis is, strictly speaking, not necessary, as the state-level and bit-level space requirements of any algorithm can differ at most by a polynomial factor of k.)

Let P be a propositional module with k variables, and let r be a propositional formula that is an observation predicate for P. Since P has 2^k states, a state s of P is reachable iff there is an initialized trajectory with sink s and length at

Algorithm 2.5 [Pspace algorithm for invariant verification] (schema)

Input: a propositional module P, and a propositional formula r.
Output: the answer to the instance (P, r) of the invariant-verification problem.

Let k be the number of module variables of P;
foreach $s, t \in \Sigma_P$ do
  if $s \in \sigma_P^I$ and not $t \models r$ then
    if $PSpaceSearch(s, t, 2^k)$ then return Yes fi
    fi
  od;
return No.

function $PSpaceSearch(s, t, i)$: $\mathbb{B}$
  if $s = t$ then return *true* fi;
  if $i > 1$ and $s \rightarrow_P t$ then return *true* fi;
  if $i > 2$ then
    foreach $u \in \Sigma_P$ do
      if $PSpaceSearch(s, u, \lceil i/2 \rceil)$ and $PSpaceSearch(u, t, \lceil i/2 \rceil)$ then
        return *true*
        fi
      od
    fi;
  return *false*.

most 2^k. This suggests Algorithm 2.5, which performs a binary search to check reachability on the transition graph G_P. Given two states s and t and a positive integer i, the boolean function *PSpaceSearch*(s, t, i) returns *true* iff there is a trajectory with source s and sink t of length at most i. The function is computed recursively by attempting to find a state u at the midpoint of the trajectory. Since every state of P is a bitvector of length k, all pairs of states can be enumerated one after the other in $O(k)$ space. By Lemma 2.6, given two states s and t, it can be determined in $O(|P| + |r|)$ time, and therefore linear space, if s is initial, if t satisfies r, and if t is a successor of s. In Algorithm 2.5, the space used by the first recursive call of the function *PSpaceSearch* can be reused by the second recursive call. Hence, the total space used by Algorithm 2.5 is $O(k \cdot d + |P| + |r|)$, where d is the depth of the recursion. Since each recursive call searches for a trajectory of half the length, starting from length 2^k, the depth of the recursion is bounded by k. Thus, Algorithm 2.5 can be implemented in $O(k^2 + |P| + |r|)$ space, which is quadratic in the size of the input. This establishes that the propositional invariant-verification problem can be solved in Pspace.

Theorem 2.5 [Complexity of invariant verification] The propositional invariant-verification problem is Pspace-complete.

Remark 2.20 [Depth-first search versus Pspace search] Algorithm 2.5 has an $\Omega(8^k)$ time complexity, even in the state-level model, and thus pays a price in running time for achieving polynomial space. In practice, one always prefers search algorithms whose running time is at worst proportional to the number of transitions (say, $O(4^k \cdot (|P| + |r|))$ for propositional instances of the invariant-verification problem with k variables), or better yet, proportional to the number of states and reachable transitions (as is the case for on-the-fly depth-first-search). ■

Remark 2.21 [Nondeterministic complexity of invariant verification and reachability] An alternative, conceptually simpler proof that the propositional invariant-verification problem belongs to Pspace can be based on the knowledge that deterministic and nondeterministic polynomial space coincide (i.e., every nondeterministic polynomial-space Turing machine can be simulated, using only polynomial space, by a deterministic Turing machine). Hence it suffices to give a nondeterministic approach for solving the propositional invariant-verification problem in polynomial space. Such an approach is outlined in Algorithm 2.6. The nondeterministic algorithm solves propositional invariant-verification questions with k variables using only the local variables s, m, i, and t, in $\Theta(k + |P| + |r|)$ space. Essentially the same nondeterministic algorithm, applied to inputs of the form (G, σ^T), where G is a transition graph and σ^T is a region of G, shows that the reachability problem belongs to Nlogspace (i.e., the algorithm uses only logarithmic space in addition to the space occupied by the input). ■

Algorithm 2.6 [Npspace schema for invariant verification]

Input: a propositional module P, and a propositional formula r.
Output: one of the nondeterministic runs returns Yes iff the instance (P, r) of the invariant-verification problem has the answer Yes.

```
Let k be the number of module variables of P;
Choose an arbitrary state s ∈ Σ_P;
if not s ∈ σ^I_P then return No fi;
Choose an arbitrary nonnegative integer m between 0 and 2^k;
for i := 1 to m do
  Choose an arbitrary state t ∈ Σ_P;
  if not s →_P t then return No fi;
  s := t
  od;
if not s ⊨ r then return No fi;
return Yes.
```

2.5 Compositional Reasoning

Complex systems are often built from parts of small or moderate complexity. For example, circuits are built from individual gates and memory cells. In such a setting, state explosion is caused by the parallel composition of many modules, each with a small number of variables. When possible, we want to make use of the structure inherent in such designs for verification purposes. In particular, the state-explosion problem may be avoided if an invariant of a compound module can be derived from invariants of the component modules.

2.5.1 Composing Invariants

A divide-and-conquer approach to verification attempts to reduce a verification task about a complex system to subtasks about subsystems of manageable complexity. If the reduction follows the operators that are used in the construction of the complex system, then the divide-and-conquer approach is called compositional reasoning. Complex reactive modules are built from simple modules using the three operations of parallel composition, variable renaming, and variable hiding. Hence, for compositional invariant verification, we need to know how invariants distribute over the three module operations.

Proposition 2.5 [Compositionality of invariants] If the observation predicate r is an invariant of the module P, then the following three statements hold.

Parallel composition For every module Q that is compatible with P, the observation predicate r is an invariant of the compound module $P \| Q$.

Variable renaming For every variable renaming ρ for the module variables of P, the renamed observation predicate $r[\rho]$ is an invariant of the renamed module $P[\rho]$.

Variable hiding For every interface variable x of P, the observation predicate $(\exists x \mid r)$ is an invariant of the module **hide** x **in** P. In particular, if x does not occur freely in r, then r is an invariant of **hide** x **in** P.

Proof. The first part of Proposition 2.5 follows from Proposition 2.2. The second and third parts are immediate. ■

The compositionality and monotonicity of invariants (the first part of Proposition 2.5 and Remark 2.9) suggest the following verification strategy, called compositional invariant verification:

> Let P and Q be two compatible modules, and let r be an observation predicate for the compound module $P \| Q$. In order to show that r is an invariant of $P \| Q$, it suffices to find an observation predicate p for P, and an observation predicate q for Q, such that (1) p is an invariant of P, (2) q is an invariant of Q, and (3) the implication $p \wedge q \rightarrow r$ is valid.

The local invariants p and q represent the guarantees that the components P and Q make as to jointly maintain the global invariant r. Compositional invariant verification is usually beneficial if the state spaces of P and Q are smaller than the state space of the compound module $P \| Q$, which is the typical scenario. It should be noted, however, that the compound module may have fewer reachable states than either component module, in which case decomposition does not achieve the desired effect. This happens when the two components are tighly coupled, strongly restraining each others behaviors, and have few private variables.

Example 2.11 [Compositional verification of railroad control] Let us revisit Example 2.8 and prove that the railroad controller *Controller2* from Figure 2.9 enforces the train-safety requirement that in all rounds, at most one train is on the bridge. More precisely, we wish to establish that the observation predicate

$$r^{safe}: \quad \neg(pc_W = bridge \wedge pc_E = bridge)$$

is an invariant of the module

module *RailroadSystem2* **is**
  hide $arrive_W, arrive_E, leave_W, leave_E$ **in**
    $\|$ $Train_W$
    $\|$ $Train_E$
    $\|$ *Controller2*.

To decompose the verification problem, we observe that the controller ensures that (1) the train traveling clockwise is allowed to proceed onto the bridge only when the western signal is green and the eastern signal is red, and symmetrically, (2) the train traveling counterclockwise is allowed to proceed onto the bridge only when the eastern signal is green and the western signal is red. The conjunction of these two assertions entails the train-safety requirement. The formal argument proceeds in six steps:

1. We establish that

$$r_W^{safe}: \quad pc_W = bridge \rightarrow (signal_W = green \wedge signal_E = red)$$

   is an invariant of the module $Train_W \,\|\, Controller2$.

2. By the second part of Proposition 2.5 (variable renaming in invariants), we deduce that the renamed predicate

$$r_E^{safe}: \quad pc_E = bridge \rightarrow (signal_E = green \wedge signal_W = red)$$

   is an invariant of the renamed module $Train_E \,\|\, Controller2$.

3. By the first part of Proposition 2.5 (compositionality of invariants), we deduce that both r_W^{safe} and r_E^{safe} are invariants of the compound module $Train_W \,\|\, Train_E \,\|\, Controller2$.

4. By the second part of Remark 2.9 (monotonicity of invariants), we deduce that the conjunction $r_W^{safe} \wedge r_E^{safe}$ is an invariant of the module $Train_W \,\|\, Train_E \,\|\, Controller2$.

5. Since the implication $r_W^{safe} \wedge r_E^{safe} \rightarrow r^{safe}$ is valid, by the first part of Remark 2.9 (monotonicity of invariants), we deduce that r^{safe} is also an invariant of $Train_W \,\|\, Train_E \,\|\, Controller2$.

6. Finally, by the third part of Proposition 2.5 (variable hiding in invariants), we conclude r_W^{safe} is an invariant of the module *RailroadSystem2*.

Only the first step requires state-space exploration, namely, the solution of a reachability problem on the latch-reduced transition graph of the module $Train_W \,\|\, Controller2$. The savings are easy to compute: the latch-reduced transition graph of $Train_W \,\|\, Controller2$ has 48 states; the latch-reduced transition graph of entire system *RailroadSystem2* has 144 states. ■

Exercise 2.23 {P2} [Composing invariants] Consider the two-bit arbiter module *BinArbiter* from Figure 2.15. When the control input *akout* is high, then one of the two data inputs in_1 and in_2 is relayed to the data output *out*. If in_i is relayed, $i = 1, 2$, then the corresponding control output $akin_i$ is set to 1, and

$$
\begin{array}{l}
\textbf{module } \mathit{BinArbiter} \textbf{ is} \\
\quad \textbf{private } \mathit{turn} \colon \{1,2\} \\
\quad \textbf{interface } \mathit{akin}_1, \mathit{akin}_2, \mathit{out} \colon \mathbb{B} \\
\quad \textbf{external } \mathit{in}_1, \mathit{in}_2, \mathit{akout} \colon \mathbb{B} \\
\quad \textbf{atom controls } \mathit{turn} \textbf{ reads } \mathit{turn} \textbf{ awaits } \mathit{akout} \\
\qquad \textbf{init} \\
\qquad\quad [\!] \ \mathit{true} \to \mathit{turn}' := 1 \\
\qquad\quad [\!] \ \mathit{true} \to \mathit{turn}' := 2 \\
\qquad \textbf{update} \\
\qquad\quad [\!] \ \mathit{akout}' \wedge \mathit{turn} = 1 \to \mathit{turn}' := 2 \\
\qquad\quad [\!] \ \mathit{akout}' \wedge \mathit{turn} = 2 \to \mathit{turn}' := 1 \\
\quad \textbf{atom controls } \mathit{akin}_1, \mathit{akin}_2, \mathit{out} \textbf{ awaits } \mathit{turn}, \mathit{in}_1, \mathit{in}_2, \mathit{akout} \\
\qquad \textbf{initupdate} \\
\qquad\quad [\!] \ \mathit{turn}' = 1 \to \mathit{akin}_1' := \mathit{akout}';\ \mathit{akin}_2' := 0;\ \mathit{out}' := \mathit{in}_1' \\
\qquad\quad [\!] \ \mathit{turn}' = 2 \to \mathit{akin}_1' := 0;\ \mathit{akin}_2' := \mathit{akout}';\ \mathit{out}' := \mathit{in}_2'
\end{array}
$$

Figure 2.15: Two-bit round-robin arbiter

the other control output, akin_{3-i}, is set to 0. When the control input akout is low, then both control outputs akin_1 and akin_2 are set to 0, as to indicate that none of the data inputs is relayed. The variable turn controls which data input is relayed and alternates the two choices. (a) Using three two-bit arbiters, we can build the four-bit round-robin arbiter

$$
\begin{array}{l}
\textbf{module } \mathit{QuadArbiter} \textbf{ is} \\
\quad \textbf{hide } \mathit{in}_{12}, \mathit{in}_{34}, \mathit{akin}_{12}, \mathit{akin}_{34} \textbf{ in} \\
\qquad \| \ \mathit{BinArbiter}[\mathit{in}_1, \mathit{in}_2, \mathit{akin}_1, \mathit{akin}_2 := \mathit{in}_{12}, \mathit{in}_{34}, \mathit{akin}_{12}, \mathit{akin}_{34}] \\
\qquad \| \ \mathit{BinArbiter}[\mathit{out}, \mathit{akout} := \mathit{in}_{12}, \mathit{akin}_{12}] \\
\qquad \| \ \mathit{BinArbiter}[\mathit{in}_1, \mathit{in}_2, \mathit{akin}_1, \mathit{akin}_2, \mathit{out}, \mathit{akout} := \mathit{in}_3, \mathit{in}_4, \mathit{akin}_3, \mathit{akin}_4, \mathit{in}_{34}, \mathit{akin}_{34}]
\end{array}
$$

whose abstract block diagram is shown in Figure 2.16. Why is $\mathit{QuadArbiter}$ called a round-robin arbiter? Prove compositionally that the two observation predicates

$$
\begin{array}{ll}
p_2\colon & (\forall\, 1 \le i \le 4 \mid \mathit{akin}_i \to \mathit{out} = \mathit{in}_i) \\
q_2\colon & \mathit{akout} \leftrightarrow (\exists\, 1 \le i \le 4 \mid \mathit{akin}_i)
\end{array}
$$

are invariants of the module $\mathit{QuadArbiter}$. First, find suitable invariants p_1 and q_1 for the two-bit arbiter $\mathit{BinArbiter}$ and prove them by inspecting the latch-reduced transition graph (Proposition 2.4). Then, use compositional reasoning (Proposition 2.5 and Remark 2.9) to establish the invariants p_2 and q_2 of the four-bit arbiter $\mathit{QuadArbiter}$. (b) Compositional reasoning permits us to prove invariants for entire module classes, not only individual modules. An example

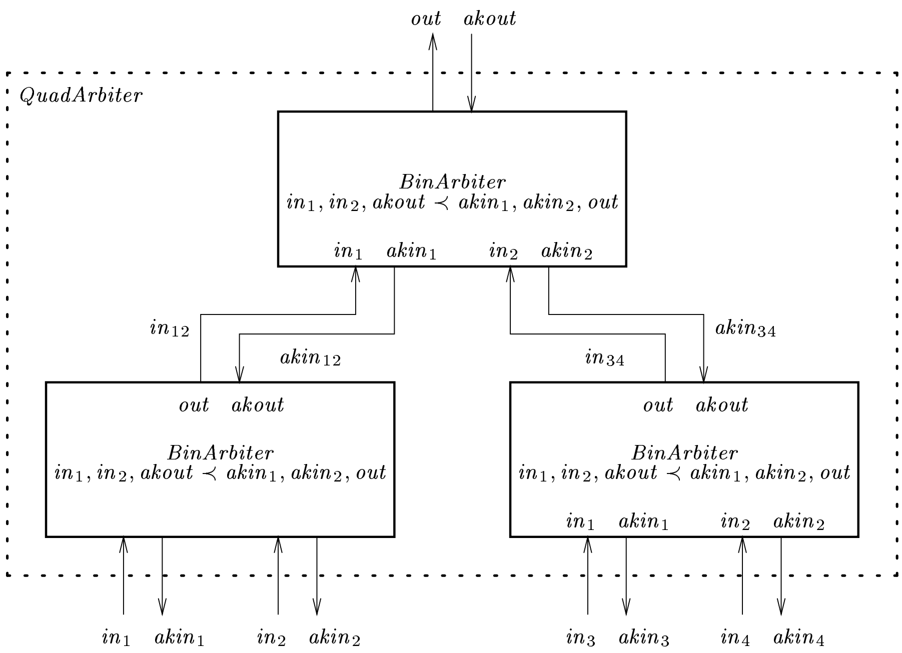

Figure 2.16: Abstract block diagram for four-bit round-robin arbiter

of this is the class of all 2^k-bit round-robin arbiters, for positive integers k, which are built by connecting $2^k - 1$ two-bit arbiters to form a binary tree of height k. The construction of the resulting module schema *TreeArbiter* is shown in Figure 2.17. Use compositional reasoning to derive, for all $k \geq 1$, the invariants

$$
\begin{array}{ll}
p_k\text{:} & (\forall\ 1 \leq i \leq 2^k \mid akin_i\ \rightarrow\ out = in_i) \\
q_k\text{:} & akout\ \leftrightarrow\ (\exists\ 1 \leq i \leq 2^k \mid akin_i)
\end{array}
$$

of the 2^k-bit arbiter *TreeArbiter* from the invariants p_1 and q_1 of the two-bit arbiter *BinArbiter*. The integer k is a parameter that occurs in the module definition, the invariant definition, and the derivation. ■

2.5.2 Assuming Invariants

The compositional approach that was advocated in the previous section has limited applicability. Suppose that the predicate r is an invariant of the compound module $P \| Q$. A decomposition of the global invariant r into a local invariant p of P and a local invariant q of Q, which together imply r, may not be possible. Rather, it is often necessary to make certain assumptions about

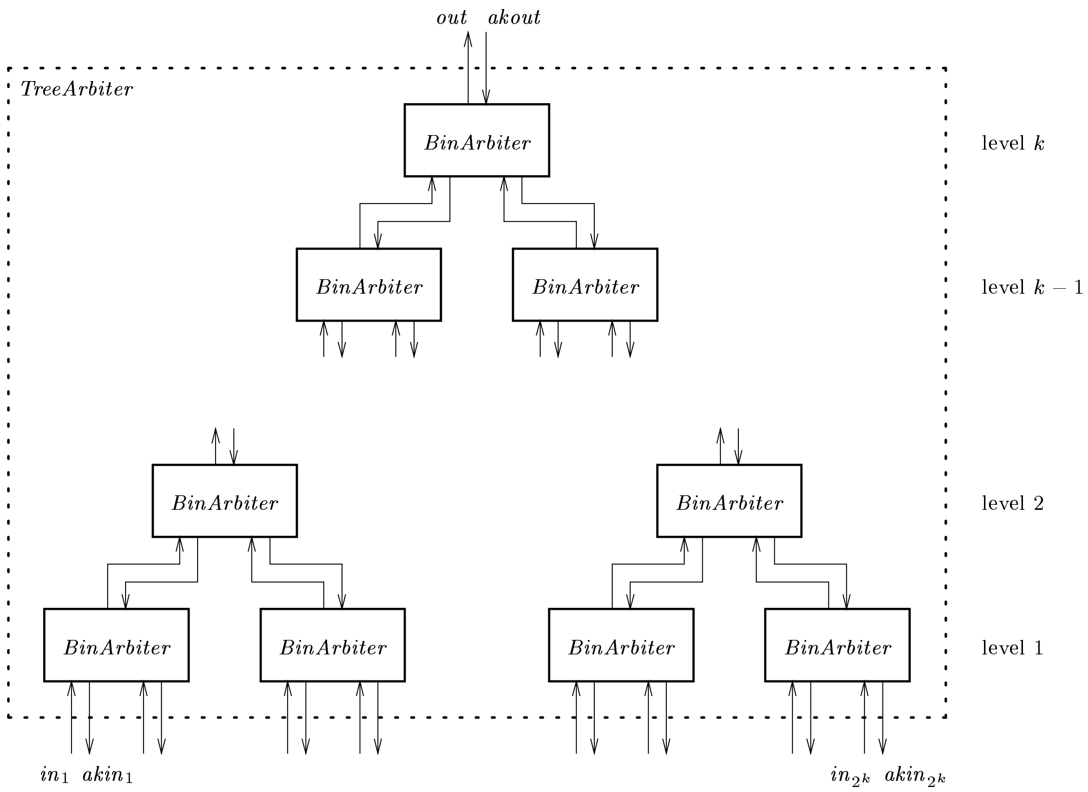

Figure 2.17: Schematic construction of 2^k-bit round-robin arbiter

the environment of the component P in order for P to do its share in ensuring the global invariant r by maintaining the local invariant p. The assumptions on the environment of P need then to be discharged against Q. Symmetrically, Q may contribute to r by maintaining q only if its environment meets assumptions that can be discharged against P. The situation becomes apparently cyclic if the environment invariant q is the very assumption necessary for establishing that p is invariant with respect to P, and the environment invariant p is the assumption needed to establish the invariance of q with respect to Q.

For a concrete illustration, Figure 2.18 presents a series of small examples. First, consider the two modules P_1, controlling x, and Q_1, controlling y. We want to prove compositionally the invariant $y = 0$ of the compound module $P_1 \| Q_1$. This follows from the invariant $x = 0$ of P_1 and the invariant $y = x$ of Q_1. Second, consider the two modules P_2 and Q_2. In this case, the invariant $y = 0$ of the module $P_2 \| Q_2$ cannot be established compositionally, because the truth of $y = 0$ in one round depends on the truth of $x = 0$ in the previous round. Still,

module P_1 **is**
  interface $x : \mathbb{B}$
  external $y : \mathbb{B}$
  atom controls x **reads** x
    init
      $[\!]$ *true* $\rightarrow$ $x' := 0$
    update
      $[\!]$ *true* $\rightarrow$ $x' := x$

module Q_1 **is**
  interface $y : \mathbb{B}$
  external $x : \mathbb{B}$
  atom controls y **awaits** x
    initupdate
      $[\!]$ *true* $\rightarrow$ $y' := x'$

module P_2 **is**
  interface $x : \mathbb{B}$
  external $y : \mathbb{B}$
  atom controls x **reads** x
    init
      $[\!]$ *true* $\rightarrow$ $x' := 0$
    update
      $[\!]$ *true* $\rightarrow$ $x' := x$

module Q_2 **is**
  interface $y : \mathbb{B}$
  external $x : \mathbb{B}$
  atom controls y **reads** x
    init
      $[\!]$ *true* $\rightarrow$ $y' := 0$
    update
      $[\!]$ *true* $\rightarrow$ $y' := x$

module P_3 **is**
  interface $x : \mathbb{B}$
  external $y : \mathbb{B}$
  atom controls x **reads** y
    init
      $[\!]$ *true* $\rightarrow$ $x' := 0$
    update
      $[\!]$ *true* $\rightarrow$ $x' := y$

module Q_3 **is**
  interface $y : \mathbb{B}$
  external $x : \mathbb{B}$
  atom controls y **reads** x
    init
      $[\!]$ *true* $\rightarrow$ $y' := 0$
    update
      $[\!]$ *true* $\rightarrow$ $y' := x$

Figure 2.18: Three forms of collaboration to maintain the invariant $y = 0$

the problem can be solved with the help of transition invariants (Exercise 2.6). The transition invariant $y' = 0$ of $P_2 \,\|\, Q_2$ follows from the invariant $x = 0$ of P_1 and the transition invariant $y' = x$ of Q_1. Together with the fact that initially $y = 0$, this establishes the invariant $y = 0$ of $P_2 \,\|\, Q_2$. Finally, consider the two modules P_3 and Q_3. In this case, the invariant $y' = 0$ of the module $P_3 \,\|\, Q_3$ cannot be established from the transition invariants $y' = x$ of P_3 and $x' = y$ of Q_3. In order for Q_3 to guarantee the desired invariant $y = 0$, we need to assume that the environment of Q_3 maintains the invariant $x = 0$. Symmetrically, P_3 guarantees the invariant $x = 0$ only under the assumption that, in turn, the environment of P_3 keeps $y = 0$ invariant. Then, induction on the length of the initialized trajectories of the compound module $P_3 \,\|\, Q_3$ resolves the cyclic interdependence between assumptions and guarantees and establishes the global invariant $x = 0 \wedge y = 0$. This kind of compositional proof strategy is called assume-guarantee reasoning. In this chapter, we restrict ourselves to both assumptions and guarantees which are invariants.

Let P be a module, and let r be an external predicate for P. The assumption that the environment of P maintains the invariant r can be represented by composing P with a simple module whose only purpose is to keep r invariant and, while doing so, permitting as many initialized traces as possible. The most permissive module that ensures the invariance of r is called the r-assertion module.

Assertion module

Let r be a satisfiable boolean expression whose free variables have finite types. We define the r-assertion module $Assert(r)$ as follows. The sets of private and external variables of $Assert(r)$ are empty; the set of interface variables of $Assert(r)$ is the set $free(r)$ of variables which occur freely in r. The module $Assert(r)$ has a single atom, which controls all variables in $free(r)$, neither reads nor awaits any variables, and has identical initial and update commands: for every valuation s of the variables in $free(r)$ which satisfies r, the initial and update commands contain a guarded assignment with the guard *true* and for each variable $x \in free(r)$, the assignment $s(x)$.

Remark 2.22 [Assertion module] The boolean expression r is an interface predicate and an invariant of the r-assertion module $Assert(r)$. In fact, r is the strongest invariant of $Assert(r)$; that is, for every invariant q of $Assert(r)$, the implication $r \to q$ is valid. All variables of the r-assertion module $Assert(r)$ are history-free, and therefore, the latch-reduced transition graph of $Assert(r)$ has a single state. ■

The following theorem formalizes our contention that if (1) the module P guarantees the invariant p assuming the environment maintains the external predicate q invariant, and (2) the module Q guarantees the invariant q assuming the environment maintains p invariant, then the compound module $P \| Q$ has both p and q as invariants. As in the third example from Figure 2.18, the proof will proceed by induction on the length of the initialized trajectories of $P \| Q$.

Theorem 2.6 [Assume-guarantee reasoning for invariants] Let P and Q be two compatible modules. Let p be an external predicate for Q, and let q be an external predicate for P, such that all free variables of p and q have finite types. If p is an invariant of $P \| Assert(q)$, and q is an invariant of $Assert(p) \| Q$, then $p \wedge q$ is an invariant of $P \| Q$.

Proof. Consider two compatible modules P and Q, an external predicate p for Q, and an external predicate q for P. The free variables of p and q have finite types, so that the assertion modules $Assert(p)$ and $Assert(q)$ are well-defined. Assume that p is an invariant of $P \| Assert(q)$, and q is an invariant of $Assert(p) \| Q$; that is, for every initialized trajectory $\overline{s}$ of $P \| Assert(q)$, the projection $\overline{s}[\mathsf{extl}X_P]$ is an initialized trajectory of $Assert(p)$, and for every initialized trajectory $\overline{s}$ of $Assert(p) \| Q$, the projection $\overline{s}[\mathsf{extl}X_Q]$ is an initialized trajectory of $Assert(q)$. We show that for every initialized trajectory $\overline{s}$ of $P \| Q$, the projection $\overline{s}[X_P]$ is an initialized trajectory of $P \| Assert(q)$, and the projection $\overline{s}[X_Q]$ is an initialized trajectory of $Assert(p) \| Q$. It follows that $p \wedge q$ is an invariant of $P \| Q$.

We need to define some additional concepts. Given a module R, a set $X \subseteq X_R$ of module variables is await-closed for R if for all variables x and y of R, if $y \prec_R x$ and $y \in X$, then $x \in X$. For an await-closed set X, the pair $(\overline{s}, t)$ consisting of an initialized trajectory $\overline{s}$ of R and a valuation t for X is an X-partial trajectory of R if there exists a state u of R such that (1) $u[X] = t$, and (2) $\overline{s}u$ is an initialized trajectory of R. Thus, partial trajectories are obtained by executing several complete rounds followed by a partial round, in which only some of the atoms are executed. The following two crucial facts about partial trajectories follow from the definitions.

(A) The partial trajectories of a compound module are determined by the partial trajectories of the component modules: for every pair R_1 and R_2 of compatible modules, every await-closed set X for $R_1 \| R_2$, every sequence $\overline{s}$ of states of $R_1 \| R_2$, and every valuation t for X, the pair $(\overline{s}, t)$ is an X-partial trajectory of $R_1 \| R_2$ iff $\overline{s}[X_{R_1}]$ is an $(X \cap X_{R_1})$-partial trajectory of R_1 and $\overline{s}[X_{R_2}]$ is an $(X \cap X_{R_2})$-partial trajectory of R_2. This property generalizes Proposition 2.2.

(B) If $(\overline{s}, t)$ is an X-partial trajectory of R, and u is a valuation for a set $Y \subseteq \mathsf{extl}X_R$ of external variables of R which is disjoint from X, then $(\overline{s}, t \cup u)$ is an $(X \cup Y)$-partial trajectory of R. This property is due the nonblocking nature of modules; it generalizes Lemmas 2.1 and 2.2.

Let $X_1, \ldots, X_m$ be a partition of $X_{P\|Q}$ into disjoint subsets such that (1) each X_i either contains only external variables of $P\|Q$, or contains only interface variables of P, or contains only interface variables of Q, and (2) if $y \prec_{P\|Q} x$ and $y \in X_i$, then $x \in X_j$ for some $j < i$. Define $Y_0 = \emptyset$, and for all $0 \leq i < m$, define $Y_{i+1} = Y_i \cup X_i$. Each set Y_i is await-closed for $P\|Q$. For all $0 \leq i \leq m$, let L_i be the set of Y_i-partial trajectories of $P\|Q$, and let $L = (\cup\, 0 \leq i \leq m \mid L_i)$. We define the following order $<$ on the partial trajectories in L: for $i < m$, if $(\overline{s}, t) \in L_i$ and $(\overline{s}, u) \in L_{i+1}$ and $u[Y_i] = t$, then $(\overline{s}, t) < (\overline{s}, u)$; for $i = m$, if $(\overline{s}, t) \in L_i$, then $(\overline{s}, t) < (\overline{s}t, \emptyset)$. Clearly, the order $<$ is well-founded. We prove by well-founded induction with respect to $<$ that for all $0 \leq i \leq m$, if $(\overline{s}, t)$ is a partial trajectory in L_i, then $(\overline{s}[X_P], t[X_P])$ is a $(X_i \cap X_P)$-partial trajectory of $P \,\|\, Assert(q)$, and $(\overline{s}[X_Q], t[X_Q])$ is a $(X_i \cap X_Q)$-partial trajectory of $Assert(p) \,\|\, Q$. In the following, for simplicity, we suppress projections.

Consider $(\overline{s}, \emptyset)$ in L_0. If $\overline{s}$ is the empty trajectory, then $(\overline{s}, \emptyset)$ is a trajectory of all modules. Otherwise, $\overline{s} = \overline{t}u$ for some state sequence $\overline{t}$ and state u of $P\|Q$. Then $(\overline{t}, u)$ is a Y_m-partial trajectory of $P\|Q$, and $(\overline{t}, u) < (\overline{s}, \emptyset)$. By induction hypothesis, $(\overline{t}, u)$ is a Y_m-partial trajectory of both $P \,\|\, Assert(q)$ and $Assert(p) \,\|\, Q$, and hence, $(\overline{s}, \emptyset)$ is a Y_0-partial trajectory of both $P \,\|\, Assert(q)$ and $Assert(p) \,\|\, Q$.

Consider $(\overline{s}, t)$ in L_{i+1} for some $0 \leq i < m$. Let $u = t[Y_i]$. Then $(\overline{s}, u)$ is a Y_i-partial trajectory of $P\|Q$, and $(\overline{s}, u) < (\overline{s}, t)$. By induction hypothesis, $(\overline{s}, u)$ is a Y_i-partial trajectory of both $P \,\|\, Assert(q)$ and $Assert(p) \,\|\, Q$. By fact (A) about partial trajectories, $(\overline{s}, t)$ is a Y_{i+1}-partial trajectory of P and Q, and $(\overline{s}, u)$ is a Y_i-partial trajectory of $Assert(p)$ and $Assert(q)$. It suffices to show that $(\overline{s}, t)$ is a Y_{i+1}-partial trajectory of both $Assert(p)$ and $Assert(q)$. Consider $Y_{i+1} = Y_i \cup X_i$. Without loss of generality, assume that X_i contains only interface variables of P. Then clearly, $(\overline{s}, t)$ is a Y_{i+1}-partial trajectory of $Assert(q)$. By fact (A), $(\overline{s}, t)$ is also a Y_{i+1}-partial trajectory of $P \,\|\, Assert(q)$. By fact (B), there is an initialized trajectory $\overline{s}v$ of $P \,\|\, Assert(q)$ such that $v[Y_{i+1}] = t$. By assumption, $\overline{s}v$ is an initialized trajectory of $Assert(p)$, which implies that $(\overline{s}, t)$ is a Y_{i+1}-partial trajectory of $Assert(p)$. ■

Remark 2.23 [Assume-guarantee reasoning]* For those interested in exactly which of our modeling choices make assume-guarantee reasoning possible, let us review the conditions of Theorem 2.6 by inspecting the proof. The condition that all variables that occur freely in the predicates p and q must have finite types is necessary for the assertion modules $Assert(p)$ and $Assert(p)$ to be well-defined, because the initial and update commands of reactive modules contain only finitely many choices in the form of guarded assignments. This requirement of reactive modules, which is useful for other purposes, is not needed for the soundness of assume-guarantee reasoning. The condition that all variables that occur freely in the predicate q must not be interface variables of the module P is necessary for the assertion module $Assert(q)$ to be compatible with P. In this strong form the condition is not needed for the soundness of assume-guarantee reasoning, as long as it is ensured that the system $P \,\|\, \mathit{Assert}(q)$ which represents the component P together with the invariance assumption q is nonblocking —that is, as long as the assumed invariant q does not prevent the module P from having in every state at least one successor state. A symmetric comment holds for the predicate p and the module Q. ■

Theorem 2.6, in conjunction with the monotonicity of invariants, suggests the following verification strategy, called assume-guarantee invariant verification:

> Let P and Q be two compatible modules, and let r be an observation predicate for the compound module $P\|Q$. In order to show that r is an invariant of $P\|Q$, it suffices to find an external predicate p for Q, and an external predicate q for P, such that (1) p is an invariant of $P \,\|\, \mathit{Assert}(q)$, (2) q is an invariant of $\mathit{Assert}(p) \,\|\, Q$, and (3) the implication $p \wedge q \rightarrow r$ is valid.

While condition (1) holds whenever p is an invariant of P, and condition (2) holds whenever q is an invariant of Q, either converse may fail. Therefore, provided one decomposes the desired invariant r of $P\|Q$ into two parts such that the first part, p, contains no interface variables of Q, and the second part, q, contains no interface variables of P, assume-guarantee invariant verification is more often successful than compositional invariant verification as presented in Section 2.5.1. Furthermore, since the latch-reduced transition graphs of the assertion modules $\mathit{Assert}(p)$ and $\mathit{Assert}(q)$ each contain only a single state, the invariant-verification problem (1) depends on the state space of P, and the invariant-verification problem (2) depends on the state space of Q, but neither involves the state space of the compound module $P\|Q$. In fact, after latch reduction, the reachable states of $P \,\|\, \mathit{Assert}(q)$ are a subset of the reachable states of P, so that the performance of assume-guarantee invariant verification can be no worse, only better, than the performance of compositional invariant verification.

Exercise 2.24 {`P2`} [Conditional invariant verification] An instance (P, q, r) of the conditional invariant-verification problem consists of a module P, an external predicate q for P, and an observation predicate r for P. The answer to the conditional invariant-verification question (P, q, r) is Yes iff r is an invariant of $P \,\|\, \mathit{Assert}(q)$. Note that in the special case that the condition q is the boolean constant *true*, we obtain the standard invariant-verification problem. (a) Define the conditional reachability problem so that conditional invariant-verification questions of the form (P, q, r) can be reduced to conditional reachability questions of the form $(G_P, [\![q]\!]_P, [\![\neg r]\!]_P)$, which do not involve the transition graph of the compound module $P \,\|\, \mathit{Assert}(q)$. (b) Give a depth-first algorithm for conditional reachability checking and analyze its time and space requirements. Your algorithm should perform no worse, and in some cases better, than standard reachability checking. ■

Example 2.12 [] ■

Exercise 2.25 {`T3`} [Compositional reasoning with transition invariants] Generalize Proposition 2.5 and Theorem 2.6 to transition invariants (cf. Exercise 2.6).
■

Index