Chapter 3

Linear-Time Properties

For verification purposes, the transition system model of the system under consideration needs to be accompanied with a specification of the property of interest that is to be verified. This chapter introduces some important, though relatively simple, classes of properties. These properties are formally defined and basic model-checking algorithms are presented to check such properties in an automated manner. This chapter focuses on linear-time behavior and establishes relations between the different classes of properties and trace behavior. Elementary forms of fairness are introduced and compared.

3.1 Deadlock

Sequential programs that are not subject to divergence (i.e., endless loops) have a terminal state, a state without any outgoing transitions. For parallel systems, however, computations typically do not terminate—consider, for instance, the mutual exclusion programs treated so far. In such systems, terminal states are undesirable and mostly represent a design error. Apart from "trivial" design errors where it has been forgotten to indicate certain activities, in most cases such terminal states indicate a *deadlock*. A deadlock occurs if the complete system is in a terminal state, although at least one component is in a (local) nonterminal state. The entire system has thus come to a halt, whereas at least one component has the possibility to continue to operate. A typical deadlock scenario occurs when components mutually wait for each other to progress.

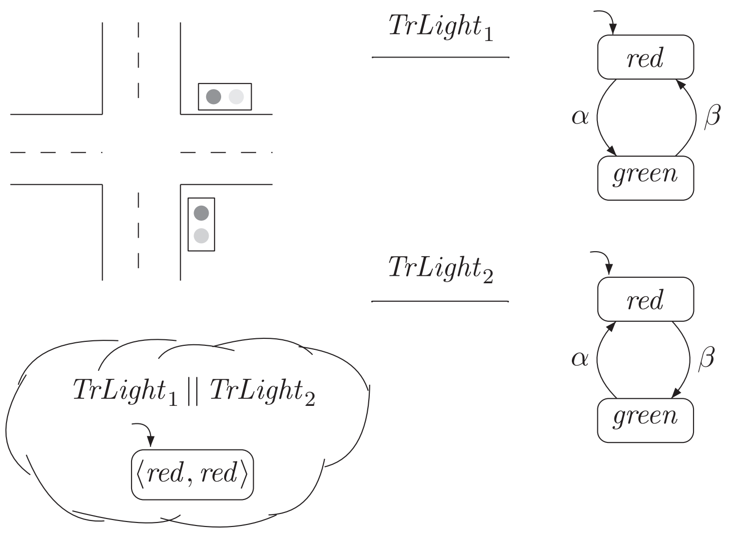


Figure 3.1: An example of a deadlock situation.

Example 3.1. Deadlock for Fault Designed Traffic Lights

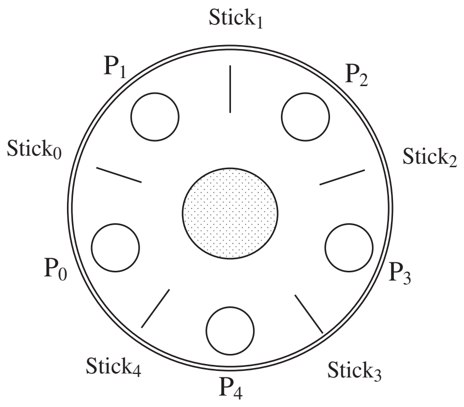
Consider the parallel composition of two transition systems

$$TrLight_1 \,\|\, TrLight_2$$

modeling the traffic lights of two intersecting roads. Both traffic lights synchronize by means of the actions α and β that indicate the change of light (see Figure 3.1). The apparently trivial error to let both traffic lights start with a red light results in a deadlock. While the first traffic light is waiting to be synchronized on action α, the second traffic light is blocked, since it is waiting to be synchronized with action β. ■

Example 3.2. Dining Philosophers

This example, originated by Dijkstra, is one of the most prominent examples in the field of concurrent systems.



Five philosophers are sitting at a round table with a bowl of rice in the middle. For the philosophers (being a little unworldly) life consists of thinking and eating (and waiting, as we will see). To take some rice out of the bowl, a philosopher needs two chopsticks. In between two neighboring philosophers, however, there is only a single chopstick. Thus, at any time only one of two neighboring philosophers can eat. Of course, the use of the chopsticks is exclusive and eating with hands is forbidden.

Note that a deadlock scenario occurs when all philosophers possess a single chopstick. The problem is to design a protocol for the philosophers, such that the complete system is deadlock-free, i.e., at least one philosopher can eat and think infinitely often. Additionally, a fair solution may be required with each philosopher being able to think and eat infinitely often. The latter characteristic is called freedom of *individual starvation*.

The following obvious design cannot ensure deadlock freedom. Assume the philosophers and the chopsticks are numbered from 0 to 4. Furthermore, assume all following calculations be “modulo 5”, e.g., chopstick $i{-}1$ for $i{=}0$ denotes chopstick 4, and so on.

Philosopher i has stick i on his left and stick $i{-}1$ on his right side. The action $\mathit{request}_{i,i}$ express that stick i is picked up by philosopher i. Accordingly, $\mathit{request}_{i-1,i}$ denotes the action by means of which philosopher i picks up the $(i{-}1)$th stick. The actions $\mathit{release}_{i,i}$ and $\mathit{release}_{i-1,i}$ have a corresponding meaning.

The behavior of philosopher i (called process Phil_i) is specified by the transition system depicted in the left part of Figure 3.2. Solid arrows depict the synchronizations with the i-th stick, dashed arrows refer to communications with the $i{-}1$th stick. The sticks are modeled as independent processes (called Stick_i) with which the philosophers synchronize via actions *request* and *release*; see the right part of Figure 3.2 that represents the process of stick i. A stick process prevents philosopher i from picking up the ith stick when philosopher $i{+}1$ is using it.

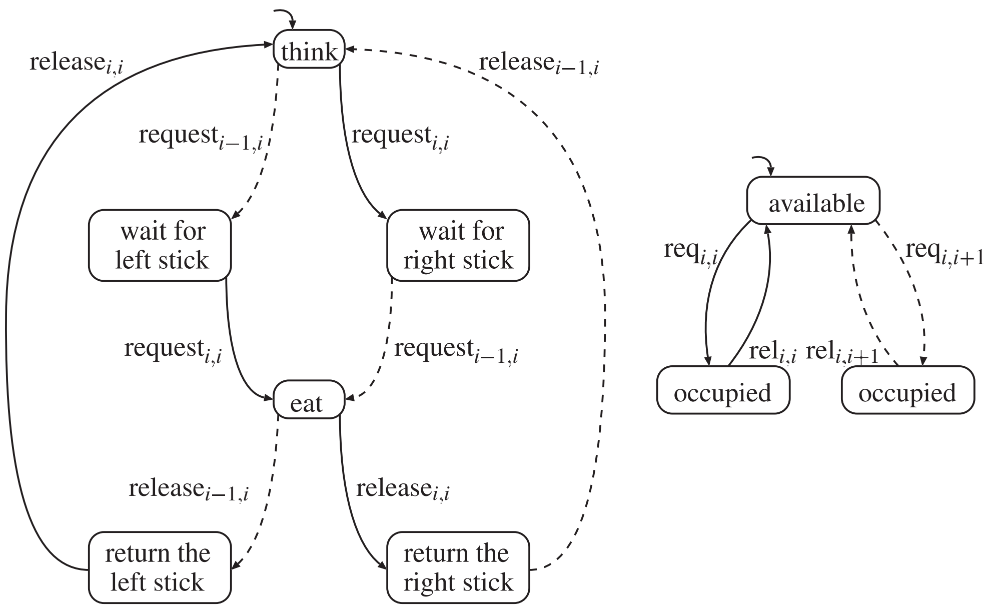


Figure 3.2: Transition systems for the ith philosopher and the ith stick.

The complete system is of the form:

$$Phil_4 \parallel Stick_3 \parallel Phil_3 \parallel Stick_2 \parallel Phil_2 \parallel Stick_1 \parallel Phil_1 \parallel Stick_0 \parallel Phil_0 \parallel Stick_4$$

This (initially obvious) design leads to a deadlock situation, e.g., if all philosophers pick up their left stick at the same time. A corresponding execution leads from the initial state

$$\langle think_4, avail_3, think_3, avail_2, think_2, avail_1, think_1, avail_0, think_0, avail_4 \rangle$$

by means of the action sequence $request_4$, $request_3$, $request_2$, $request_1$, $request_0$ (or any other permutation of these 5 request actions) to the terminal state

$$\langle wait_{4,0}, occ_{4,4}, wait_{3,4}, occ_{3,3}, wait_{2,3}, occ_{2,2}, wait_{1,2}, occ_{1,1}, wait_{0,1}, occ_{0,0} \rangle.$$

This terminal state represents a deadlock with each philosopher waiting for the needed stick to be released.

A possible solution to this problem is to make the sticks available for only one philosopher at a time. The corresponding chopstick process is depicted in the right part of Figure 3.3. In state $available_{i,j}$ only philosopher j is allowed to pick up the ith stick. The above-mentioned deadlock situation can be avoided by the fact that some sticks (e.g., the first, the third, and the fifth stick) start in state $available_{i,i}$, while the remaining sticks start in state $available_{i,i+1}$. It can be verified that this solution is deadlock- and starvation-free.

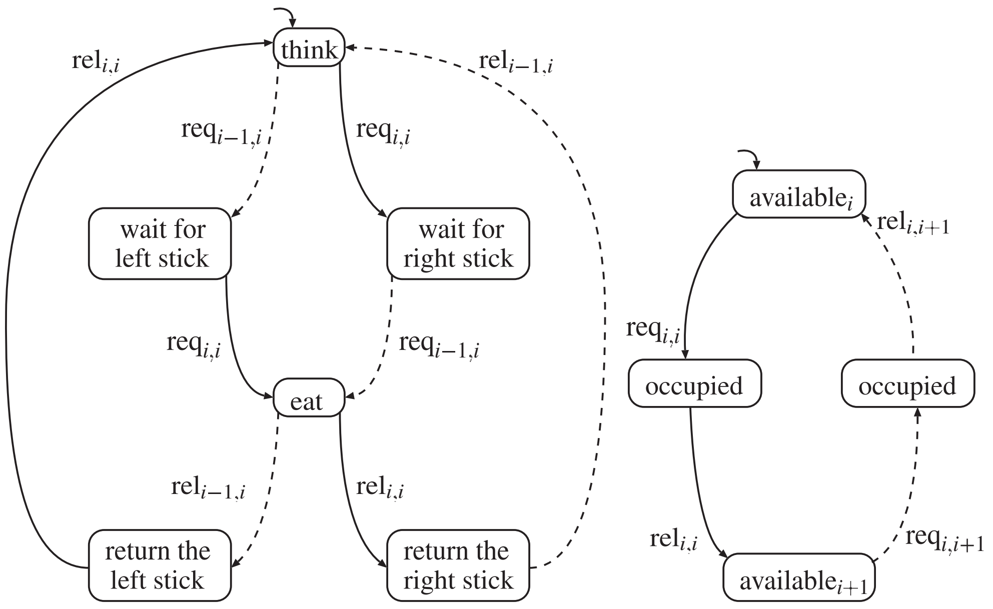


Figure 3.3: Improved variant of the ith philosopher and the ith stick.

A further characteristic often required for concurrent systems is robustness against failure of their components. In the case of the dining philosophers, robustness can be formulated in a way that ensures deadlock and starvation freedom even if one of the philosophers is "defective" (i.e., does not leave the think phase anymore).[1] The above-sketched deadlock- and starvation-free solution can be modified to a fault-tolerant solution by changing the transition systems of philosophers and sticks such that philosopher $i+1$ can pick up the ith stick even if philosopher i is thinking (i.e., does not need stick i) independent of whether stick i is in state $available_{i,i}$ or $available_{i,i+1}$. The corresponding is also true when the roles of philosopher i and $i+1$ are reversed. This can be established by adding a single Boolean variable x_i to philosopher i (see Figure 3.4). The variable x_i informs the neighboring philosophers about the current location of philosopher i. In the indicated sketch, x_i is a Boolean variable which is true if and only if the ith philosopher is thinking. Stick i is made available to philosopher i if stick i is in location $available_i$ (as before), or if stick i is in location $available_{i+1}$ while philosopher $i+1$ is thinking.

Note that the above description is at the level of program graphs. The complete system is a channel system with *request* and *release* actions standing for handshaking over a channel of capacity 0. ■

[1]Formally, we add a loop to the transition system of a defective philosopher at state $think_i$.

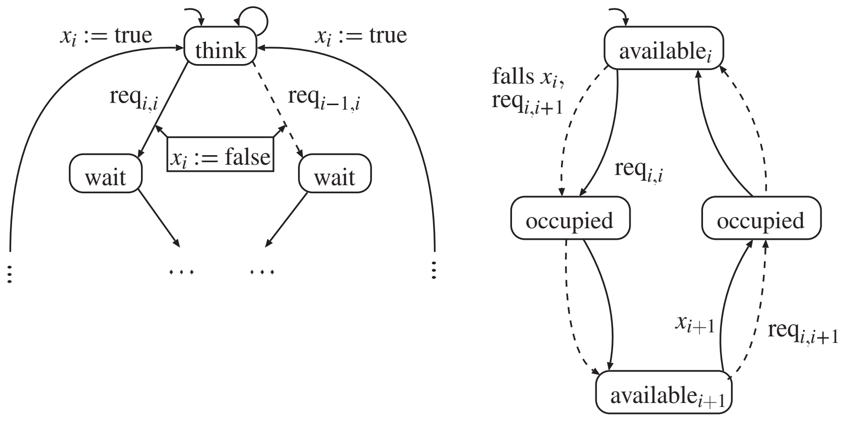


Figure 3.4: Fault-tolerant variant of the dining philosophers.

3.2 Linear-Time Behavior

To analyze a computer system represented by a transition system, either an action-based or a state-based approach can be followed. The state-based approach abstracts from actions; instead, only labels in the state sequences are taken into consideration. In contrast, the action-based view abstracts from states and refers only to the action labels of the transitions. (A combined action- and state-based view is possible, but leads to more involved definitions and concepts. For this reason it is common practice to abstract from either action or state labels.) Most of the existing specification formalisms and associated verification methods can be formulated in a corresponding way for both perspectives.

In this chapter, we mainly focus on the state-based approach. Action labels of transitions are only necessary for modeling communication; thus, they are of no relevance in the following chapters. Instead, we use the atomic propositions of the states to formulate system properties. Therefore, the verification algorithms operate on the *state graph* of a transition system, the digraph originating from a transition system by abstracting from action labels.

3.2.1 Paths and State Graph

Let $TS = (S, Act, \to, I, AP, L)$ be a transition system.

Definition 3.3. State Graph

The *state graph* of TS, notation $G(TS)$, is the digraph (V, E) with vertices $V = S$ and edges $E = \{(s, s') \in S \times S \mid s' \in Post(s)\}$. ■

The state graph of transition system TS has a vertex for each state in TS and an edge between vertices s and s' whenever s' is a direct successor of s in TS for some action α. It is thus simply obtained from TS by omitting all state labels (i.e., the atomic propositions), all transition labels (i.e., the actions), and by ignoring the fact whether a state is initial or not. Moreover, multiple transitions (that have different action labels) between states are represented by a single edge. This seems to suggest that the state labels are no longer of any use; later on, we will see how these state labels will be used to check the validity of properties.

Let $Post^*(s)$ denote the states that are reachable in state graph $G(TS)$ from s. This notion is generalized toward sets of states in the usual way (i.e., pointwise extension): for $C \subseteq S$ let

$$Post^*(C) \; = \; \bigcup_{s \in C} Post^*(s).$$

The notations $Pre^*(s)$ and $Pre^*(C)$ have analogous meaning. The set of states that are reachable from some initial state, notation $Reach(TS)$, equals $Post^*(I)$.

As explained in Chapter 2, the possible behavior of a transition system is defined by an execution fragment. Recall that an execution fragment is an alternating sequence of states and actions. As we consider a state-based approach, the actions are not of importance and are omitted. The resulting "runs" of a transition system are called *paths*. The following definitions define path fragments, initial and maximal path fragments, and so on. These notions are easily obtained from the same notions for executions by omitting the actions.

Definition 3.4. Path Fragment

A *finite* path fragment $\widehat{\pi}$ of TS is a finite state sequence $s_0\, s_1 \ldots s_n$ such that $s_i \in Post(s_{i-1})$ for all $0 < i \leqslant n$, where $n \geqslant 0$. An *infinite* path fragment π is an infinite state sequence $s_0\, s_1\, s_2 \ldots$ such that $s_i \in Post(s_{i-1})$ for all $i > 0$. ■

We adopt the following notational conventions for infinite path fragment $\pi = s_0\, s_1 \ldots$. The initial state of π is denoted by $first(\pi) = s_0$. For $j \geqslant 0$, let $\pi[j] = s_j$ denote the jth state of

π and $\pi[..j]$ denote the jth prefix of π, i.e., $\pi[..j] = s_0\, s_1 \ldots s_j$. Similarly, the jth suffix of π, notation $\pi[j..]$, is defined as $\pi[j..] = s_j\, s_{j+1} \ldots$. These notions are defined analogously for finite paths. Besides, for finite path $\widehat{\pi} = s_0\, s_1 \ldots s_n$, let $\mathit{last}(\widehat{\pi}) = s_n$ denote the last state of $\widehat{\pi}$, and $\mathit{len}(\widehat{\pi}) = n$ denote the length of $\widehat{\pi}$. For infinite path π these notions are defined by $\mathit{len}(\pi) = \infty$ and $\mathit{last}(\pi) = \perp$, where $\perp$ denotes "undefined".

Definition 3.5. Maximal and Initial Path Fragment

A *maximal* path fragment is either a finite path fragment that ends in a terminal state, or an infinite path fragment. A path fragment is called *initial* if it starts in an initial state, i.e., if $s_0 \in I$. ■

A maximal path fragment is a path fragment that cannot be prolonged: either it is infinite or it is finite but ends in a state from which no transition can be taken. Let $\mathit{Paths}(s)$ denote the set of maximal path fragments π with $\mathit{first}(\pi) = s$, and $\mathit{Paths}_{\mathit{fin}}(s)$ denote the set of all finite path fragments $\widehat{\pi}$ with $\mathit{first}(\widehat{\pi}) = s$.

Definition 3.6. Path

A *path* of transition system *TS* is an initial, maximal path fragment.[2] ■

Let $\mathit{Paths}(TS)$ denote the set of all paths in *TS*, and $\mathit{Paths}_{\mathit{fin}}(TS)$ the set of all initial, finite path fragments of *TS*.

Example 3.7. Beverage Vending Machine

Consider the beverage vending machine of Example 2.2 on page 21. For convenience, its transition system is repeated in Figure 3.5. As the state labeling is simply $L(s) = \{ s \}$ for each state s, the names of states may be used in paths (as in this example), as well as atomic propositions (as used later on). Example path fragments of this transition system are

$$\begin{aligned} \pi_1 &= \mathit{pay\ select\ soda\ pay\ select\ soda} \ldots \\ \pi_2 &= \mathit{select\ soda\ pay\ select\ beer} \ldots \\ \widehat{\pi} &= \mathit{pay\ select\ soda\ pay\ select\ soda}\,. \end{aligned}$$

These path fragments result from the execution fragments indicated in Example 2.8 on page 25. Only π_1 is a path. The infinite path fragment π_2 is maximal but not initial. $\widehat{\pi}$ is initial but not maximal since it is finite while ending in a state that has outgoing

[2]It is important to realize the difference between the notion of a path in a transition system and the notion of a path in a digraph. A path in a transition system is maximal, whereas a path in a digraph in the graph-theoretical sense is not always maximal. Besides, paths in a digraph are usually required to be finite whereas paths in transition systems may be infinite.

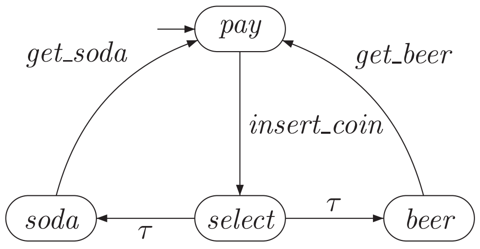


Figure 3.5: A transition system of a simple beverage vending machine.

transitions. We have that $\mathit{last}(\widehat{\pi}) = \mathit{soda}$, $\mathit{first}(\pi_2) = \mathit{select}$, $\pi_1[0] = \mathit{pay}$, $\pi_1[3] = \mathit{pay}$, $\pi_1[..5] = \widehat{\pi}$, $\widehat{\pi}[..2] = \widehat{\pi}[3..]$, $\mathit{len}(\widehat{\pi}) = 5$, and $\mathit{len}(\pi_1) = \infty$. ■

3.2.2 Traces

Executions (as introduced in Chapter 2) are alternating sequences consisting of states and actions. Actions are mainly used to model the (possibility of) interaction, be it synchronous or asynchronous communication. In the sequel, interaction is not our prime interest, but instead we focus on the states that are visited during executions. In fact, the states themselves are not "observable", but just their atomic propositions. Thus, rather than having an execution of the form $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \ldots$ we consider sequences of the form $L(s_0)\, L(s_1)\, L(s_2) \ldots$ that register the (set of) atomic propositions that are valid along the execution. Such sequences are called *traces*.

The traces of a transition system are thus words over the alphabet 2^{AP}. In the following it is assumed that a transition system has no terminal states. In this case, all traces are infinite words. (Recall that the traces of a transition system have been defined as traces induced by its initial maximal path fragments. See also Appendix A.2, page 912). This assumption is made for simplicity and does not impose any serious restriction. First of all, prior to checking any (linear-time) property, a reachability analysis could be carried out to determine the set of terminal states. If indeed some terminal state is encountered, the system contains a deadlock and has to be repaired before any further analysis. Alternatively, each transition system *TS* (that probably has a terminal state) can be extended such that for each terminal state s in *TS* there is a new state s_{stop}, transition $s \rightarrow s_{stop}$, and s_{stop} is equipped with a self-loop, i.e., $s_{stop} \rightarrow s_{stop}$. The resulting "equivalent" transition system obviously has no terminal states.[3]

[3]A further alternative is to adapt the linear-time framework for transition systems with terminal states. The main concepts of this chapter are still applicable, but require some adaptions to distinguish nonmax-

Definition 3.8. Trace and Trace Fragment

Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states. The *trace* of the infinite path fragment $\pi = s_0\, s_1 \ldots$ is defined as $trace(\pi) = L(s_0)\, L(s_1) \ldots$. The trace of the finite path fragment $\widehat{\pi} = s_0\, s_1 \ldots s_n$ is defined as $trace(\widehat{\pi}) = L(s_0)\, L(s_1) \ldots L(s_n)$. ■

The trace of a path fragment is thus the induced finite or infinite word over the alphabet 2^{AP}, i.e., the sequence of sets of atomic propositions that are valid in the states of the path. The set of traces of a set Π of paths is defined in the usual way:

$$trace(\Pi) \;=\; \{\, trace(\pi) \mid \pi \in \Pi \,\}.$$

A trace of state s is the trace of an infinite path fragment π with $first(\pi) = s$. Accordingly, a finite trace of s is the trace of a finite path fragment that starts in s. Let $Traces(s)$ denote the set of traces of s, and $Traces(TS)$ the set of traces of the initial states of transition system TS:

$$Traces(s) = trace(Paths(s)) \quad \text{and} \quad Traces(TS) = \bigcup_{s \in I} Traces(s).$$

In a similar way, the finite traces of a state and of a transition system are defined:

$$Traces_{fin}(s) = trace(Paths_{fin}(s)) \quad \text{and} \quad Traces_{fin}(TS) = \bigcup_{s \in I} Traces_{fin}(s).$$

Example 3.9. Semaphore-Based Mutual Exclusion

Consider the transition system TS_{Sem} as depicted in Figure 3.6. This two-process mutual exclusion example has been described before in Example 2.24 (page 43).

Assume the available atomic propositions are $crit_1$ and $crit_2$, i.e.,

$$AP = \{\, crit_1, crit_2 \,\}.$$

The proposition $crit_1$ holds in any state of the transition system TS_{Sem} where the first process (called P_1) is in its critical section. Proposition $crit_2$ has the same meaning for the second process (i.e., P_2).

Consider the execution in which the processes P_1 and P_2 enter their critical sections in an alternating fashion. Besides, they only request to enter the critical section when the

imal and maximal finite paths.

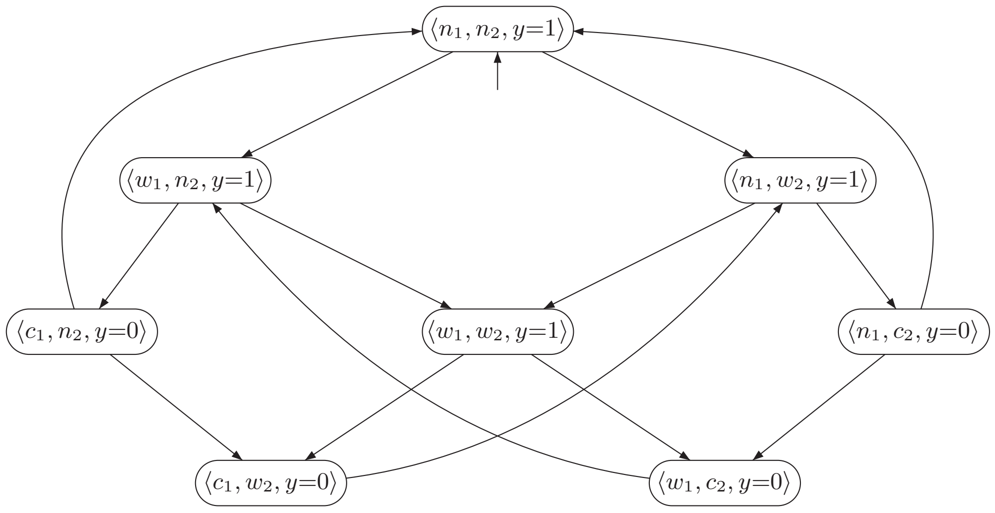

$\langle n_1, n_2, y{=}1\rangle$
$\langle w_1, n_2, y{=}1\rangle$
$\langle n_1, w_2, y{=}1\rangle$
$\langle c_1, n_2, y{=}0\rangle$
$\langle w_1, w_2, y{=}1\rangle$
$\langle n_1, c_2, y{=}0\rangle$
$\langle c_1, w_2, y{=}0\rangle$
$\langle w_1, c_2, y{=}0\rangle$

Figure 3.6: Transition system of semaphore-based mutual exclusion algorithm.

other process is no longer in its critical section. Situations in which one process is in its critical section whereas the other is moving from the noncritical state to the waiting state are impossible.

The path π in the state graph of TS_{Sem} where process P_1 is the first to enter its critical section is of the form

$$\begin{aligned}\pi \quad = \quad & \langle n_1, n_2, y = 1\rangle \rightarrow \langle w_1, n_2, y = 1\rangle \rightarrow \langle c_1, n_2, y = 0\rangle \rightarrow \\ & \langle n_1, n_2, y = 1\rangle \rightarrow \langle n_1, w_2, y = 1\rangle \rightarrow \langle n_1, c_2, y = 0\rangle \rightarrow \ldots\end{aligned}$$

The trace of this path is the infinite word:

$$trace(\pi) \;=\; \varnothing\,\varnothing\,\{\,crit_1\,\}\,\varnothing\,\varnothing\,\{\,crit_2\,\}\,\varnothing\,\varnothing\,\{\,crit_1\,\}\,\varnothing\,\varnothing\,\{\,crit_2\,\}\ldots.$$

The trace of the finite path fragment

$$\begin{aligned}\widehat{\pi} \quad = \quad & \langle n_1, n_2, y = 1\rangle \rightarrow \langle w_1, n_2, y = 1\rangle \rightarrow \langle w_1, w_2, y = 1\rangle \rightarrow \\ & \langle w_1, c_2, y = 0\rangle \rightarrow \langle w_1, n_2, y = 1\rangle \rightarrow \langle c_1, n_2, y = 0\rangle\end{aligned}$$

is $trace(\widehat{\pi}) \;=\; \varnothing\,\varnothing\,\varnothing\,\{\,crit_2\,\}\,\varnothing\,\{\,crit_1\,\}$. ■

3.2.3 Linear-Time Properties

Linear-time properties specify the traces that a transition system should exhibit. Informally speaking, one could say that a linear-time property specifies the admissible (or desired) behavior of the system under consideration. In the following we provide a formal definition of such properties. This definition is rather elementary, and gives a good basic understanding of what a linear-time property is. In Chapter 5, a logical formalism will be introduced that allows for the specification of linear-time properties.

In the following, we assume a fixed set of propositions AP. A linear-time (LT) property is a requirement on the traces of a transition system. Such property can be understood as a requirement over all words over AP, and is defined as the set of words (over AP) that are admissible:

Definition 3.10. LT Property

A *linear-time property* (LT property) over the set of atomic propositions AP is a subset of $(2^{AP})^\omega$. ■

Here, $(2^{AP})^\omega$ denotes the set of words that arise from the infinite concatenation of words in 2^{AP}. An LT property is thus a language (set) of infinite words over the alphabet 2^{AP}. Note that it suffices to consider infinite words only (and not finite words), as transition systems without terminal states are considered. The fulfillment of an LT property by a transition system is defined as follows.

Definition 3.11. Satisfaction Relation for LT Properties

Let P be an LT property over AP and $TS = (S, Act, \to, I, AP, L)$ a transition system without terminal states. Then, $TS = (S, Act, \to, I, AP, L)$ *satisfies* P, denoted $TS \models P$, iff $Traces(TS) \subseteq P$. State $s \in S$ satisfies P, notation $s \models P$, whenever $Traces(s) \subseteq P$. ■

Thus, a transition system satisfies the LT property P if all its traces respect P, i.e., if all its behaviors are admissible. A state satisfies P whenever all traces starting in this state fulfill P.

Example 3.12. Traffic Lights

Consider two simplified traffic lights that only have two possible settings: red and green. Let the propositions of interest be

$$AP = \{ red_1, green_1, red_2, green_2 \}$$

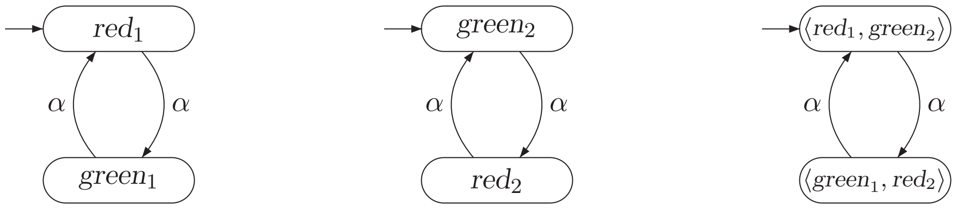


Figure 3.7: Two fully synchronized traffic lights (left and middle) and their parallel composition (right).

We consider two LT properties of these traffic lights and give some example words that are contained by such properties. First, consider the property P that states:

"The first traffic light is infinitely often green".

This LT property corresponds to the set of infinite words of the form $A_0\,A_1\,A_2\,\ldots$ over 2^{AP}, such that $green_1 \in A_i$ holds for infinitely many i. For example, P contains the infinite words

$$\begin{array}{l}
\{\,red_1, green_2\,\}\,\{\,green_1, red_2\,\}\,\{\,red_1, green_2\,\}\,\{\,green_1, red_2\,\}\ldots,\\
\varnothing\,\{\,green_1\,\}\,\varnothing\,\{\,green_1\,\}\,\varnothing\,\{\,green_1\,\}\,\varnothing\,\{\,green_1\,\}\;\varnothing\ldots\\
\{\,red_1, green_1\,\}\,\{\,red_1, green_1\,\}\,\{\,red_1, green_1\,\}\,\{\,red_1, green_1\,\}\ldots \quad \text{and}\\
\{\,green_1, green_2\,\}\,\{\,green_1, green_2\,\}\,\{\,green_1, green_2\,\}\,\{\,green_1, green_2\,\}\ldots
\end{array}$$

The infinite word $\{\,red_1, green_1\,\}\,\{\,red_1, green_1\,\}\,\varnothing\,\varnothing\,\varnothing\,\varnothing\ldots$ is not in P as it contains only finitely many occurrences of $green_1$.

As a second LT property, consider P':

"The traffic lights are never both green simultaneously".

This property is formalized by the set of infinite words of the form $A_0\,A_1\,A_2\ldots$ such that either $green_1 \notin A_i$ or $green_2 \notin A_i$, for all $i \geqslant 0$. For example, the following infinite words are in P':

$$\begin{array}{l}
\{\,red_1, green_2\,\}\,\{\,green_1, red_2\,\}\,\{\,red_1, green_2\,\}\,\{\,green_1, red_2\,\}\ldots,\\
\varnothing\,\{\,green_1\,\}\,\varnothing\,\{\,green_1\,\}\,\varnothing\,\{\,green_1\,\}\,\varnothing\,\{\,green_1\,\}\,\varnothing\ldots \quad \text{and}\\
\{\,red_1, green_1\,\}\,\{\,red_1, green_1\,\}\,\{\,red_1, green_1\,\}\,\{\,red_1, green_1\,\}\ldots,
\end{array}$$

whereas the infinite word $\{\,red_1\, green_2\,\}\,\{\,green_1, green_2\,\},\ldots$ is not in P'.

The traffic lights depicted in Figure 3.7 are at intersecting roads and their switching is synchronized, i.e., if one light switches from red to green, the other switches from green to red. In this way, the lights always have complementary colors. Clearly, these traffic lights satisfy both P and P'. Traffic lights that switch completely autonomously will neither satisfy P—there is no guarantee that the first traffic light is green infinitely often—nor P'. ■

Often, an LT property does not refer to all atomic propositions occurring in a transition system, but just to a relatively small subset thereof. For a property P over a set of propositions $AP' \subseteq AP$, only the labels in AP' are relevant. Let $\widehat{\pi}$ be a finite path fragment of TS. We write $trace_{AP'}(\widehat{\pi})$ to denote the finite trace of $\widehat{\pi}$ where only the atomic propositions in AP' are considered. Accordingly, $trace_{AP'}(\pi)$ denotes the trace of an infinite path fragment π by focusing on propositions in AP' Thus, for $\pi = s_0\, s_1\, s_2 \ldots$, we have

$$trace_{AP'}(\pi) \;=\; L'(s_0)\, L'(s_1) \ldots \;=\; (L(s_0) \cap AP')\; (L(s_1) \cap AP') \ldots$$

Let $Traces_{AP'}(TS)$ denote the set of traces $trace_{AP'}(Paths(TS))$. Whenever the set AP' of atomic propositions is clear from the context, the subscript AP' is omitted. In the rest of this chapter, the restriction to a relevant subset of atomic propositions is often implicitly made.

Example 3.13. The Mutual Exclusion Property

In Chapter 2, several mutual exclusion algorithms have been considered. For specifying the mutual exclusion property—always at most one process is in its critical section—it suffices to only consider the atomic propositions $crit_1$ and $crit_2$. Other atomic propositions are not of any relevance for this property. The formalization of the mutual exclusion property is given by the LT property

$$P_{mutex} \;=\; \text{set of infinite words } A_0\, A_1\, A_2 \ldots \text{ with } \{\, crit_1, crit_2 \,\} \not\subseteq A_i \text{ for all } 0 \leqslant i.$$

For example, the infinite words

$$\begin{array}{ll} \{\, crit_1 \,\}\,\{\, crit_2 \,\}\,\{\, crit_1 \,\}\,\{\, crit_2 \,\}\,\{\, crit_1 \,\}\,\{\, crit_2 \,\}\,\ldots, & \text{and} \\ \{\, crit_1 \,\}\,\{\, crit_1 \,\}\,\{\, crit_1 \,\}\,\{\, crit_1 \,\}\,\{\, crit_1 \,\}\,\{\, crit_1 \,\}\ldots, & \text{and} \\ \varnothing\,\varnothing\,\varnothing\,\varnothing\,\varnothing\,\varnothing\,\varnothing \ldots & \end{array}$$

are all contained in P_{mutex}. However, this does not apply to words of the form

$$\{\, crit_1 \,\}\,\varnothing\,\{\, crit_1, crit_2 \,\}\ldots$$

The transition system $TS_{Arb} = (TS_1 \,|||\, TS_2) \,\|\, Arbiter$ described in Example 2.28 (page 50) fulfills the mutex property, i.e.,

$$TS_{Arb} \;\models\; P_{mutex}.$$

It is left to the reader to check that the mutex property is also fulfilled by the semaphore-based mutual exclusion algorithm (see Figure 3.6 on page 99) and Peterson's algorithm (see Example 2.25 on page 45). ■

Example 3.14. Starvation Freedom

Guaranteeing mutual exclusion is a significant property of mutual exclusion algorithms, but is not the only relevant property. An algorithm that never allows a process to enter its critical section will do, but is certainly not intended. Besides, a property is imposed that requires a process that wants to enter the critical section to be able to eventually do so. This property prevents a process from waiting ad infinitum and is formally specified as the LT property $P_{finwait}$ = set of infinite words $A_0\, A_1\, A_2 \ldots$ such that

$$\forall j. lwait_i \in A_j \;\Rightarrow\; \exists k \geqslant j. wait_i \in A_k \;\text{ for each } i \in \{\,1, 2\,\}.$$

Here, we assumed the set of propositions to be:

$$AP \;=\; \{\, wait_1, crit_1, wait_2, crit_2 \,\}.$$

Property $P_{finwait}$ expresses that each of the two processes enters its critical section eventually if they are waiting. That is, a process has to wait some finite amount before entering the critical section. It does not express that a process that waits often, is often entering the critical section.

Consider the following variant. The LT property $P_{nostarve}$ = set of infinite words $A_0\, A_1\, A_2 \ldots$ such that:

$$(\forall k \geqslant 0.\, \exists j \geqslant k.\; wait_i \in A_j \;) \;\Rightarrow\; (\forall k \geqslant 0.\, \exists j \geqslant k.\; crit_i \in A_j \;) \qquad \text{for each } i \in \{\,1, 2\,\}.$$

In abbreviated form we write:

$$\left(\overset{\infty}{\exists}\; j.\; wait_i \in A_j\right) \;\Rightarrow\; \left(\overset{\infty}{\exists}\; j.\; crit_i \in A_j\right) \qquad \text{for each } i \in \{\,1, 2\,\}$$

where $\overset{\infty}{\exists}$ stands for "there are infinitely many".

Property $P_{nostarve}$ expresses that each of the two processes enters its critical section infinitely often if they are waiting infinitely often. This natural requirement is, however, *not* satisfied for the semaphore-based solution, since

$$\varnothing\; (\{\, wait_2 \,\}\, \{\, wait_1, wait_2 \,\}\, \{\, crit_1, wait_2 \,\}\,)^{\omega}$$

is a possible trace of the transition system but does not belong to $P_{nostarve}$. This trace represents an execution in which only the first process enters its critical section infinitely often. In fact, the second process waits infinitely long to enter its critical section.

It is left to the reader to check that the transition system modeling Peterson's algorithm (see Example 2.25, page 45) does indeed satisfy $P_{nostarve}$. ■

3.2.4 Trace Equivalence and Linear-Time Properties

LT properties specify the (infinite) traces that a transition system should exhibit. If transition systems TS and TS' have the same traces, one would expect that they satisfy the same LT properties. Clearly, if $TS \models P$, then all traces of TS are contained in P, and when $Traces(TS) = Traces(TS')$, the traces of TS' are also contained in P. Otherwise, whenever $TS \not\models P$, there is a trace in $Traces(TS)$ that is prohibited by P, i.e., not included in the set P of traces. As $Traces(TS) = Traces(TS')$, also TS' exhibits this prohibited trace, and thus $TS' \not\models P$. The precise relationship between trace equivalence, trace inclusion, and the satisfaction of LT properties is the subject of this section.

We start by considering trace inclusion and its importance in concurrent system design. Trace inclusion between transition systems TS and TS' requires that all traces exhibited by TS can also be exhibited by TS', i.e., $Traces(TS) \subseteq Traces(TS')$. Note that transition system TS' may exhibit more traces, i.e., may have some (linear-time) behavior that TS does not have. In stepwise system design, where designs are successively refined, trace inclusion is often viewed as an implementation relation in the sense that

$$Traces(TS) \subseteq Traces(TS') \text{ means } TS \text{ “is a correct implementation of” } TS'.$$

For example, let TS' be a (more abstract) design where parallel composition is modeled by interleaving, and TS its realization where (some of) the interleaving is resolved by means of some scheduling mechanism. TS may thus be viewed as an “implementation” of TS', and clearly, $Traces(TS) \subseteq Traces(TS')$.

What does trace inclusion have to do with LT properties? The following theorem shows that trace inclusion is *compatible* with requirement specifications represented as LT properties.

Theorem 3.15. Trace Inclusion and LT Properties

Let TS and TS′ be transition systems without terminal states and with the same set of propositions AP. Then the following statements are equivalent:

(a) $Traces(TS) \subseteq Traces(TS')$

(b) For any LT property P: $TS' \models P$ *implies* $TS \models P$.

Proof: (a) $\Longrightarrow$ (b): Assume $Traces(TS) \subseteq Traces(TS')$, and let P be an LT property such that $TS' \models P$. From Definition 3.11 it follows that $Traces(TS') \subseteq P$. Given $Traces(TS) \subseteq$

$Traces(TS')$, it now follows that $Traces(TS) \subseteq P$. By Definition 3.11 it follows that $TS \models P$.

(b) $\Longrightarrow$ (a): Assume that for all LT properties it holds that: $TS' \models P$ implies $TS \models P$. Let $P = Traces(TS')$. Obviously, $TS' \models P$, as $Traces(TS') \subseteq Traces(TS')$. By assumption, $TS \models P$. Hence, $Traces(TS) \subseteq Traces(TS')$. ■

This simple observation plays a decisive role for the design by means of successive refinement. If TS' is the transition system representing a preliminary design and TS is a transition system originating from a refinement of TS' (i.e., a more detailed design), then it can immediately—without explicit proof—be concluded from the relation $Traces(TS) \subseteq Traces(TS')$ that any LT property that holds in TS' also holds for TS.

Example 3.16. Refining the Semaphore-Based Mutual Exclusion Algorithm

Let $TS' = TS_{Sem}$, the transition system representing the semaphore-based mutual exclusion algorithm (see Figure 3.6 on page 99) and let TS be the transition system obtained from TS' by removing the transition

$$\langle wait_1, wait_2, y = 1\rangle \;\rightarrow\; \langle wait_1, crit_2, y = 0\rangle.$$

Stated in words, from the situation in which both processes are waiting, it is no longer possible that the second process (P_2) acquires access to the critical section. This thus yields a model that assigns higher priority to process P_1 than to process P_2 when both processes are competing to access the critical section. As a transition is removed, it immediately follows that $Traces(TS) \subseteq Traces(TS')$. Consequently, by the fact that TS' ensures mutual exclusion, i.e., $TS' \models P_{mutex}$, it follows by Theorem 3.15 that $TS \models P_{mutex}$. ■

Transition systems are said to be trace-equivalent if they have the same set of traces:

Definition 3.17. Trace Equivalence

Transition systems TS and TS' are *trace-equivalent* with respect to the set of propositions AP if $Traces_{AP}(TS) = Traces_{AP}(TS')$. [4] ■

Theorem 3.15 implies equivalence of two trace-equivalent transition systems with respect to requirements formulated as LT properties.

[4]Here, we assume two transition systems with sets of propositions that include AP.

Corollary 3.18. **Trace Equivalence and LT Properties**

Let TS and TS′ be transition systems without terminal states and with the same set of atomic propositions. Then:

$$\text{Traces}(TS) = \text{Traces}(TS') \iff \textit{TS and TS' satisfy the same LT properties.}$$

There thus does not exist an LT property that can distinguish between trace-equivalent transition systems. Stated differently, in order to establish that the transition systems TS and TS' are *not* trace-equivalent it suffices to find one LT property that holds for one but not for the other.

Example 3.19. *Two Beverage Vending Machines*

Consider the two transition systems in Figure 3.8 that both model a beverage vending

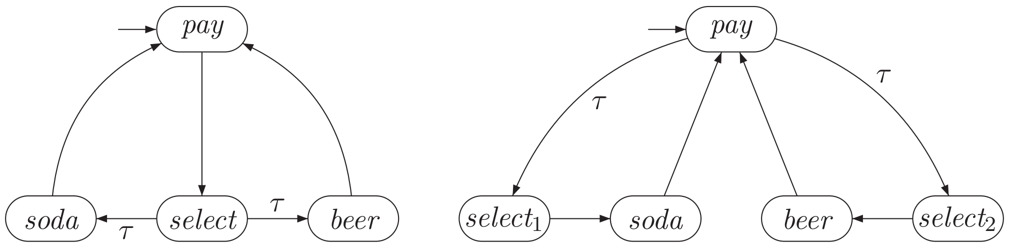


Figure 3.8: Two beverage vending machines.

machine. For simplicity, the observable action labels of transitions have been omitted. Both machines are able to offer soda and beer. The left transition system models a beverage machine that after insertion of a coin nondeterministically chooses to either provide soda or beer. The right one, however, has two selection buttons (one for each beverage), and after insertion of a coin, nondeterministically blocks one of the buttons. In either case, the user has no control over the beverage obtained—the choice of beverage is under full control of the vending machine.

Let $AP = \{ pay, soda, beer \}$. Although the two vending machines behave differently, it is not difficult to see that they exhibit the same traces when considering AP, as for both machines traces are alternating sequences of *pay* and either *soda* or *beer*. The vending machines are thus trace-equivalent. By Corollary 3.18 both vending machines satisfy exactly the same LT properties. Stated differently, it means that there does not exist an LT property that distinguishes between the two vending machines. ■

3.3 Safety Properties and Invariants

Safety properties are often characterized as “nothing bad should happen”. The mutual exclusion property—always at most one process is in its critical section—is a typical safety property. It states that the bad thing (having two or more processes in their critical section simultaneously) never occurs. Another typical safety property is deadlock freedom. For the dining philosophers (see Example 3.2, page 90), for example, such deadlock could be characterized as the situation in which all philosophers are waiting to pick up the second chopstick. This bad (i.e., unwanted) situation should never occur.

3.3.1 Invariants

In fact, the above safety properties are of a particular kind: they are *invariants*. Invariants are LT properties that are given by a condition Φ for the states and require that Φ holds for all reachable states.

Definition 3.20. Invariant

An LT property P_{inv} over AP is an *invariant* if there is a propositional logic formula[5] Φ over AP such that

$$P_{inv} = \left\{ A_0 A_1 A_2 \ldots \in (2^{AP})^\omega \mid \forall j \geqslant 0.\ A_j \models \Phi \right\}.$$

Φ is called an invariant condition (or state condition) of P_{inv}. ■

Note that

$$\begin{array}{lll} TS \models P_{inv} & \text{iff} & trace(\pi) \in P_{inv} \text{ for all paths } \pi \text{ in } TS \\ & \text{iff} & L(s) \models \Phi \text{ for all states } s \text{ that belong to a path of } TS \\ & \text{iff} & L(s) \models \Phi \text{ for all states } s \in Reach(TS). \end{array}$$

Thus, the notion ”invariant” can be explained as follows: the condition Φ has to be fulfilled by all initial states and satisfaction of Φ is invariant under all transitions in the reachable fragment of the given transition system. The latter means that if Φ holds for the source state s of a transition $s \xrightarrow{a} s'$, then Φ holds for the target state s' too.

Let us return to the examples of mutual exclusion and deadlock freedom for the dining philosophers. The mutual exclusion property can be described by an invariant using the

[5]The basic principles of propositional logic are treated in Appendix A.3.

propositional logic formula

$$\Phi \ = \ \neg crit_1 \vee \neg crit_2.$$

For deadlock freedom of the dining philosophers, the invariant ensures that at least one of the philosophers is not waiting to pick up the chopstick. This can be established using the propositional formula:

$$\Phi \ = \ \neg wait_0 \vee \neg wait_1 \vee \neg wait_2 \vee \neg wait_3 \vee \neg wait_4.$$

Here, the proposition $wait_i$ characterizes the state(s) of philosopher i in which he is waiting for a chopstick.

How do we check whether a transition system satisfies an invariant? As checking an invariant for the propositional formula Φ amounts to checking the validity of Φ in every state that is reachable from some initial state, a slight modification of standard graph traversal algorithms like depth-first search (DFS) or breadth-first search (BFS) will do, provided the given transition system TS is *finite*.

Algorithm 3 on page 109 summarizes the main steps for checking the invariant condition Φ by means of a forward depth-first search in the state graph $G(TS)$. The notion *forward search* means that we start from the initial states and investigate all states that are reachable from them. If at least one state s is visited where Φ does not hold, then the invariance induced by Φ is violated. In Algorithm 3, R stores all visited states, i.e., if Algorithm 3 terminates, then $R = Reach(TS)$ contains all reachable states. Furthermore, U is a stack that organizes all states that still have to be visited, provided they are not yet contained in R. The operations *push*, *pop*, and *top* are the standard operations on stacks. The symbol ε is used to denote the empty stack. Alternatively, a *backward search* could have been applied that starts with all states where Φ does not hold and calculates (by a DFS or BFS) the set $\bigcup_{s \in S, s \not\models \Phi} Pre^*(s)$.

Algorithm 3 could be slightly improved by aborting the computation once a state s is encountered that does not fulfill Φ. This state is a "bad" state as it makes the transition system refute the invariant and could be returned as an error indication. Such error indication, however, is not very helpful.

Instead, an initial path fragment $s_0\, s_1\, s_2 \ldots s_n$ in which all states (except the last one) satisfy Φ and $s_n \not\models \Phi$ would be more useful. Such a path fragment indicates a possible behavior of the transition system that violates the invariant. Algorithm 3 can be easily adapted such that a counterexample is provided on encountering a state that violates Φ. To that end we exploit the (depth-first search) stack U. When encountering s_n that violates Φ, the stack content, read from bottom to top, contains the required initial path fragment. Algorithm 4 on page 110 thus results.

Algorithm 3 Naïve invariant checking by forward depth-first search

Input: finite transition system TS and propositional formula Φ
Output: true if TS satisfies the invariant "always Φ", otherwise false

```
set of state $R := \varnothing$;                          (* the set of visited states *)
stack of state $U := \varepsilon$;                            (* the empty stack *)
bool $b :=$ true;                                   (* all states in $R$ satisfy $\Phi$ *)
for all $s \in I$ do
  if $s \notin R$ then
    visit($s$)                    (* perform a dfs for each unvisited initial state *)
  fi
od
return $b$
```

```
procedure visit (state $s$)
  $push(s, U)$;                                          (* push $s$ on the stack *)
  $R := R \cup \{ s \}$;                                    (* mark $s$ as reachable *)
  repeat
    $s' := top(U)$;
    if $Post(s') \subseteq R$ then
      $pop(U)$;
      $b := b \wedge (s' \models \Phi)$;                   (* check validity of $\Phi$ in $s'$ *)
    else
      let $s'' \in Post(s') \setminus R$
      $push(s'', U)$;
      $R := R \cup \{ s'' \}$;                 (* state $s''$ is a new reachable state *)
    fi
  until $(U = \varepsilon)$
endproc
```

Algorithm 4 Invariant checking by forward depth-first search

Input: finite transition system TS and propositional formula Φ
Output: "yes" if $TS \models$ "always Φ", otherwise "no" plus a counterexample

```
set of states $R := \varnothing$;                    (* the set of reachable states *)
stack of states $U := \varepsilon$;                   (* the empty stack *)
bool $b :=$ true;                                     (* all states in $R$ satisfy $\Phi$ *)
while ($I \setminus R \neq \varnothing \ \wedge\ b$) do
  let $s \in I \setminus R$;                           (* choose an arbitrary initial state not in $R$ *)
  visit($s$);                                         (* perform a DFS for each unvisited initial state *)
od
if $b$ then
  return("yes")                                       (* $TS \models$ "always $\Phi$" *)
else
  return("no", reverse($U$))                          (* counterexample arises from the stack content *)
fi
```

```
procedure visit (state $s$)
  $push(s, U)$;                                       (* push $s$ on the stack *)
  $R := R \cup \{ s \}$;                               (* mark $s$ as reachable *)
  repeat
    $s' := top(U)$;
    if $Post(s') \subseteq R$ then
      $pop(U)$;
      $b := b \ \wedge\ (s' \models \Phi)$;             (* check validity of $\Phi$ in $s'$ *)
    else
      let $s'' \in Post(s') \setminus R$
      $push(s'', U)$;
      $R := R \cup \{ s'' \}$;                         (* state $s''$ is a new reachable state *)
    fi
  until (($U = \varepsilon$) $\vee$ $\neg b$)
endproc
```

The worst-case time complexity of the proposed invariance checking algorithm is dominated by the cost for the DFS that visits all reachable states. The latter is linear in the number of states (nodes of the state graph) and transitions (edges in the state graph), provided we are given a representation of the state graph where the direct successors $s' \in \mathit{Post}(s)$ for any state s can be encountered in time $\Theta(|\mathit{Post}(s)|)$. This holds for a representation of the sets $\mathit{Post}(s)$ by adjacency lists. An explicit representation of adjacency lists is not adequate in our context where the state graph of a complex system has to be analyzed. Instead, the adjacency lists are typically given in an *implicit* way, e.g., by a syntactic description of the concurrent processes, such as program graphs or higher-level description languages with a program graph semantics such as nanoPromela, see Section 2.2.5, page 63). The direct successors of a state s are then obtained by the axioms and rules for the transition relation for the composite system. Besides the space for the syntactic descriptions of the processes, the space required by Algorithm 4 is dominated by the representation of the set R of visited states (this is typically done by appropriate hash techniques) and stack U. Hence, the additional space complexity of invariant checking is linear in the number of reachable states.

Theorem 3.21. Time Complexity of Invariant Checking

The time complexity of Algorithm 4 is $\mathcal{O}(\, N * (1 + |\Phi|) + M \,)$ where N denotes the number of reachable states, and $M = \sum_{s \in S} |\mathit{Post}(s)|$ the number of transitions in the reachable fragment of TS.

Proof: The time complexity of the forward reachability on the state graph $G(TS)$ is $\mathcal{O}(N + M)$. The time needed to check $s \models \Phi$ for some state s is linear in the length of Φ.[6] As for each state s it is checked whether Φ holds, this amounts to a total of $N + M + N * (1 + |\Phi|)$ operations. ■

3.3.2 Safety Properties

As we have seen in the previous section, invariants can be viewed as state properties and can be checked by considering the reachable states. Some safety properties, however, may impose requirements on finite path fragments, and cannot be verified by considering the reachable states only. To see this, consider the example of a cash dispenser, also known as an automated teller machine (ATM). A natural requirement is that money can only be withdrawn from the dispenser once a correct personal identifier (PIN) has been provided. This property is not an invariant, since it is not a state property. It is, however, considered

[6]To cover the special case where Φ is an atomic proposition, in which case $|\Phi| = 0$, we deal with $1 + |\Phi|$ for the cost to check whether Φ holds for a given state s.

to be a safety property, as any infinite run violating the requirement has a finite prefix that is "bad", i.e., in which money is withdrawn without issuing a PIN before.

Formally, safety property P is defined as an LT property over AP such that any infinite word σ where P does not hold contains a *bad prefix*. The latter means a finite prefix $\widehat{\sigma}$ where the bad thing has happened, and thus no infinite word that starts with this prefix $\widehat{\sigma}$ fulfills P.

Definition 3.22. Safety Properties, Bad Prefixes

An LT property P_{safe} over AP is called a *safety* property if for all words $\sigma \in (2^{AP})^\omega \setminus P_{safe}$ there exists a finite prefix $\widehat{\sigma}$ of σ such that

$$P_{safe} \cap \left\{ \sigma' \in (2^{AP})^\omega \mid \widehat{\sigma} \text{ is a finite prefix of } \sigma' \right\} \;=\; \varnothing.$$

Any such finite word $\widehat{\sigma}$ is called a *bad prefix* for P_{safe}. A *minimal* bad prefix for P_{safe} denotes a bad prefix $\widehat{\sigma}$ for P_{safe} for which no proper prefix of $\widehat{\sigma}$ is a bad prefix for P_{safe}. In other words, minimal bad prefixes are bad prefixes of minimal length. The set of all bad prefixes for P_{safe} is denoted by $BadPref(P_{safe})$, the set of all minimal bad prefixes by $MinBadPref(P_{safe})$. ■

Let us first observe that any invariant is a safety property. For propositional formula Φ over AP and its invariant P_{inv}, all finite words of the form

$$A_0\, A_1 \ldots A_n \;\in\; (2^{AP})^+$$

with $A_0 \models \Phi, \ldots, A_{n-1} \models \Phi$ and $A_n \not\models \Phi$ constitute the minimal bad prefixes for P_{inv}. The following two examples illustrate that there are safety properties that are not invariants.

Example 3.23. A Safety Property for a Traffic Light

We consider a specification of a traffic light with the usual three phases "red", "green", and "yellow". The requirement that each red phase should be immediately preceded by a yellow phase is a safety property but not an invariant. This is shown in the following.

Let *red*, *yellow*, and *green* be atomic propositions. Intuitively, they serve to mark the states describing a red (yellow or green) phase. The property "always at least one of the lights is on" is specified by:

$$\{\, \sigma = A_0\, A_1 \ldots \mid A_j \subseteq AP \;\wedge\; A_j \neq \varnothing \,\}.$$

The bad prefixes are finite words that contain $\varnothing$. A minimal bad prefix ends with $\varnothing$. The property "it is never the case that two lights are switched on at the same time" is specified by

$$\{ \sigma = A_0 A_1 \ldots \mid A_j \subseteq AP \ \wedge \ |A_j| \leqslant 1 \}.$$

Bad prefixes for this property are words containing sets such as $\{ \mathit{red}, \mathit{green} \}$, $\{ \mathit{red}, \mathit{yellow} \}$, and so on. Minimal bad prefixes end with such sets.

Now let $AP' = \{ \mathit{red}, \mathit{yellow} \}$. The property "a red phase must be preceded immediately by a yellow phase" is specified by the set of infinite words $\sigma = A_0 A_1 \ldots$ with $A_i \subseteq \{ \mathit{red}, \mathit{yellow} \}$ such that for all $i \geqslant 0$ we have that

$$\mathit{red} \in A_i \text{ implies } i > 0 \text{ and } \mathit{yellow} \in A_{i-1}.$$

The bad prefixes are finite words that violate this condition. An example of bad prefixes that are minimal is:

$$\varnothing \, \varnothing \, \{ \mathit{red} \} \quad \text{and} \quad \varnothing \, \{ \mathit{red} \}.$$

The following bad prefix is not minimal:

$$\{ \mathit{yellow} \} \{ \mathit{yellow} \} \{ \mathit{red} \} \{ \mathit{red} \} \varnothing \{ \mathit{red} \}$$

since it has a proper prefix $\{ \mathit{yellow} \} \{ \mathit{yellow} \} \{ \mathit{red} \} \{ \mathit{red} \}$ which is also a bad prefix.

The minimal bad prefixes of this safety property are regular in the sense that they constitute a regular language. The finite automaton in Figure 3.9 accepts precisely the minimal bad prefixes for the above safety property.[7] Here, $\neg \mathit{yellow}$ should be read as either $\varnothing$ or $\{ \mathit{red} \}$. Note the other properties given in this example are also regular. ■

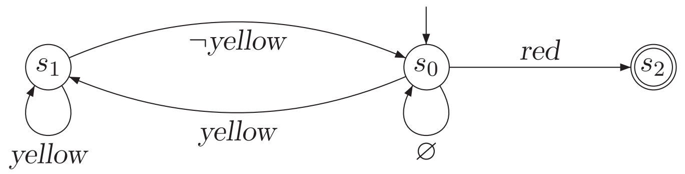


Figure 3.9: A finite automaton for the minimal bad prefixes of a regular safety property.

Example 3.24. A Safety Property for a Beverage Vending Machine

For a beverage vending machine, a natural requirement is that

"The number of inserted coins is always at least the number of dispensed drinks."

[7]The main concepts of a finite automaton as acceptors for languages over finite words are summarized in Section 4.1.

Using the set of propositions $\{ \mathit{pay}, \mathit{drink} \}$ and the obvious labeling function, this property could be formalized by the set of infinite words $A_0\, A_1\, A_2 \ldots$ such that for all $i \geqslant 0$ we have

$$|\{ 0 \leqslant j \leqslant i \mid \mathit{pay} \in A_j \}| \;\geqslant\; |\{ 0 \leqslant j \leqslant i \mid \mathit{drink} \in A_j \}|$$

Bad prefixes for this safety property are, for example

$$\begin{array}{l} \varnothing \{ \mathit{pay} \} \{ \mathit{drink} \} \{ \mathit{drink} \} \quad \text{and} \\ \varnothing \{ \mathit{pay} \} \{ \mathit{drink} \} \varnothing \{ \mathit{pay} \} \{ \mathit{drink} \} \{ \mathit{drink} \} \end{array}$$

It is left to the interested reader to check that both beverage vending machines from Figure 3.8 satisfy the above safety property. ■

Safety properties are requirements for the finite traces which is formally stated in the following lemma:

Lemma 3.25. ***Satisfaction Relation for Safety Properties***

For transition system TS without terminal states and safety property P_{safe}:

$$\mathit{TS} \models P_{\mathit{safe}} \;\; \textit{if and only if} \;\; \mathit{Traces}_{\mathit{fin}}(\mathit{TS}) \cap \mathit{BadPref}(P_{\mathit{safe}}) = \varnothing.$$

Proof: "if": By contradiction. Let $\mathit{Traces}_{\mathit{fin}}(\mathit{TS}) \cap \mathit{BadPref}(P_{\mathit{safe}}) = \varnothing$ and assume that $\mathit{TS} \not\models P_{\mathit{safe}}$. Then, $\mathit{trace}(\pi) \notin P_{\mathit{safe}}$ for some path π in TS. Thus, $\mathit{trace}(\pi)$ starts with a bad prefix $\widehat{\sigma}$ for P_{safe}. But then, $\widehat{\sigma} \in \mathit{Traces}_{\mathit{fin}}(\mathit{TS}) \cap \mathit{BadPref}(P_{\mathit{safe}})$. Contradiction.

"only if": By contradiction. Let $\mathit{TS} \models P_{\mathit{safe}}$ and assume that $\widehat{\sigma} \in \mathit{Traces}_{\mathit{fin}}(\mathit{TS}) \cap \mathit{BadPref}(P_{\mathit{safe}})$. The finite trace $\widehat{\sigma} = A_1 \ldots A_n \in \mathit{Traces}_{\mathit{fin}}(\mathit{TS})$ can be extended to an infinite trace $\sigma = A_1 \ldots A_n\, A_{n+1}\, A_{n+2} \ldots \in \mathit{Traces}(\mathit{TS})$. Then, $\sigma \notin P_{\mathit{safe}}$ and thus, $\mathit{TS} \not\models P_{\mathit{safe}}$. ■

We conclude this section with an alternative characterization of safety properties by means of their closure.

Definition 3.26. Prefix and Closure

For trace $\sigma \in (2^{AP})^\omega$, let $\mathit{pref}(\sigma)$ denote the set of finite prefixes of σ, i.e.,

$$\mathit{pref}(\sigma) = \{ \widehat{\sigma} \in (2^{AP})^* \mid \widehat{\sigma} \text{ is a finite prefix of } \sigma \}.$$

that is, if $\sigma = A_0\, A_1 \ldots$ then $pref(\sigma) = \{\varepsilon, A_0, A_0A_1, A_0A_1A_2, \ldots\}$ is an infinite set of finite words. This notion is lifted to sets of traces in the usual way. For property P over AP:

$$pref(P) = \bigcup_{\sigma \in P} pref(\sigma).$$

The *closure* of LT property P is defined by

$$closure(P) \;=\; \{\sigma \in (2^{AP})^{\omega} \mid pref(\sigma) \;\subseteq\; pref(P)\}.$$

■

For instance, for infinite trace $\sigma = ABABAB\ldots$ (where $A, B \subseteq AP$) we have $pref(\sigma) = \{\,\varepsilon, A, AB, ABA, ABAB, \ldots\,\}$ which equals the regular language given by the regular expression $(AB)^*(A + \varepsilon)$.

The *closure* of an LT property P is the set of infinite traces whose finite prefixes are also prefixes of P. Stated differently, infinite traces in the closure of P do not have a prefix that is not a prefix of P itself. As we will see below, the closure is a key concept in the characterization of safety and liveness properties.

Lemma 3.27. Alternative Characterization of Safety Properties

Let P be an LT property over AP. Then, P is a safety property iff $closure(P) \;=\; P$.

Proof: "if": Let us assume that $closure(P) \;=\; P$. To show that P is a safety property, we take an element $\sigma \in (2^{AP})^{\omega} \setminus P$ and show that σ starts with a bad prefix for P. Since $\sigma \notin P = closure(P)$ there exists a finite prefix $\widehat{\sigma}$ of σ with $\widehat{\sigma} \notin pref(P)$. By definition of $pref(P)$, none of the words $\sigma' \in (2^{AP})^{\omega}$ where $\widehat{\sigma} \in pref(\sigma')$ belongs to P. Hence, $\widehat{\sigma}$ is a bad prefix for P, and by definition, P is a safety property.

"only if": Let us assume that P is a safety property. We have to show that $P = closure(P)$. The inclusion $P \subseteq closure(P)$ holds for all LT properties. It remains to show that $closure(P) \subseteq P$. We do so by contradiction. Let us assume that there is some $\sigma = A_1\, A_2 \ldots \in closure(P) \setminus P$. Since P is a safety property and $\sigma \notin P$, σ has a finite prefix

$$\widehat{\sigma} = A_1 \ldots A_n \in BadPref(P).$$

As $\sigma \in closure(P)$ we have $\widehat{\sigma} \in pref(\sigma) \subseteq pref(P)$. Hence, there exists a word $\sigma' \in P$ of the form

$$\sigma' = \underbrace{A_1 \ldots A_n}_{\text{bad prefix}} B_{n+1}\, B_{n+2} \ldots$$

This contradicts the fact that P is a safety property. ■

3.3.3 Trace Equivalence and Safety Properties

We have seen before that there is a strong relationship between trace inclusion of transition systems and the satisfaction of LT properties (see Theorem 3.15, page 104):

$$\mathit{Traces}(TS) \subseteq \mathit{Traces}(TS') \quad \text{if and only if} \quad \begin{array}{l}\text{for all LT properties } P\text{:}\\ TS' \models P \text{ implies } TS \models P\end{array}$$

for transition systems TS and TS' without terminal states. Note that this result considers all *infinite* traces. The above thus states a relationship between infinite traces of transition systems and the validity of LT properties. When considering only finite traces instead of infinite ones, a similar connection with the validity of safety properties can be established, as stated by the following theorem.

Theorem 3.28. Finite Trace Inclusion and Safety Properties

Let TS and TS′ be transition systems without terminal states and with the same set of propositions AP. Then the following statements are equivalent:

(a) $\mathit{Traces}_{\mathit{fin}}(TS) \subseteq \mathit{Traces}_{\mathit{fin}}(TS')$,

(b) For any safety property P_{safe}: $TS' \models P_{\mathit{safe}}$ *implies* $TS \models P_{\mathit{safe}}$.

Proof:

(a) $\Longrightarrow$ (b): Let us assume that $\mathit{Traces}_{\mathit{fin}}(TS) \subseteq \mathit{Traces}_{\mathit{fin}}(TS')$ and let P_{safe} be a safety property with $TS' \models P_{\mathit{safe}}$. By Lemma 3.25, we have $\mathit{Traces}_{\mathit{fin}}(TS') \cap \mathit{BadPref}(P_{\mathit{safe}}) = \varnothing$, and hence, $\mathit{Traces}_{\mathit{fin}}(TS) \cap \mathit{BadPref}(P_{\mathit{safe}}) = \varnothing$. Again by Lemma 3.25, we get $TS \models P_{\mathit{safe}}$.

(b) $\Longrightarrow$ (a): Assume that (b) holds. Let $P_{\mathit{safe}} = \mathit{closure}(\mathit{Traces}(TS'))$. Then, P_{safe} is a safety property and we have $TS' \models P_{\mathit{safe}}$ (see Exercise 3.9, page 147). Hence, (b) yields $TS \models P_{\mathit{safe}}$, i.e.,

$$\mathit{Traces}(TS) \subseteq \mathit{closure}(\mathit{Traces}(TS')).$$

From this, we may derive

$$\begin{array}{rcl}\mathit{Traces}_{\mathit{fin}}(TS) & = & \mathit{pref}(\mathit{Traces}(TS))\\ & \subseteq & \mathit{pref}(\mathit{closure}(\mathit{Traces}(TS'))\\ & = & \mathit{pref}(\mathit{Traces}(TS'))\\ & = & \mathit{Traces}_{\mathit{fin}}(TS').\end{array}$$

Here we use the property that for any P it holds that $\mathit{pref}(\mathit{closure}(P)) = \mathit{pref}(P)$ (see Exercise 3.10, page 147). ■

Theorem 3.28 is of relevance for the gradual design of concurrent systems. If a preliminary design (i.e., a transition system) TS' is refined to a design TS such that

$$Traces(TS) \not\subseteq Traces(TS'),$$

then the LT properties of TS' cannot be carried over to TS. However, if the finite traces of TS are finite traces of TS' (which is a weaker requirement than full trace inclusion of TS and TS'), i.e.,

$$Traces_{fin}(TS) \subseteq Traces_{fin}(TS'),$$

then all safety properties that have been established for TS' also hold for TS. Other requirements for TS, i.e., LT properties that fall outside the scope of safety properties, need to be checked using different techniques.

Corollary 3.29. **Finite Trace Equivalence and Safety Properties**

Let TS and TS' be transition systems without terminal states and with the same set AP of atomic propositions. Then, the following statements are equivalent:

(a) $Traces_{fin}(TS) = Traces_{fin}(TS')$,

(b) For any safety property P_{safe} over AP: $TS \models P_{safe} \Longleftrightarrow TS' \models P_{safe}$.

A few remarks on the difference between finite trace inclusion and trace inclusion are in order. Since we assume transition systems without terminal states, there is only a slight difference between trace inclusion and finite trace inclusion. For *finite* transition systems TS and TS' without terminal states, trace inclusion and finite trace inclusion coincide. This can be derived from the following theorem.

Theorem 3.30. **Relating Finite Trace and Trace Inclusion**

Let TS and TS' be transition systems with the same set AP of atomic propositions such that TS has no terminal states and TS' is finite. Then:

$$Traces(TS) \subseteq Traces(TS') \quad \Longleftrightarrow \quad Traces_{fin}(TS) \subseteq Traces_{fin}(TS').$$

Proof: The implication from left to right follows from the monotonicity of $pref(\cdot)$ and the fact that $Traces_{fin}(TS) = pref(Traces(TS))$ for any transition system TS.

It remains to consider the proof for the implication $\Longleftarrow$. Let us assume that $Traces_{fin}(TS) \subseteq Traces_{fin}(TS')$. As TS has no terminal states, all traces of TS are infinite. Let $A_0 A_1 \ldots \in$

Traces(TS). To prove that $A_0 A_1 \ldots \in$ *Traces*(TS') we have to show that there exists a path in TS', say $s_0\, s_1 \ldots$, that generates this trace, i.e., $\mathit{trace}(s_0\, s_1 \ldots) = A_0\, A_1 \ldots$.

Any finite prefix $A_0\, A_1 \ldots A_m$ of the infinite trace $A_0 A_1 \ldots$ is in $\mathit{Traces}_{\mathit{fin}}(TS)$, and as $\mathit{Traces}_{\mathit{fin}}(TS) \subseteq \mathit{Traces}_{\mathit{fin}}(TS')$, also in $\mathit{Traces}_{\mathit{fin}}(TS')$. Thus, for any natural number m, there exists a finite path $\pi^m = s_0^m s_1^m \ldots s_m^m$ in TS' such that

$$\mathit{trace}(\pi^m) \;=\; L(s_0^m)L(s_1^m)\ldots L(s_m^m) = A_0 A_1 \ldots A_m$$

where L denotes the labeling function of TS'. Thus, $L(s_j^m) = A_j$ for all $0 \leqslant j \leqslant m$.

Although $A_0 \ldots A_m$ is a prefix of $A_0 \ldots A_{m+1}$, it is not guaranteed that path π^m is a prefix of π^{m+1}. Due to the finiteness of TS', however, there is an infinite subsequence $\pi^{m_0}\, \pi^{m_1}\, \pi^{m_2} \ldots$ of $\pi^0\, \pi^1\, \pi^2 \ldots$ such that π^{m_i} and $\pi^{m_{i+1}}$ agree on the first i states. Thus, $\pi^{m_0}\, \pi^{m_1}\, \pi^{m_2} \ldots$ induces an infinite path π in TS' with the desired property.

This is formally proven using a so-called *diagonalization technique*. This goes as follows. Let $I_0, I_1, I_2, \ldots$ be an infinite series of infinite sets of indices (i.e., natural numbers) with $I_n \subseteq \{\, m \in \mathbb{N} \mid m \geqslant n \,\}$ and $s_0, s_1, \ldots$ be states in TS' such that for all natural numbers n it holds that

(1) $n \geqslant 1$ implies $I_{n-1} \supseteq I_n$,

(2) $s_0\, s_1\, s_2\, \ldots\, s_n$ is an initial, finite path fragment in TS',

(3) for all $m \in I_n$ it holds that $s_0\, \ldots\, s_n = s_0^m\, \ldots\, s_n^m$.

The definition of the sets I_n and states s_n is by induction on n.

Base case ($n = 0$): As $\{\, s_0^m \mid m \in \mathbb{N} \,\}$ is finite (since it is a subset of the finite set of initial states of TS'), there exists an initial state s_0 in TS' and an infinite index set I_0 such that $s_0 = s_0^m$ for all $m \in I_0$.

Induction step $n \Longrightarrow n{+}1$. Assume that the index sets $I_0, \ldots, I_n$ and states $s_0, \ldots, s_n$ are defined. Since TS' is finite, $\mathit{Post}(s_n)$ is finite. Furthermore, by the induction hypothesis $s_n = s_n^m$ for all $m \in I_n$, and thus

$$\{\, s_{n+1}^m \mid m \in I_n, m \geqslant n{+}1 \,\} \;\subseteq\; \mathit{Post}(s_n).$$

Since I_n is infinite, there exists an infinite subset $I_{n+1} \subseteq \{\, m \in I_n \mid m \geqslant n{+}1 \,\}$ and a state $s_{n+1} \in \mathit{Post}(s_n)$ such that $s_{n+1}^m = s_{n+1}$ for all $m \in I_{n+1}$. It follows directly that the above properties (1) through (3) are fulfilled.

We now consider the state sequence $s_0\, s_1\, \ldots$ in TS'. Obviously, this state sequence is a path in TS' satisfying $trace(s_0\, s_1\, \ldots) = A_0\, A_1\, \ldots$. Consequently, $A_0\, A_1\, \ldots \in Traces(TS')$. ■

Remark 3.31. Image-Finite Transition Systems

The result stated in Theorem 3.30 also holds under slightly weaker conditions: it suffices to require that TS has no terminal states (as in Theorem 3.30) and that TS' is AP image-finite (rather than being finite).

Let $TS' = (S, Act, \rightarrow, I, AP, L)$. Then, TS' is called AP image-finite (or briefly *image-finite*) if

(i) for all $A \subseteq AP$, the set $\{\, s_0 \in I \mid L(s_0) = A \,\}$ is finite and

(ii) for all states s in TS' and all $A \subseteq AP$, the set of successors $\{\, s' \in Post(s) \mid L(s') = A \,\}$ is finite.

Thus, any finite transition system is image-finite. Moreover, any transition system that is AP-deterministic is image-finite. (Recall that AP-determinism requires $\{\, s_0 \in I \mid L(s_0) = A \,\}$ and $\{\, s' \in Post(s) \mid L(s') = A \,\}$ to be either singletons or empty sets; see Definition 2.5, page 24.)

In fact, a careful inspection of the proof of Theorem 3.30 shows that (i) and (ii) for TS' are used in the construction of the index sets I_n and states s_n. Hence, we have $Traces(TS) \subseteq Traces(TS')$ iff $Traces_{fin}(TS) \subseteq Traces_{fin}(TS')$, provided TS has no terminal states and TS' is image-finite. ■

Trace and finite trace inclusion, however, coincide neither for infinite transition systems nor for finite ones which have terminal states.

Example 3.32. Finite vs. Infinite Transition System

Consider the transition systems sketched in Figure 3.10, where b stands for an atomic proposition. Transition system TS (on the left) is finite, whereas TS' (depicted on the right) is infinite and not image-finite, because of the infinite branching in the initial state. It is not difficult to observe that

$$Traces(TS) \not\subseteq Traces(TS') \quad \text{and} \quad Traces_{fin}(TS) \subseteq Traces_{fin}(TS').$$

This stems from the fact that TS can take the self-loop infinitely often and never reaches a b-state, whereas TS' does not exhibit such behavior. Moreover, any finite trace of TS is of the form $(\varnothing)^n$ for $n \geqslant 0$ and is also a finite trace of TS'. Consequently, LT properties of TS' do not carry over to TS (and those of TS may not hold for TS'). For example, the LT property "eventually b" holds for TS', but not for TS. Similarly, the LT property "never b" holds for TS, but not for TS'.

Although these transition systems might seem rather artificial, this is not the case: TS could result from an infinite loop in a program, whereas TS' could model the semantics of a program fragment that nondeterministically chooses a natural number k and then performs k steps. ■

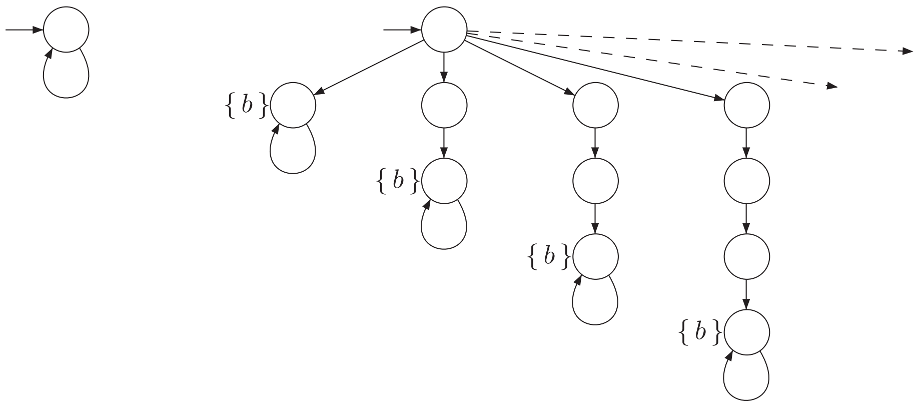


Figure 3.10: Distinguishing trace inclusion from finite trace inclusion.

3.4 Liveness Properties

Informally speaking, safety properties specify that "something bad never happens". For the mutual exclusion algorithm, the "bad" thing is that more than one process is in its critical section, while for the traffic light the "bad" situation is whenever a red light phase is not preceded by a yellow light phase. An algorithm can easily fulfill a safety property by simply doing nothing as this will never lead to a "bad" situation. As this is usually undesired, safety properties are complemented by properties that require some progress. Such properties are called "liveness" properties (or sometimes "progress" properties). Intuitively, they state that "something good" will happen in the future. Whereas safety properties are violated in finite time, i.e., by a finite system run, liveness properties are violated in infinite time, i.e., by infinite system runs.

3.4.1 Liveness Properties

Several (nonequivalent) notions of liveness properties have been defined in the literature. We follow here the approach of Alpern and Schneider [5, 6, 7]. They provided a formal notion of liveness properties which relies on the view that liveness properties do not constrain the finite behaviors, but require a certain condition on the infinite behaviors. A typical example for a liveness property is the requirement that certain events occur infinitely often. In this sense, the "good event" of a liveness property is a condition on the infinite behaviors, while the "bad event" for a safety property occurs in a finite amount of time, if it occurs at all.

In our approach, a liveness property (over AP) is defined as an LT property that does not rule out any prefix. This entails that the set of finite traces of a system are of no use at all to decide whether a liveness property holds or not. Intuitively speaking, it means that any finite prefix can be extended such that the resulting infinite trace satisfies the liveness property under consideration. This is in contrast to safety properties where it suffices to have one finite trace (the "bad prefix") to conclude that a safety property is refuted.

Definition 3.33. Liveness Property

LT property P_{live} over AP is a *liveness* property whenever $pref(P_{live}) = (2^{AP})^*$. ■

Thus, a liveness property (over AP) is an LT property P such that each finite word can be extended to an infinite word that satisfies P. Stated differently, P is a liveness property if and only if for all finite words $w \in (2^{AP})^*$ there exists an infinite word $\sigma \in (2^{AP})^\omega$ satisfying $w\sigma \in P$.

Example 3.34. Repeated Eventually and Starvation Freedom

In the context of mutual exclusion algorithms the natural safety property that is required ensures the mutual exclusion property stating that the processes are never simultaneously in their critical sections. (This is even an invariant.) Typical liveness properties that are desired assert that

- (eventually) each process will eventually enter its critical section;
- (repeated eventually) each process will enter its critical section infinitely often;
- (starvation freedom) each waiting process will eventually enter its critical section.

Let's see how these liveness properties are formalized as LT properties and let us check that they are liveness properties. As in Example 3.14, we will deal with the atomic propositions $wait_1, crit_1, wait_2, crit_2$ where $wait_i$ characterizes the states where process P_i has requested access to its critical section and is in its waiting phase, while $crit_i$ serves as a label for the states where P_i has entered its critical section. We now formalize the three properties by LT properties over $AP = \{wait_1, crit_1, wait_2, crit_2\}$. The first property (eventually) consists of all infinite words $A_0\,A_1\,\ldots$ with $A_j \subseteq AP$ such that

$$(\exists j \geqslant 0.\ crit_1 \in A_j) \ \wedge\ (\exists j \geqslant 0.\ crit_2 \in A_j)$$

which requires that P_1 and P_2 are in their critical sections at least once. The second property (repeated eventually) poses the condition

$$(\forall k \geqslant 0. \exists j \geqslant k.\ crit_1 \in A_j) \ \wedge\ (\forall k \geqslant 0. \exists j \geqslant k.\ crit_2 \in A_j)$$

stating that P_1 and P_2 are infinitely often in their critical sections. This formula is often abbreviated by

$$\left(\overset{\infty}{\exists}\ j \geqslant 0.\ crit_1 \in A_j\right) \ \wedge\ \left(\overset{\infty}{\exists}\ j \geqslant 0.\ crit_2 \in A_j\right).$$

The third property (starvation freedom) requires that

$$\begin{aligned}
&\forall j \geqslant 0.\ (wait_1 \in A_j \ \Rightarrow\ (\exists k > j.\, crit_1 \in A_k)) \ \wedge \\
&\forall j \geqslant 0.\ (wait_2 \in A_j \ \Rightarrow\ (\exists k > j.\, crit_2 \in A_k)).
\end{aligned}$$

It expresses that each process that is waiting will acquire access to the critical section at some later time point. Note that here we implicitly assume that a process that starts waiting to acquire access to the critical section does not "give up" waiting, i.e., it continues waiting until it is granted access.

All aforementioned properties are liveness properties, as any finite word over AP is a prefix of an infinite word where the corresponding condition holds. For instance, for starvation freedom, a finite trace in which a process is waiting but never acquires access to its critical section can always be extended to an infinite trace that satisfies the starvation freedom property (by, e.g., providing access in an strictly alternating fashion from a certain point on). ■

3.4.2 Safety vs. Liveness Properties

This section studies the relationship between liveness and safety properties. In particular, it provides answers to the following questions:

- Are safety and liveness properties disjoint?, and

- Is any linear-time property a safety or liveness property?

As we will see, the first question will be answered affirmatively while the second question will result in a negative answer. Interestingly enough, though, for any LT property P an equivalent LT property P' does exist which is a combination (i.e., intersection) of a safety and a liveness property. All in all, one could say that the identification of safety and liveness properties thus provides an essential characterization of linear-time properties.

The first result states that safety and liveness properties are indeed almost disjoint. More precisely, it states that the only property that is both a safety and a liveness property is nonrestrictive, i.e., allows all possible behaviors. Logically speaking, this is the equivalent of "true".

Lemma 3.35. Intersection of Safety and Liveness Properties

The only LT property over AP that is both a safety and a liveness property is $(2^{AP})^\omega$.

Proof: Assume P is a liveness property over AP. By definition, $\mathit{pref}(P) = (2^{AP})^*$. It follows that $\mathit{closure}(P) = (2^{AP})^\omega$. If P is a safety property too, $\mathit{closure}(P) = P$, and hence $P = (2^{AP})^\omega$. ■

Recall that the closure of property P (over AP) is the set of infinite words (over 2^{AP}) for which all prefixes are also prefixes of P. In order to show that an LT property can be considered as a conjunction of a liveness and a safety property, the following result is helpful. It states that the closure of the union of two properties equals the union of their closures.

Lemma 3.36. Distributivity of Union over Closure

For any LT properties P and P':

$$\mathit{closure}(P) \cup \mathit{closure}(P') = \mathit{closure}(P \cup P').$$

Proof: $\subseteq$: As $P \subseteq P'$ implies $\mathit{closure}(P) \subseteq \mathit{closure}(P')$, we have $P \subseteq P \cup P'$ implies $\mathit{closure}(P) \subseteq \mathit{closure}(P \cup P')$. In a similar way it follows that $\mathit{closure}(P') \subseteq \mathit{closure}(P \cup P')$. Thus, $\mathit{closure}(P) \cup \mathit{closure}(P') \subseteq \mathit{closure}(P \cup P')$.

$\supseteq$: Let $\sigma \in \mathit{closure}(P \cup P')$. By definition of closure, $\mathit{pref}(\sigma) \subseteq \mathit{pref}(P \cup P')$. As $\mathit{pref}(P \cup P') = \mathit{pref}(P) \cup \mathit{pref}(P')$, any finite prefix of σ is in $\mathit{pref}(P)$ or in $\mathit{pref}(P')$ (or in both).

As $\sigma \in (2^{AP})^\omega$, σ has infinitely many prefixes. Thus, infinitely many finite prefixes of σ belong to $\mathit{pref}(P)$ or to $\mathit{pref}(P')$ (or to both). W.l.o.g., assume $\mathit{pref}(\sigma) \cap \mathit{pref}(P)$ to be infinite. Then $\mathit{pref}(\sigma) \subseteq \mathit{pref}(P)$, which yields $\sigma \in \mathit{closure}(P)$, and thus $\sigma \in \mathit{closure}(P) \cup \mathit{closure}(P')$. The fact that $\mathit{pref}(\sigma) \subseteq \mathit{pref}(P)$ can be shown by contraposition. Assume $\widehat{\sigma} \in \mathit{pref}(\sigma) \setminus \mathit{pref}(P)$. Let $|\widehat{\sigma}| = k$. As $\mathit{pref}(\sigma) \cap \mathit{pref}(P)$ is infinite, there exists $\widehat{\sigma'} \in \mathit{pref}(\sigma) \cap \mathit{pref}(P)$ with length larger than k. But then, there exists $\sigma' \in P$ with $\widehat{\sigma'} \in \mathit{pref}(\sigma')$. It then follows that $\widehat{\sigma} \in \mathit{pref}(\widehat{\sigma'})$ (as both $\widehat{\sigma}$ and $\widehat{\sigma'}$ are prefixes of σ) and as $\mathit{pref}(\widehat{\sigma'}) \subseteq \mathit{pref}(P)$, it follows that $\widehat{\sigma} \in \mathit{pref}(P)$. This contradicts $\widehat{\sigma} \in \mathit{pref}(\sigma) \setminus \mathit{pref}(P)$. ■

Consider the beverage vending machine of Figure 3.5 (on page 97), and the following property:

"the machine provides beer infinitely often
after initially providing soda three times in a row"

In fact, this property consists of two parts. On the one hand, it requires beer to be provided infinitely often. As any finite trace can be extended to an infinite trace that enjoys this property it is a liveness property. On the other hand, the first three drinks it provides should all be soda. This is a safety property, since any finite trace in which one of the first three drinks provided is beer violates it. The property is thus a combination (in fact, a conjunction) of a safety and a liveness property. The following result shows that *every* LT property can be decomposed in this way.

Theorem 3.37. Decomposition Theorem

For any LT property P over AP there exists a safety property P_{safe} and a liveness property P_{live} (both over AP) such that

$$P = P_{safe} \cap P_{live}.$$

Proof: Let P be an LT property over AP. It is easy to see that $P \subseteq \mathit{closure}(P)$. Thus: $P = \mathit{closure}(P) \cap P$, which by set calculus can be rewritten into:

$$P = \underbrace{\mathit{closure}(P)}_{=P_{safe}} \cap \underbrace{\Big(P \cup \Big((2^{AP})^\omega \setminus \mathit{closure}(P)\Big)\Big)}_{=P_{live}}$$

By definition, $P_{safe} = \mathit{closure}(P)$ is a safety property. It remains to prove that $P_{live} = P \cup \Big((2^{AP})^\omega \setminus \mathit{closure}(P)\Big)$ is a liveness property. By definition, P_{live} is a liveness property whenever $\mathit{pref}(P_{live}) = (2^{AP})^*$. This is equivalent to $\mathit{closure}(P_{live}) = (2^{AP})^\omega$. As for any

LT property P, $closure(P) \subseteq (2^{AP})^\omega$ holds true, it suffices to showing that $(2^{AP})^\omega \subseteq closure(P_{live})$. This goes as follows:

$$\begin{array}{rcl} closure(P_{live}) & = & closure\Big(P \cup ((2^{AP})^\omega \setminus closure(P))\Big) \\ & \overset{\text{Lemma 3.36}}{=} & closure(P) \cup closure\Big((2^{AP})^\omega \setminus closure(P)\Big) \\ & \supseteq & closure(P) \cup \Big((2^{AP})^\omega \setminus closure(P)\Big) \\ & = & (2^{AP})^\omega \end{array}$$

where in the one-but-last step in the derivation, we exploit the fact that $closure(P') \supseteq P'$ for all LT properties P'. ■

The proof of Theorem 3.37 shows that $P_{safe} = closure(P)$ is a safety property and $P_{live} = P \cup ((2^{AP})^\omega \setminus closure(P))$ a liveness property with $P = P_{safe} \cap P_{live}$. In fact, this decomposition is the "sharpest" one for P since P_{safe} is the *strongest* safety property and P_{live} the *weakest* liveness property that can serve for a decomposition of P:

Lemma 3.38. Sharpest Decomposition

Let P be an LT property and $P = P_{safe} \cap P_{live}$ where P_{safe} is a safety property and P_{live} a liveness property. We then have

1. $closure(P) \subseteq P_{safe}$,
2. $P_{live} \subseteq P \cup ((2^{AP})^\omega \setminus closure(P))$.

Proof: See Exercise 3.12, page 147. ■

A summary of the classification of LT properties is depicted as a Venn diagram in Figure 3.11. The circle denotes the set of all LT properties over a given set of atomic propositions.

Remark 3.39. Topological Characterizations of Safety and Liveness

Let us conclude this section with a remark for readers who are familiar with basic notions of topological spaces. The set $(2^{AP})^\omega$ can be equipped with the distance function given by $d(\sigma_1, \sigma_2) = 1/2^n$ if σ_1, σ_2 are two distinct infinite words $\sigma_1 = A_1 A_2 \ldots$ and $\sigma_2 = B_1 B_2 \ldots$ and n is the length of the longest common prefix. Moreover, we put $d(\sigma, \sigma) = 0$. Then, d is a metric on $(2^{AP})^\omega$, and hence induces a topology on $(2^{AP})^\omega$. Under this topology,

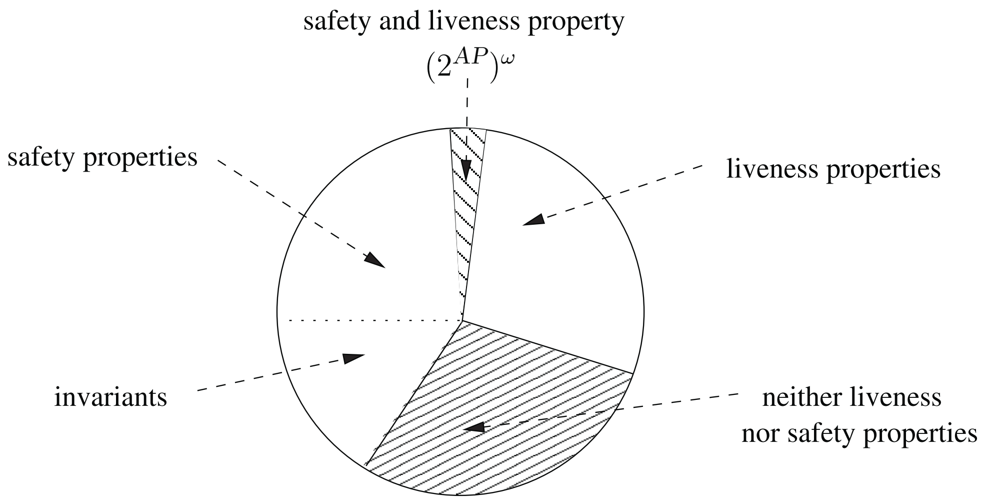


Figure 3.11: Classification of linear-time properties.

the safety properties are exactly the closed sets, while the liveness properties agree with the dense sets. In fact, *closure*(P) is the topological closure of P, i.e., the smallest closed set that contains P. The result stated in Theorem 3.37 then follows from the well-known fact that any subset of a topological space (of the kind described above) can be written as the intersection of its closure and a dense set. ■

3.5 Fairness

An important aspect of reactive systems is fairness. Fairness assumptions rule out infinite behaviors that are considered unrealistic, and are often necessary to establish liveness properties. We illustrate the concept of fairness by means of a frequently encountered problem in concurrent systems.

Example 3.40. Process Fairness

Consider N processes $P_1, \ldots, P_N$ which require a certain service. There is one server process *Server* that is expected to provide services to these processes. A possible strategy that *Server* can realize is the following. Check the processes starting with P_1, then P_2, and so on, and serve the first thus encountered process that requires service. On finishing serving this process, repeat this selection procedure once again starting with checking P_1.

Now suppose that P_1 is continuously requesting service. Then this strategy will result in *Server* always serving P_1. Since in this way another process has to wait infinitely long before being served, this is called an unfair strategy. In a fair serving strategy it is required that the server eventually responds to any request by any one of the processes. For instance, a round-robin scheduling strategy where each process is only served for a limited amount of time is a fair strategy: after having served one process, the next (in the round-robin order) is checked and, if needed, served. ■

When verifying concurrent systems one is often only interested in paths in which enabled transitions (statements) are executed in some "fair" manner. Consider, for instance, a mutual exclusion algorithm for two processes. In order to prove starvation freedom, the situation in which a process that wants to enter its critical section has to wait infinitely long, we want to exclude those paths in which the competitor process is always being selected for execution. This type of fairness is also known as *process fairness*, since it concerns the fair scheduling of the execution of processes. If we were to consider unfair paths when proving starvation freedom, we would usually fail, since there always exists an unfair strategy according to which some process is always neglected, and thus can never make progress. One might argue that such unfair strategy is unrealistic and should be avoided.

Example 3.41. Starvation Freedom

Consider the transition systems TS_{Sem} and TS_{Pet} for the semaphore-based mutual exclusion algorithms (see Example 2.24 on page 43) and Peterson's algorithm. The starvation freedom property

"Once access is requested, a process does not have to wait infinitely long before acquiring access to its critical section"

is violated by transition system TS_{Sem} while it permits only one of the processes to proceed, while the other process is starving (or only acquiring access to the critical section finitely often). The transition system TS_{Pet} for Peterson's algorithm, however, fulfills this property.

The property

"Each of the processes is infinitely often in its critical section"

is violated by both transition systems as none of them excludes the fact that a process would never (or only finitely often) request to enter the critical section. ■

Process fairness is a particular form of fairness. In general, fairness assumptions are needed to prove liveness or other properties stating that the system makes some progress ("something good will eventually happen"). This is of vital importance if the transition system to be checked contains nondeterminism. Fairness is then concerned with resolving nondeterminism in such a way that it is not biased to consistently ignore a possible option. In the above example, the scheduling of processes is nondeterministic: the choice of the next process to be executed (if there are at least two processes that can be potentially selected) is arbitrary. Another prominent example where fairness is used to "resolve" nondeterminism is in modeling concurrent processes by means of interleaving. Interleaving is equivalent to modeling the concurrent execution of two independent processes by enumerating all the possible orders in which activities of the processes can be executed (see Chapter 2).

Example 3.42. Independent Traffic Lights

Consider the transition system

$$TS = \mathit{TrLight}_1 \;|||\; \mathit{TrLight}_2$$

for the two independent traffic lights described in Example 2.17 (page 36). The liveness property

"Both traffic lights are infinitely often green"

is not satisfied, since

$$\{\, \mathit{red}_1, \mathit{red}_2 \,\}\, \{\, \mathit{green}_1, \mathit{red}_2 \,\}\, \{\, \mathit{red}_1, \mathit{red}_2 \,\}\, \{\, \mathit{green}_1, \mathit{red}_2 \,\} \ldots$$

is a trace of TS where only the first traffic light is infinitely often green. ■

What is wrong with the above examples? In fact, nothing. Let us explain this. In the traffic light example, the information whether each traffic light switches color infinitely often is lost by means of interleaving. The trace in which only the first traffic light is acting while the second light seems to be completely stopped is formally a trace of the transition system $\mathit{TrLight}_1 \,|||\, \mathit{TrLight}_2$. However, it does not represent a realistic behavior as in practice no traffic light is infinitely faster than another.

For the semaphore-based mutual exclusion algorithm, the difficulty is the degree of abstraction. A semaphore is not a willful individual that arbitrarily chooses a process which is authorized to enter its critical section. Instead, the waiting processes are administered in a queue (or another "fair" medium). The required liveness can be proven in one of the following refinement steps, in which the specification of the behavior of the semaphore is sufficiently detailed.

3.5.1 Fairness Constraints

The above considerations show that we—to obtain a realistic picture of the behavior of a parallel system modeled by a transition system—need a more alleviated form of satisfaction relation for LT properties, which implies an "adequate" resolution of the nondeterministic decisions in a transition system. In order to rule out the unrealistic computations, *fairness constraints* are imposed.

In general, a fair execution (or trace) is characterized by the fact that certain fairness constraints are fulfilled. Fairness constraints are used to rule out computations that are considered to be unreasonable for the system under consideration. Fairness constraints come in different flavors:

- *Unconditional fairness*: e.g.,"Every process gets its turn infinitely often."
- *Strong fairness*: e.g., "Every process that is enabled infinitely often gets its turn infinitely often."
- *Weak fairness:* e.g., "Every process that is continuously enabled from a certain time instant on gets its turn infinitely often."

Here, the term "is enabled" has to be understood in the sense of "ready to execute (a transition)". Similarly, "gets its turn" stands for the execution of an arbitrary transition. This can, for example, be a noncritical action, acquiring a shared resource, an action in the critical section, or a communication action.

An execution fragment is unconditionally fair with respect to, e.g., "a process enters its critical section" or "a process gets its turn", if these properties hold infinitely often. That is to say, a process enters its critical section infinitely often, or, in the second example, a process gets its turn infinitely often. Note that no condition (such as "a process is enabled") is expressed that constrains the circumstances under which a process gets its turn infinitely often. Unconditional fairness is sometimes referred to as impartiality.

Strong fairness means that if an activity is infinitely often enabled—but not necessarily always, i.e., there may be finite periods during which the activity is not enabled—then it will be executed infinitely often. An execution fragment is strongly fair with respect to activity α if it is not the case that α is infinitely often enabled without being taken beyond a certain point. Strong fairness is sometimes referred to as compassion.

Weak fairness means that if an activity, e.g., a transition in a process or an entire process itself, is continuously enabled—no periods are allowed in which the activity is not enabled—then it has to be executed infinitely often. An execution fragment is weakly fair with respect to some activity, α say, if it is not the case that α is always enabled beyond some point without being taken beyond this point. Weak fairness is sometimes referred to as justice.

How to express these fairness constraints? There are different ways to formulate fairness requirements. In the sequel, we adopt the action-based view and define strong fairness for (sets of) actions. (In Chapter 5, also state-based notions of fairness will be introduced and the relationship between action-based and state-based fairness is studied in detail.) Let A be a set of actions. The execution fragment ρ is said to be strongly A-fair if the actions in A are not continuously ignored under the circumstance that they can be executed infinitely often. ρ is unconditionally A-fair if some action in A is infinitely often executed in ρ. Weak fairness is defined in a similar way as strong fairness (see below).

In order to formulate these fairness notions formally, the following auxiliary notion is convenient. For state s, let $Act(s)$ denote the set of actions that are executable in state s, that is,

$$Act(s) \ = \ \{\alpha \in Act \mid \exists s' \in S.\, s \xrightarrow{\alpha} s' \ \}.$$

Definition 3.43. Unconditional, Strong, and Weak Fairness

For transition system $TS = (S, Act, \rightarrow, I, AP, L)$ without terminal states, $A \subseteq Act$, and infinite execution fragment $\rho = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ of TS:

1. ρ is *unconditionally A-fair* whenever $\overset{\infty}{\exists}\, j.\ \alpha_j \in A$.
2. ρ is *strongly A-fair* whenever

$$\left(\overset{\infty}{\exists}\, j.\ Act(s_j) \cap A \neq \varnothing \right) \implies \left(\overset{\infty}{\exists}\, j.\ \alpha_j \in A \right).$$

3. ρ is *weakly A-fair* whenever

$$\left(\overset{\infty}{\forall}\, j.\ Act(s_j) \cap A \neq \varnothing \right) \implies \left(\overset{\infty}{\exists}\, j.\ \alpha_j \in A \right).$$

■

Here, $\overset{\infty}{\exists}\, j$ stands for "there are infinitely many j" and $\overset{\infty}{\forall}\, j$ for "for nearly all j" in the sense of "for all, except for finitely many j". The variable j, of course, ranges over the natural numbers.

To check whether a run is unconditionally A-fair it suffices to consider the actions that occur along the execution, i.e., it is not necessary to check which actions in A are enabled in visited states. However, in order to decide whether a given execution is strongly or weakly A-fair, it does not suffice to only consider the actions actually occurring in the execution. Instead, also the enabled actions in all visited states need to be considered. These enabled actions are possible in the visited states, but do not necessarily have to be taken along the considered execution.

Example 3.44. A Simple Shared-Variable Concurrent Program

Consider the following two processes that run in parallel and share an integer variable x that initially has value 0:

$$\begin{array}{rcl} \textbf{proc}\ \mathsf{Inc} & = & \textbf{while}\ \langle\, x \geqslant 0\ \textbf{do}\ x := x+1 \,\rangle\ \textbf{od} \\ \textbf{proc}\ \mathsf{Reset} & = & x := -1 \end{array}$$

The pair of brackets $\langle \ldots \rangle$ embraces an atomic section, i.e., process Inc performs the check whether x is positive and the increment of x (if the guard holds) as one atomic step. Does this parallel program terminate? When no fairness constraints are imposed, it is possible that process Inc is permanently executing, i.e., process Reset never gets its turn, and the assignment $x = -1$ is not executed. In this case, termination is thus not guaranteed, and the property is refuted. If, however, we require unconditional process fairness, then every process gets its turn, and termination is guaranteed. ■

An important question now is: given a verification problem, which fairness notion to use? Unfortunately, there is no clear answer to this question. Different forms of fairness do exist—the above is just a small, though important, fragment of all possible fairness notions—and there is no single favorite notion. For verification purposes, fairness constraints are crucial, though. Recall that the purpose of fairness constraints is to rule out certain "unreasonable" computations. If the fairness constraint is too strong, relevant computations may not be considered. In case a property is satisfied (for a transition system), it might well be the case that some reasonable computation that is not considered (as it is ruled out by the fairness constraint) refutes this property. On the other hand, if the fairness constraint is too weak, we may fail to prove a certain property as some unreasonable computations (that are not ruled out) refute it.

The relationship between the different fairness notions is as follows. Each unconditionally A-fair execution fragment is strongly A-fair, and each strongly A-fair execution fragment is weakly A-fair. In general, the reverse direction does not hold. For instance, an execution fragment that solely visits states in which no A-actions are possible is strongly A-fair (as the premise of strong A-fairness does not hold), but not unconditionally A-fair. Besides, an execution fragment that only visits finitely many states in which some A-actions are enabled but never executes an A-action is weakly A-fair (as the premise of weak A-fairness does not hold), but not strongly A-fair. Summarizing, we have

$$\text{unconditional } A\text{-fairness} \implies \text{strong } A\text{-fairness} \implies \text{weak } A\text{-fairness}$$

where the reverse implication in general does not hold.

Example 3.45. Fair Execution Fragments

Consider the transition system TS_{Sem} for the semaphore-based mutual exclusion solution. We label the transitions with the actions req_i, $enter_i$ (for $i{=}1,2$), and rel in the obvious way, see Figure 3.12.

In the execution fragment

$$\langle n_1, n_2, y = 1\rangle \xrightarrow{req_1} \langle w_1, n_2, y = 1\rangle \xrightarrow{enter_1} \langle c_1, n_2, y = 0\rangle \xrightarrow{rel} \langle n_1, n_2, y = 1\rangle \xrightarrow{req_1} \ldots$$

only the first process gets its turn. This execution fragment is indicated by the dashed arrows in Figure 3.12. It is *not* unconditionally fair for the set of actions

$$A = \{\, enter_2 \,\}.$$

It is, however, strongly A-fair, since no state is visited in which the action $enter_2$ is executable, and hence the premise of strong fairness is vacuously false. In the alternative execution fragment

$$\langle n_1, n_2, y = 1\rangle \xrightarrow{req_2} \langle n_1, w_2, y = 1\rangle \xrightarrow{req_1} \langle w_1, w_2, y = 1\rangle \xrightarrow{enter_1}$$
$$\langle c_1, w_2, y = 0\rangle \xrightarrow{rel} \langle n_1, w_2, y = 1\rangle \xrightarrow{req_1} \ldots$$

the second process requests to enter its critical section but is ignored forever. This execution fragment is indicated by the dotted arrows in Figure 3.12. It is not strongly A-fair: although the action $enter_2$ is infinitely often enabled (viz. every time when visiting the state $\langle w_1, w_2, y = 1\rangle$ or $\langle n_1, w_2, y = 1\rangle$), it is never taken. It is, however, weakly A-fair, since the action $enter_2$ is not continuously enabled—it is not enabled in the state $\langle c_1, w_2, y = 0\rangle$. ■

A fairness constraint imposes a requirement on all actions in a set A. In order to enable different fairness constraints to be imposed on different, possibly nondisjoint, sets of actions, fairness *assumptions* are used. A fairness assumption for a transition system may require different notions of fairness with respect to several sets of actions.

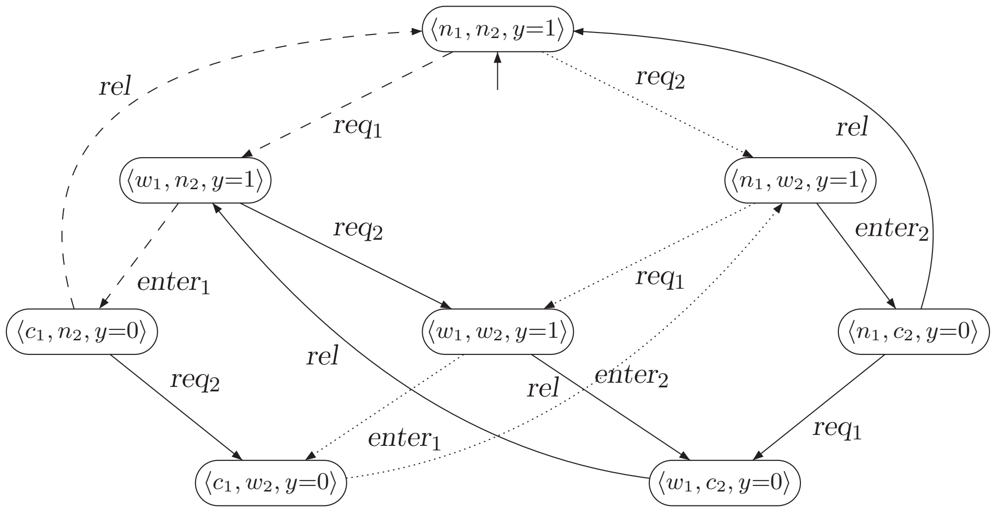

$\langle n_1, n_2, y{=}1\rangle$
$\langle w_1, n_2, y{=}1\rangle$
$\langle n_1, w_2, y{=}1\rangle$
$\langle c_1, n_2, y{=}0\rangle$
$\langle w_1, w_2, y{=}1\rangle$
$\langle n_1, c_2, y{=}0\rangle$
$\langle c_1, w_2, y{=}0\rangle$
$\langle w_1, c_2, y{=}0\rangle$
rel
req_1
req_2
rel
req_2
$enter_2$
$enter_1$
req_1
rel
req_2
rel
$enter_2$
$enter_1$
req_1

Figure 3.12: Two examples of fair execution fragments of the semaphore-based mutual exclusion algorithm.

Definition 3.46. Fairness Assumption

A *fairness assumption* for Act is a triple

$$\mathcal{F} \;=\; (\mathcal{F}_{ucond}, \mathcal{F}_{strong}, \mathcal{F}_{weak})$$

with $\mathcal{F}_{ucond}, \mathcal{F}_{strong}, \mathcal{F}_{weak} \subseteq 2^{Act}$. Execution ρ is $\mathcal{F}$-fair if

- it is unconditionally A-fair for all $A \in \mathcal{F}_{ucond}$,
- it is strongly A-fair for all $A \in \mathcal{F}_{strong}$, and
- it is weakly A-fair for all $A \in \mathcal{F}_{weak}$.

If the set $\mathcal{F}$ is clear from the context, we use the term fair instead of $\mathcal{F}$-fair. ■

Intuitively speaking, a fairness assumption is a triple of sets of (typically different) action sets, one such set of action sets is treated in a strongly fair manner, one in a weakly fair manner, and one in an unconditionally fair way. This is a rather general definition that allows imposing different fairness constraints on different sets of actions. Quite often, only a single type of fairness constraint suffices. In the sequel, we use the casual notations for these fairness assumptions. For $\mathcal{F} \subseteq 2^{Act}$, a strong fairness assumption denotes the fairness assumption $(\varnothing, \mathcal{F}, \varnothing)$. Weak, and unconditional fairness assumptions are used in a similar way.

The notion of $\mathcal{F}$-fairness as defined on execution fragments is lifted to traces and paths in the obvious way. An infinite trace σ is $\mathcal{F}$-fair if there is an $\mathcal{F}$-fair execution ρ with $\mathit{trace}(\rho) = \sigma$. $\mathcal{F}$-fair (infinite) path fragments and $\mathcal{F}$-fair paths are defined analogously.

Let $\mathit{FairPaths}_{\mathcal{F}}(s)$ denote the set of $\mathcal{F}$-paths of s (i.e., infinite $\mathcal{F}$-fair path fragments that start in state s), and $\mathit{FairPaths}_{\mathcal{F}}(\mathit{TS})$ the set of $\mathcal{F}$-fair paths that start in some initial state of TS. Let $\mathit{FairTraces}_{\mathcal{F}}(s)$ denote the set of $\mathcal{F}$-fair traces of s, and $\mathit{FairTraces}_{\mathcal{F}}(\mathit{TS})$ the set of $\mathcal{F}$-fair traces of the initial states of transition system TS:

$$\begin{aligned} \mathit{FairTraces}_{\mathcal{F}}(s) &= \mathit{trace}(\mathit{FairPaths}_{\mathcal{F}}(s)) \quad \text{and} \\ \mathit{FairTraces}_{\mathcal{F}}(\mathit{TS}) &= \bigcup_{s \in I} \mathit{FairTraces}_{\mathcal{F}}(s). \end{aligned}$$

Note that it does not make much sense to define these notions for finite traces as any finite trace is fair by default.

Example 3.47. *Mutual Exclusion Again*

Consider the following fairness requirement for two-process mutual exclusion algorithms:

"process P_i acquires access to its critical section infinitely often"

for any $i \in \{1, 2\}$. What kind of fairness assumption is appropriate to achieve this? Assume each process P_i has three states n_i (noncritical), w_i (waiting), and c_i (critical). As before, the actions req_i, enter_i, and rel are used to model the request to enter the critical section, the entering itself, and the release of the critical section. The strong-fairness assumption

$$\{\{\mathit{enter}_1, \mathit{enter}_2\}\}$$

ensures that one of the actions enter_1 or enter_2, is executed infinitely often. A behavior in which one of the processes gets access to the critical section infinitely often while the other gets access only finitely many times is strongly fair with respect to this assumption. This is, however, not intended. The strong-fairness assumption

$$\{\{\mathit{enter}_1\}, \{\mathit{enter}_2\}\}$$

indeed realizes the above requirement. This assumption should be viewed as a requirement on how to resolve the contention when both processes are awaiting to get access to the critical section. ■

Fairness assumptions can be *verifiable* properties whenever all infinite execution fragments are fair. For example, it can be verified that the transition system for Peterson's algorithm satisfies the strong-fairness assumption

$$\mathcal{F}_{strong} \; = \; \{ \{ \, enter_1 \, \}, \{ \, enter_2 \, \} \}.$$

But in many cases it is necessary to *assume* the validity of the fairness conditions to verify liveness properties.

A transition system TS satisfies the LT property P under fairness assumption $\mathcal{F}$ if all $\mathcal{F}$-fair paths fulfill the property P. However, no requirements whatsoever are imposed on the unfair paths. This is formalized as follows.

Definition 3.48. Fair Satisfaction Relation for LT Properties

Let P be an LT property over AP and $\mathcal{F}$ a fairness assumption over Act. Transition system $TS = (S, Act, \rightarrow, I, AP, L)$ *fairly satisfies* P, notation $TS \models_{\mathcal{F}} P$, if and only if $FairTraces_{\mathcal{F}}(TS) \; \subseteq \; P$. ■

For a transition system that satisfies the fairness assumption $\mathcal{F}$ (i.e., *all* paths are $\mathcal{F}$-fair), the satisfaction relation $\models$ without fairness assumptions (see Definition 3.11, page 100) corresponds with the fair satisfaction relation $\models_{\mathcal{F}}$. In this case, the fairness assumption does not rule out any trace. However, in case a transition system has traces that are not $\mathcal{F}$-fair, then in general we are confronted with a situation

$$TS \; \models_{\mathcal{F}} \; P \quad \text{whereas} \quad TS \; \not\models \; P.$$

By restricting the validity of a property to the set of fair paths, the verification can be restricted to "realistic" executions.

Before turning to some examples, a few words on the relationship between unconditional, strong, and weak fairness are (again) in order. As indicated before, we have that the set of unconditional A-fair executions is a subset of all strong A-fair executions. In a similar way, the latter set of executions is a subset of all weak A-fair executions. Stated differently, unconditional fairness rules out more behaviors than strong fairness, and strong excludes more behaviors than weak fairness. For $\mathcal{F} = \{ A_1, \ldots, A_k \}$, let fairness assumption $\mathcal{F}_{ucond} = (\mathcal{F}, \varnothing, \varnothing)$, $\mathcal{F}_{strong} = (\varnothing, \mathcal{F}, \varnothing)$, and $\mathcal{F}_{weak} = (\varnothing, \varnothing, \mathcal{F})$. Then for any transition system TS and LT property P it follows that:

$$TS \models_{\mathcal{F}_{weak}} P \; \Rightarrow \; TS \models_{\mathcal{F}_{strong}} P \; \Rightarrow \; TS \models_{\mathcal{F}_{ucond}} P.$$

Example 3.49. Independent Traffic Lights

Consider again the independent traffic lights. Let action *switch2green* denote the switching to green. Similarly *switch2red* denotes the switching to red. The fairness assumption

$$\mathcal{F} \;=\; \{\,\{\,switch2green_1, switch2red_1\,\}, \{\,switch2green_2, switch2red_2\,\}\,\}$$

expresses that both traffic lights infinitely often switch color. In this case, it is irrelevant whether strong, weak, or unconditional fairness is required.

Note that in this example $\mathcal{F}$ is *not* a verifiable system property (as it is not guaranteed to hold), but a natural property which is satisfied for a practical implementation of the system (with two independent processors). Obviously,

$$TrLight_1 \;|||\; TrLight_2 \;\models_{\mathcal{F}}\; \text{“each traffic light is green infinitely often”}$$

while the corresponding proposition for the nonfair relation $\models$ is refuted. ■

Example 3.50. Fairness for Mutual Exclusion Algorithms

Consider again the semaphore-based mutual exclusion algorithm, and assume the fairness assumption $\mathcal{F}$ consists of

$$\mathcal{F}_{weak} \;=\; \{\{\,req_1\,\}, \{\,req_2\,\}\} \quad \text{and} \quad \mathcal{F}_{strong} \;=\; \{\{\,enter_1\,\}, \{\,enter_2\,\}\}$$

and $\mathcal{F}_{ucond} = \varnothing$. The strong fairness constraint requires each process to enter its critical section infinitely often when it infinitely often gets the opportunity to do so. This does not forbid a process to never leave its noncritical section. To avoid this unrealistic scenario, the weak fairness constraint requires that any process infinitely often requests to enter the critical section. In order to do so, each process has to leave the noncritical section infinitely often. It follows that $TS_{Sem} \;\models_{\mathcal{F}}\; P$ where P stands for the property “every process enters its critical section infinitely often”.

Weak fairness is sufficient for request actions, as such actions are not critical: if req_i is executable in (global) state s, then it is executable in all direct successor states of s that are reached by an action that differs from req_i.

Peterson’s algorithm satisfies the strong fairness property

“Every process that requests access to the critical section
will eventually be able to do so”.

We can, however, not ensure that a process will ever leave its noncritical section and request the critical section. That is, the property P is refuted. This can be “repaired” by imposing the weak fairness constraint $\mathcal{F}_{weak} \;=\; \{\,\{\,req_1\,\}, \{\,req_2\,\}\,\}$. We now have $TS_{Pet} \models_{\mathcal{F}_{weak}} P$. ■

3.5.2 Fairness Strategies

The examples in the previous section indicate that fairness assumptions may be necessary to verify liveness properties of transition system TS. In order to rule out the "unrealistic" computations, fairness assumptions are imposed on the traces of TS, and it is checked whether $TS \models_{\mathcal{F}} P$ as opposed to checking $TS \models P$ (without fairness). Which fairness assumptions are appropriate to check P? Many model-checking tools provide the possibility to work with built-in fairness assumptions. Roughly speaking, the intention is to rule out executions that cannot occur in a realistic implementation. But what does that exactly mean? In order to give some insight into this, we consider several fairness assumptions for synchronizing concurrent systems. The aim is to establish a fair communication mechanism between the various processes involved. A rule of thumb is: Strong fairness is needed to obtain an adequate resolution of contentions (between processes), while weak fairness suffices for sets of actions that represent the concurrent execution of independent actions (i.e., interleaving).

For modeling *asynchronous* concurrency by means of transition systems, the following rule of thumb can be adopted:

$$\boxed{\text{concurrency} \quad = \quad \text{interleaving (i.e., nondeterminism)} \; + \; \text{fairness}}$$

Example 3.51. Fair Concurrency with Synchronization

Consider the concurrent transition system:

$$TS = TS_1 \parallel TS_2 \parallel \ldots \parallel TS_n \quad ,$$

where $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i)$, for $1 \leqslant i \leqslant n$, is a transition system without terminal states. Recall that each pair of processes TS_i and TS_j (for $i \neq j$) has to synchronize on their common sets of actions, i.e., $Syn_{i,j} = Act_i \cap Act_j$. It is assumed that $Syn_{i,j} \cap Act_k = \varnothing$ for any $k \neq i, j$. For simplicity, it is assumed that TS has no terminal states. (In case there are terminal states, each finite execution is considered to be fair.)

We consider several fairness assumptions on the transition system TS. First, consider the strong fairness assumption

$$\{Act_1, Act_2, \ldots, Act_n\}$$

which ensures that each transition system TS_i executes an action infinitely often, provided the composite system TS is infinitely often in a (global) state with a transition being executable in which TS_i participates. This fairness assumption, however, cannot ensure that a communication will ever occur—it is possible for each TS_i to only execute local actions ad infinitum.

In order to force a synchronization to take place every now and then, the strong fairness assumption

$$\{\,\{\,\alpha\,\} \mid \alpha \in \mathit{Syn}_{i,j}, 0 < i < j \leqslant n\,\} \tag{3.1}$$

could be imposed. It forces every synchronization action to happen infinitely often. Alternatively, a somewhat weaker fairness assumption can be imposed by requiring every pair of processes to synchronize—regardless of the synchronization action—infinitely often. The corresponding strong fairness assumption is

$$\{\,\mathit{Syn}_{i,j} \mid 0 < i < j \leqslant n\,\}. \tag{3.2}$$

Whereas (3.2) allows processes to always synchronize on the same action, (3.1) does not permit this. The strong fairness assumption:

$$\{\,\bigcup_{0<i<j\leqslant n} \mathit{Syn}_{i,j}\,\}$$

goes even one step further as it only requires a synchronization to take place infinitely often, regardless of the process involved. This fairness assumption does not rule out executions in which always the same synchronization takes place or in which always the same pair of processes synchronizes.

Note that all fairness assumptions in this example so far are strong. This requires that infinitely often a synchronization is enabled. As the constituting transition systems TS_i may execute internal actions, synchronizations are not continuously enabled, and hence weak fairness is in general inappropriate.

If the internal actions should be fairly considered, too, then we may use, e.g., the strong fairness assumption

$$\{\,\mathit{Act}_1 \setminus \mathit{Syn}_1, \ldots, \mathit{Act}_n \setminus \mathit{Syn}_n\,\} \;\cup\; \{\,\{\,\alpha\,\} \mid \alpha \in \mathit{Syn}\,\},$$

where $\mathit{Syn}_i \;=\; \bigcup_{j \neq i} \mathit{Syn}_{i,j}$ denotes the set of all synchronization actions of TS_i and $\mathit{Syn} \;=\; \bigcup_i \mathit{Syn}_i$.

Under the assumption that in every (local) state either only internal actions or only synchronization actions are executable, it suffices to impose the *weak* fairness constraint

$$\{\,\mathit{Act}_1 \setminus \mathit{Syn}_1, \ldots, \mathit{Act}_n \setminus \mathit{Syn}_n\,\}.$$

Weak fairness is appropriate for the internal actions $\alpha \in \mathit{Act}_i \setminus \mathit{Syn}_i$, as the ability to perform an internal action is preserved until it will be executed. ■

As an example of another form of fairness we consider the following sequential hardware circuit.

Example 3.52. *Circuit Fairness*

For sequential circuits we have modeled the environmental behavior, which provides the input bits, by means of nondeterminism. It may be necessary to impose fairness assumptions on the environment in order to be able to verify liveness properties, such as "the values 0 and 1 are output infinitely often". Let us illustrate this by means of a concrete example. Consider a sequential circuit with input variable x, output variable y, and register r. Let the transition function and the output function be defined as

$$\lambda_y = \delta_r = x \leftrightarrow \neg r.$$

That is, the circuit inverts the register and output evaluation if and only if the input bit is set. If x=0, then the register evaluation remains unchanged. The value of the register is output. Suppose all transitions leading to a state with a register evaluation of the form $[r = 1, \ldots]$ are labeled with the action *set*. Imposing the unconditional fairness assumption $\{\{\, set \,\}\}$ ensures that the values 0 and 1 are output infinitely often. ■

3.5.3 Fairness and Safety

While fairness assumptions may be necessary to verify liveness properties, they are irrelevant for verifying safety properties, provided that they can always be ensured by means of an appropriate scheduling strategy. Such fairness assumptions are called *realizable* fairness assumptions. A fairness assumption cannot be realized in a transition system whenever there exists a reachable state from where *no* fair path begins. In this case, it is impossible to design a scheduler that resolves the nondeterminism such that only fair paths remain.

Example 3.53. *A Nonrealizable Fairness Assumption*

Consider the transition system depicted in Figure 3.13, and suppose the unconditional fairness assumption $\{\,\{\,\alpha\,\}\,\}$ is imposed. As the α-transition can only be taken once, it is evident that the transition system can never guarantee this form of fairness. As there is a reachable state from which no unconditional fair path exists, this fairness assumption is nonrealizable. ■

Definition 3.54. **Realizable Fairness Assumption**

Let *TS* be a transition system with the set of actions *Act* and $\mathcal{F}$ a fairness assumption for *Act*. $\mathcal{F}$ is called *realizable* for *TS* if for every reachable state s: $FairPaths_{\mathcal{F}}(s) \neq \varnothing$. ■

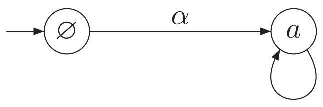


Figure 3.13: Unconditional fairness.

Stated in words, a fairness assumption is realizable in a transition system *TS* whenever in any reachable state at least one fair execution is possible. This entails that every initial finite execution fragment of *TS* can be completed to a fair execution. Note that there is no requirement on the unreachable states.

The following theorem shows the irrelevance of realizable fairness assumptions for the verification of safety properties. The *suffix property* of fairness assumptions is essential for its proof. This means the following. If

$$\rho \;=\; s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \;\ldots$$

is an (infinite) execution fragment, then ρ is fair if and only if every suffix

$$s_j \xrightarrow{\alpha_{j+1}} s_{j+1} \xrightarrow{\alpha_{j+2}} s_{j+2} \xrightarrow{\alpha_{j+3}} \;\ldots$$

of ρ is fair too. Conversely, every fair execution fragment ρ (as above) starting in state s_0 can be preceded by an arbitrary finite execution fragment

$$s'_0 \xrightarrow{\beta_1} s'_1 \xrightarrow{\beta_2} \;\ldots \xrightarrow{\beta_n} s'_n = s_0$$

ending in s_0. Proceeding in s_0 by execution ρ yields the fair execution fragment:

$$\underbrace{s'_0 \xrightarrow{\beta_1} s'_1 \xrightarrow{\beta_2} \;\ldots \xrightarrow{\beta_n} s'_n}_{\text{arbitrary starting fragment}} = \underbrace{s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \;\ldots}_{\text{fair continuation}}$$

Theorem 3.55. Realizable Fairness is Irrelevant for Safety Properties

Let TS be a transition system with set of propositions AP, $\mathcal{F}$ a realizable fairness assumption for TS, and P_{safe} a safety property over AP. Then:

$$TS \models P_{safe} \quad \textit{if and only if} \quad TS \models_{\mathcal{F}} P_{safe}.$$

Proof: "$\Rightarrow$": Assume $TS \models P_{safe}$. Then, by definition of $\models$ and the fact that the fair traces of *TS* are a subset of the traces of *TS*, we have

$$\mathit{FairTraces}_{\mathcal{F}}(TS) \;\subseteq\; \mathit{Traces}(TS) \;\subseteq\; P_{safe}.$$

Thus, by definition of $\models_{\mathcal{F}}$ it follows that $TS \models_{\mathcal{F}} P_{safe}$.

"$\Leftarrow$": Assume $TS \models_{\mathcal{F}} P_{safe}$. It is to be shown $TS \models P_{safe}$, i.e., $Traces(TS) \subseteq P_{safe}$. This is done by contraposition. Let $\sigma \in Traces(TS)$ and assume $\sigma \notin P_{safe}$. As $\sigma \notin P_{safe}$, there is a bad prefix of σ, $\widehat{\sigma}$ say, for P_{safe}. Hence, the set of properties that has $\widehat{\sigma}$ as a prefix, i.e.,

$$P \;=\; \left\{\sigma' \in \left(2^{AP}\right)^{\omega} \mid \widehat{\sigma} \in pref(\sigma')\right\},$$

satisfies $P \cap P_{safe} = \varnothing$. Further, let $\widehat{\pi} = s_0\, s_1 \ldots s_n$ be a finite path fragment of TS with

$$trace(\widehat{\pi}) \;=\; \widehat{\sigma}.$$

Since $\mathcal{F}$ is a realizable fairness assumption for TS and $s_n \in Reach(TS)$, there is an $\mathcal{F}$-fair path starting in s_n. Let

$$s_n\, s_{n+1}\, s_{n+2} \ldots \in FairPaths_{\mathcal{F}}(s_n).$$

The path $\pi = s_0 \ldots s_n\, s_{n+1}\, s_{n+2} \ldots$ is in $FairPaths_{\mathcal{F}}(TS)$ and thus,

$$trace(\pi) \;=\; L(s_0) \ldots L(s_n)\, L(s_{n+1})\, L(s_{n+2}) \ldots \;\in\; FairTraces_{\mathcal{F}}(TS) \;\subseteq\; P_{safe}.$$

On the other hand, $\widehat{\sigma} = L(s_0) \ldots L(s_n)$ is a prefix of $trace(\pi)$. Thus, $trace(\pi) \in P$. This contradicts $P \cap P_{safe} = \varnothing$. ■

Theorem 3.55 does not hold if arbitrary (i.e., possibly nonrealizable) fairness assumptions are permitted. This is illustrated by the following example.

Example 3.56. Nonrealizable Fairness may harm Safety Properties

Consider the transition system TS in Figure 3.14 and suppose the unconditional fairness assumption $\mathcal{F} = \{\{\alpha\}\}$ is imposed. $\mathcal{F}$ is not realizable for TS, as the noninitial state (referred to as state s_1), is reachable, but has no $\mathcal{F}$-fair execution. Obviously, TS has only one fair path (namely the path that never leaves the initial state s_0). In contrast, paths of the form $s_0 \ldots s_0\, s_1\, s_1\, s_1 \ldots$ are not fair, since α is only executed finitely often. Accordingly, we have that

$$TS \models_{\mathcal{F}} \text{"never } a\text{"} \quad \text{but} \quad TS \not\models \text{"never } a\text{"}.$$

■

3.6 Summary

- The set of reachable states of a transition system TS can be determined by a search algorithm on the state graph of TS.

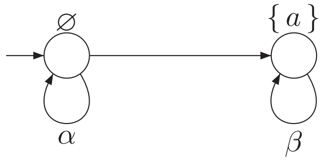


Figure 3.14: Unconditional fairness may be relevant for safety properties.

- A trace is a sequence of sets (!) of atomic propositions. The traces of a transition system *TS* are obtained from projecting the paths to the sequence of state labels.
- A linear-time (LT, for short) property is a set of infinite words over the alphabet 2^{AP}.
- Two transition systems are trace-equivalent (i.e., they exhibit the same traces) if and only if they satisfy the same LT properties.
- An invariant is an LT property that is purely state-based and requires a propositional logic formula Φ to hold for all reachable states. Invariants can be checked using a depth-first search where the depth-first search stack can be used to provide a counterexample in case an invariant is refuted.
- Safety properties are generalizations of invariants. They constrain the finite behaviors. The formal definition of safety properties can be provided by means of their bad prefixes in the sense that each trace that refutes a safety property has a finite prefix, the bad prefix, that causes this.
- Two transition systems exhibit the same finite traces if and only if they satisfy the same safety properties.
- A liveness property is an LT property if it does not rule out any finite behavior. It constrains infinite behavior.
- Any LT property is equivalent to an LT property that is a conjunction of a safety and a liveness property.
- Fairness assumptions serve to rule out traces that are considered to be unrealistic. They consist of unconditional, strong, and weak fairness constraints, i.e., constraints on the actions that occur along infinite executions.
- Fairness assumptions are often necessary to establish liveness properties, but they are—provided they are realizable—irrelevant for safety properties.

3.7 Bibliographic Notes

The dining philosophers example discussed in Example 3.2 has been developed by Dijkstra [128] in the early seventies to illustrate the intricacies of concurrency. Since then it has become one of the standard examples for reasoning about parallel systems.

The depth-first search algorithm that we used as a basis for the invariance checking algorithm goes back to Tarjan [387]. Further details about graph traversal algorithms can be found in any textbook on algorithms and data structures, e.g. [100], or on graph algorithms [188].

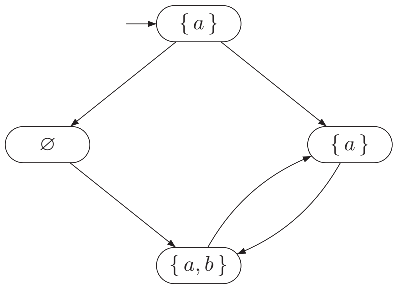
Traces. Traces have been introduced by Hoare [202] to describe the linear-time behavior of transition systems and have been used as the initial semantical model for the process algebra CSP. Trace theory has further been developed by, among others, van de Snepscheut [403] and Rem [354] and has successfully been used to design and analyze fine-grained parallel programs that occur in, e.g., asynchronous hardware circuits. Several extensions to traces and their induced equivalences have been proposed, such as failures [65] where a trace is equipped with information about which actions are rejected after execution of such trace. The FDR model checker [356] supports the automated checking of failure-divergence refinement and the checking of safety properties. A comprehensive survey of these refined notions of trace equivalence and trace inclusion has recently been given by Bruda [68].

Safety and liveness. The specification of linear-time properties using sets of infinite sequences of states (and their topological characterizations) goes back to Alpern and Schneider [5, 6, 7]. An earlier approach by Gerth [164] was based on finite sequences. Lamport [257] categorized properties as either safety, liveness, or properties that are neither. Alternative characterizations have been provided by Rem [355] and Gumm [178]. Subclasses of liveness and safety properties in the linear-time framework have been identified by Sistla [371], and Chang, Manna, and Pnueli [80]. Other definitions of liveness properties have been provided by Dederichs and Weber [119] and Naumovich and Clarke [312] (for linear-time properties), and Manolios and Trefler [285, 286] (for branching-time properties). A survey of safety and liveness has been given by Kindler [239].

Fairness. Fairness has implicitly been introduced by Dijkstra [126, 127] by assuming that one should abstract from the speed of processors and that each process gets its turn once it is initiated. Park [321] studied the notion of fairness in providing a semantics to data-flow languages. Weak and strong fairness have been introduced by Lehmann, Pnueli, and Stavi [267] in the context of shared variable concurrent programs. Queille and Sifakis [348] consider fairness for transition systems. An overview of the fairness notions has been provided by Kwiatkowska [252]. An extensive treatment of fairness can be found in the monograph by Francez [155]. A recent characterization of fairness in terms of topology, language theory, and game theory has been provided by Völzer, Varacca, and Kindler [415].

3.8 Exercises

Exercise 3.1. Give the traces on the set of atomic propositions $\{ a, b \}$ of the following transition system:



Exercise 3.2. On page 97, a transformation is described of a transition system TS with possible terminal states into an "equivalent" transition system TS^* without terminal states. Questions:

(a) Give a formal definition of this transformation $TS \mapsto TS^*$

(b) Prove that the transformation preserves trace-equivalence, i.e., show that if TS_1, TS_2 are transition systems (possibly with terminal states) such that $Traces(TS_1) = Traces(TS_2)$, then $Traces(TS_1^*) = Traces(TS_2^*)$.[8]

Exercise 3.3. Give an algorithm (in pseudocode) for invariant checking such that in case the invariant is refuted, a *minimal* counterexample, i.e., a counterexample of minimal length, is provided as an error indication.

Exercise 3.4. Recall the definition of AP-deterministic transition systems (Definition 2.5 on page 24). Let TS and TS' be transition systems with the same set of atomic propositions AP. Prove the following relationship between trace inclusion and finite trace inclusion:

(a) For AP-deterministic TS and TS':

$$Traces(TS) = Traces(TS') \text{ if and only if } Traces_{fin}(TS) = Traces_{fin}(TS').$$

[8]If TS is a transition system with terminal states, then $Traces(TS)$ is defined as the set of all words $trace(\pi)$ where π is an initial, maximal path fragment in TS.

(b) Give concrete examples of TS and TS' where at least one of the transition systems is not AP-deterministic, but

$$\mathit{Traces}(TS) \not\subseteq \mathit{Traces}(TS') \ \text{ and } \ \mathit{Traces}_{\mathit{fin}}(TS) = \mathit{Traces}_{\mathit{fin}}(TS').$$

Exercise 3.5. Consider the set AP of atomic propositions defined by $AP = \{ x = 0, x > 1 \}$ and consider a nonterminating sequential computer program P that manipulates the variable x. Formulate the following informally stated properties as LT properties:

(a) false

(b) initially x is equal to zero

(c) initially x differs from zero

(d) initially x is equal to zero, but at some point x exceeds one

(e) x exceeds one only finitely many times

(f) x exceeds one infinitely often

(g) the value of x alternates between zero and two

(h) true

(This exercise has been adopted from [355].) Determine which of the provided LT properties are safety properties. Justify your answers.

Exercise 3.6. Consider the set $AP = \{ A, B \}$ of atomic propositions. Formulate the following properties as LT properties and characterize each of them as being either an invariance, safety property, or liveness property, or none of these.

(a) A should never occur,

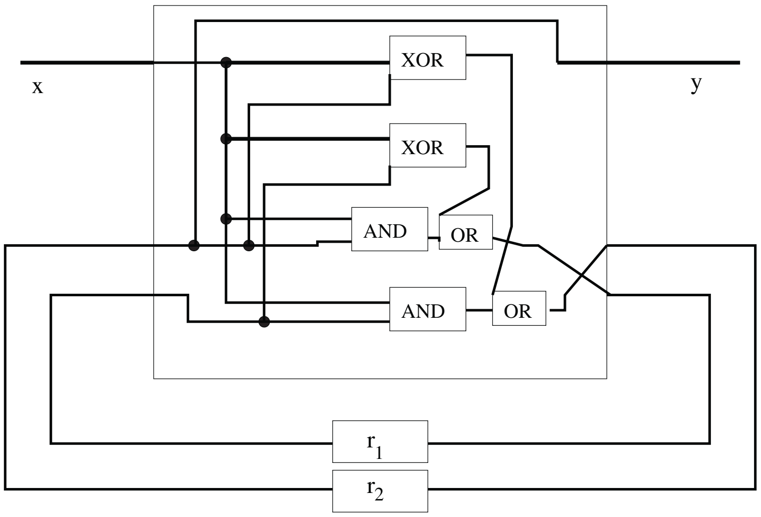
(b) A should occur exactly once,

(c) A and B alternate infinitely often,

(d) A should eventually be followed by B.

(This exercise has been inspired by [312].)

Exercise 3.7. Consider the following sequential hardware circuit:



The circuit has input variable x, output variable y, and registers r_1 and r_2 with initial values $r_1 = 0$ and $r_2 = 1$. The set AP of atomic propositions equals $\{ x, r_1, r_2, y \}$. Besides, consider the following informally formulated LT properties over AP:

P_1 : Whenever the input x is continuously high (i.e., x=1), then the output y is infinitely often high.

P_2 : Whenever currently r_2=0, then it will never be the case that after the next input, r_1=1.

P_3 : It is never the case that two successive outputs are high.

P_4 : The configuration with x=1 and r_1=0 never occurs.

Questions:

(a) Give for each of these properties an example of an infinite word that belongs to P_i. Do the same for the property $\left(2^{AP}\right)^{\omega} \setminus P_i$, i.e., the complement of P_i.

(b) Determine which properties are satisfied by the hardware circuit that is given above.

(c) Determine which of the properties are safety properties. Indicate which properties are invariants.

   (i) For each safety property P_i, determine the (regular) language of bad prefixes.

   (ii) For each invariant, provide the propositional logic formula that specifies the property that should be fulfilled by each state.

Exercise 3.8. Let LT properties P and P' be equivalent, notation $P \cong P'$, if and only if *pref*(P) = *pref*(P'). Prove or disprove: $P \cong P'$ if and only if *closure*(P) = *closure*(P').

EXERCISE 3.9. Show that for any transition system TS, the set $closure(Traces(TS))$ is a safety property such that $TS \models closure(Traces(TS))$.

EXERCISE 3.10. Let P be an LT property. Prove: $pref(closure(P)) = pref(P)$.

EXERCISE 3.11. Let P and P' be liveness properties over AP. Prove or disprove the following claims:

(a) $P \cup P'$ is a liveness property,

(b) $P \cap P'$ is a liveness property.

Answer the same question for P and P' being safety properties.

EXERCISE 3.12. Prove Lemma 3.38 on page 125.

EXERCISE 3.13. Let $AP = \{ a, b \}$ and let P be the LT property of all infinite words $\sigma = A_0 A_1 A_2 \ldots \in \left(2^{AP}\right)^{\omega}$ such that there exists $n \geqslant 0$ with $a \in A_i$ for $0 \leqslant i < n$, $\{ a, b \} = A_n$ and $b \in A_j$ for infinitely many $j \geqslant 0$. Provide a decomposition $P = P_{safe} \cap P_{live}$ into a safety and a liveness property.

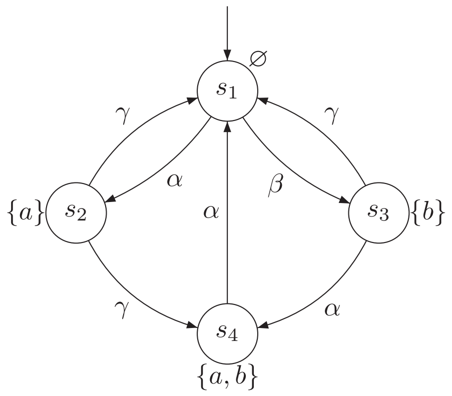
EXERCISE 3.14. Let TS_{Sem} and TS_{Pet} be the transition systems for the semaphore-based mutual exclusion algorithm (Example 2.24 on page 43) and Peterson's algorithm (Example 2.25 on page 45), respectively. Let $AP = \{ wait_i, crit_i \mid i = 1, 2 \}$. Prove or disprove:

$$Traces(TS_{Sem}) = Traces(TS_{Pet}).$$

If the property does not hold, provide an example trace of one transition system that is not a trace of the other one.

EXERCISE 3.15. Consider the transition system TS outlined on the right and the sets of actions $B_1 = \{ \alpha \}$, $B_2 = \{ \alpha, \beta \}$, and $B_3 = \{ \beta \}$. Further, let E_b, E_a and E' be the following LT properties:

- E_b = the set of all words $A_0A_1 \cdots \in \left(2^{\{a,b\}}\right)^\omega$ with $A_i \in \{\{a,b\},\{b\}\}$ for infinitely many i (i.e., infinitely often b).
- E_a = the set of all words $A_0A_1 \cdots \in \left(2^{\{a,b\}}\right)^\omega$ with $A_i \in \{\{a,b\},\{a\}\}$ for infinitely many i (i.e., infinitely often a).
- E' = set of all words $A_0A_1 \cdots \in \left(2^{\{a,b\}}\right)^\omega$ for which there does not exist an $i \in \mathbb{N}$ s.t. $A_i = \{a\}$, $A_{i+1} = \{a,b\}$ and $A_{i+2} = \varnothing$.



Questions:

(a) For which sets of actions B_i ($i \in \{1,2,3\}$) and LT properties $E \in \{E_a, E_b, E'\}$ it holds that $TS \models_{\mathcal{F}_i} E$? Here, $\mathcal{F}_i$ is a strong fairness condition with respect to B_i that does not impose any unconditional or weak fairness conditions (i.e., $\mathcal{F}_i = (\varnothing, \{B_i\}, \varnothing)$).

(b) Answer the same question in the case of weak fairness (instead of strong fairness, i.e., $\mathcal{F}_i = (\varnothing, \varnothing, \{B_i\})$).

EXERCISE 3.16. Let TS_i (for i=1, 2) be the transition system $(S_i, Act, \rightarrow_i, I_i, AP_i, L_i)$ and $\mathcal{F} = (\mathcal{F}_{ucond}, \mathcal{F}_{strong}, \mathcal{F}_{weak})$ be a fairness assumption with $\mathcal{F}_{ucond} = \varnothing$. Prove or disprove (i.e., give a counterexample for) the following claims:

(a) $Traces(TS_1) \subseteq Traces(TS_1 \parallel TS_2)$ where $Syn \subseteq Act$

(b) $Traces(TS_1) \subseteq Traces(TS_1 \,|||\, TS_2)$

(c) $Traces(TS_1 \parallel TS_2) \subseteq Traces(TS_1)$ where $Syn \subseteq Act$

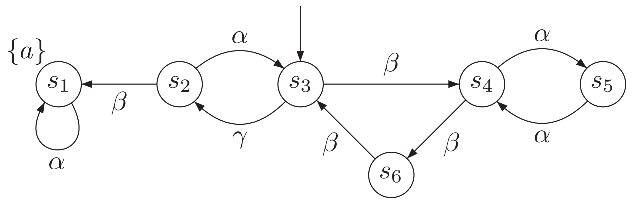
(d) $Traces(TS_1) \subseteq Traces(TS_2) \;\Rightarrow\; FairTraces_{\mathcal{F}}(TS_1) \subseteq FairTraces_{\mathcal{F}}(TS_2)$

(e) For liveness property P with $TS_2 \models_{\mathcal{F}} P$ we have

$$Traces(TS_1) \subseteq Traces(TS_2) \quad \Rightarrow \quad TS_1 \models_{\mathcal{F}} P.$$

Assume that in items (a) through (c), we have $AP_2 = \varnothing$ and that $TS_1 \parallel TS_2$ and $TS_1 \,|||\, TS_2$, respectively, have $AP = AP_1$ as atomic propositions and $L(\langle s, s' \rangle) = L_1(s)$ as labeling function. In items (d) and (e) you may assume that $AP_1 = AP_2$.

EXERCISE 3.17. Consider the following transition system TS with the set of atomic propositions $\{a\}$:

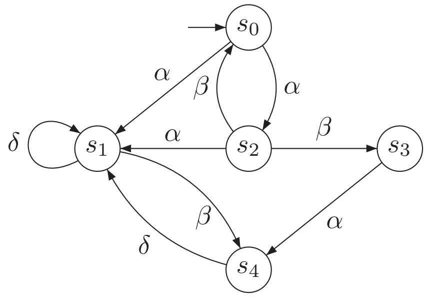


Let the fairness assumption

$$\mathcal{F} = (\varnothing, \{\{\alpha\},\{\beta\}\}, \{\{\beta\}\}).$$

Determine whether $TS \models_{\mathcal{F}}$ "eventually a". Justify your answer!

EXERCISE 3.18. Consider the following transition system TS (without atomic propositions):



Decide which of the following fairness assumptions $\mathcal{F}_i$ are realizable for TS. Justify your answers!

(a) $\mathcal{F}_1 = (\{\{\alpha\}\}, \{\{\delta\}\}, \{\{\alpha,\beta\}\})$

(b) $\mathcal{F}_2 = (\{\{\delta,\alpha\}\}, \{\{\alpha,\beta\}\}, \{\{\delta\}\})$

(c) $\mathcal{F}_3 = (\{\{\alpha,\delta\},\{\beta\}\}, \{\{\alpha,\beta\}\}, \{\{\delta\}\})$

EXERCISE 3.19. Let $AP = \{a, b\}$.

(a) P_1 denotes the LT property that consists of all infinite words $\sigma = A_0A_1A_2\ldots \in \left(2^{AP}\right)^\omega$ such that there exists $n \geqslant 0$ with

$$\forall j < n.\ A_j = \varnothing \quad \wedge \quad A_n = \{a\} \quad \wedge \quad \forall k > n.\ (A_k = \{a\} \Rightarrow A_{k+1} = \{b\}).$$

(i) Give an ω–regular expression for P_1.

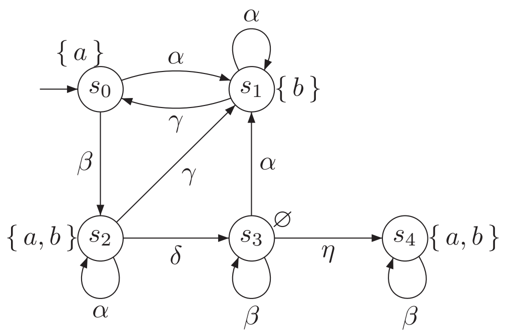
(ii) Apply the decomposition theorem and give expressions for P_{safe} and P_{live}.

(iii) Justify that P_{live} is a liveness and that P_{safe} is a safety property.

(b) Let P_2 denote the set of traces of the form $\sigma = A_0 A_1 A_2 \ldots \in \left(2^{AP}\right)^\omega$ such that

$$\overset{\infty}{\exists}\, k.\ A_k = \{\, a, b \,\} \quad \wedge \quad \exists n \geqslant 0.\ \forall k > n.\ \big(a \in A_k \Rightarrow b \in A_{k+1}\big).$$

Consider the following transition system *TS*:



Consider the following fairness assumptions:

(a) $\mathcal{F}_1 = \Big(\big\{\{\alpha\}\big\}, \big\{\{\beta\}, \{\delta, \gamma\}, \{\eta\}\big\}, \varnothing\Big)$. Decide whether $TS \models_{\mathcal{F}_1} P_2$.

(b) $\mathcal{F}_2 = \Big(\big\{\{\alpha\}\big\}, \big\{\{\beta\}, \{\gamma\}\big\}, \big\{\{\eta\}\big\}\Big)$. Decide whether $TS \models_{\mathcal{F}_2} P_2$.

Justify your answers.

Exercise 3.20. Let $TS = (S, Act, \to, I, AP, L)$ be a transition system without terminal states and let $A_1, \ldots, A_k, A'_1, \ldots, A'_l \subseteq Act$.

(a) Let $\mathcal{F}$ be the fairness assumption $\mathcal{F} = (\varnothing, \mathcal{F}_{strong}, \mathcal{F}_{weak})$ where

$$\mathcal{F}_{strong} = \{A_1, \ldots, A_k\} \text{ and } \mathcal{F}_{weak} = \{A'_1, \ldots, A'_l\}.$$

Provide a sketch of a scheduling algorithm that resolves the nondeterminism in *TS* in an $\mathcal{F}$-fair way.

(b) Let $\mathcal{F}_{ucond} = \{A_1, \ldots, A_k\}$, viewed as an unconditional fairness assumption for *TS*. Design a (scheduling) algorithm that checks whether $\mathcal{F}_{ucond}$ for *TS* is realizable, and if so, generates an $\mathcal{F}_{ucond}$-fair execution for *TS*.