Contents

Chapter 8

Fair Modules

8.1 Safety versus Liveness

So far, we have considered safety requirements of reactive modules. Intuitively, a *safety requirement* is a requirement that can be violated by a finite trace. For example, the mutual-exclusion and equal-opportunity requirements can be violated by finite traces. More generally, all SAL requirements can be violated by finite traces. Why would we care about requirements that cannot be violated by finite traces? Such requirements would not be violated within the next year, nor within our lifetime, nor within the lifetime of the universe. The answer is convenience in system and requirement description.

Let us assume, for the sake of argument, that there is no truly nondeterministic physical process. Even with this assumption, the nondeterministic update command

$$\begin{array}{l} \textbf{update } x \\ \quad [\!] \ \mathit{true} \rightarrow x' := 0 \\ \quad [\!] \ \mathit{true} \rightarrow x' := 1 \end{array}$$

is useful for describing systems that assign 0s and 1s to x, because the nondeterminism frees us from the responsibility of being specific when x is 0, and when x is 1. The actual law that determines the value of x in each round may be arbitrarily complex, and yet irrelevant for our purposes of proving certain system requirements. Similarly, it is often convenient to assert that an event will happen, without giving detailed information on when it will happen. For example, we may want to assert that x never stays 0, and it never stays 1, without being specific on how many rounds can expire between consecutive changes in the value of x. No finite trace can violate this assertion, yet an infinite trace can violate it by having one the value of x remain unchanged after some round. We use the notion of *fair update choices* for enforcing the eventual execution of particular guarded assignments of a nondeterministic update command:

$$\begin{array}{l} \textbf{update } x \textbf{ weaklyfair } a, b \\ \quad [\!]\ \mathit{true} \xrightarrow{a} x' := 0 \\ \quad [\!]\ \mathit{true} \xrightarrow{b} x' := 1 \end{array}$$

The declaration **weakly fair** a ensures that the update choice a, which sets x to 0, cannot be neglected forever, and the declaration **weakly fair** b ensures the same for the update choice b, which sets x to 1.

Requirements that can be violated by infinite traces only are called *liveness requirements.* Trajectories and traces of reactive modules are finite sequences of states and observations, respectively. In order to specify whether a module satisfies or violates a liveness requirement, we need to define infinite trajectories and infinite traces, called ω*-trajectories* and ω*-traces.*

8.1.1 ω-Words and ω-Languages

Let A be a set of symbols. An ω*-word* $\underline{a} = a_0 a_1 a_2 \cdots$ *over the alphabet* A is an infinite sequence of symbols a_i from A. We write A^ω for the set of ω-words over A. An ω*-language* $\mathcal{L}$ *over the alphabet* A is a set of ω-words over A; that is, $\mathcal{L} \subseteq A^\omega$.

For a word $\overline{b} \in A^*$ and an ω-word $\underline{a}$, by $\overline{b} \cdot \underline{a}$ we denote the ω-word that results from *concatenating* the two. The word $\overline{a}$ is a *prefix* of the ω-word $\underline{b}$ if there exists an ω-word $\underline{c}$ such that $\underline{b} = \overline{a} \cdot \underline{c}$, and $\underline{a}$ is a *suffix* of $\underline{b}$ if there exists a word $\overline{c}$ such that $\underline{b} = \overline{c} \cdot \underline{a}$. The set of prefixes of the ω-word $\underline{a}$ is denoted by $\mathit{pref}(\underline{a})$. For an ω-language $\mathcal{L}$, $\mathit{pref}(\mathcal{L})$ is the language $(\cup \underline{a} \in \mathcal{L}.\, \mathit{pref}(\underline{a}))$. For a language $L \subseteq A^*$, the ω-language L^ω consists of ω-words $\underline{a}$ such that $\underline{a} = \overline{a}_0 \cdot \overline{a}_1 \cdot \overline{a}_2 \cdots$ with $\overline{a}_i \in L$ for all $i \geq 0$. In other words, the ω-words in L^ω are obtained by concatenating infinitely many words in L. Consequently, we freely use the superscript ω in regular expressions.

An ω-word $\underline{a}$ is *periodic* if there is a word $\overline{b}$ such that $\underline{a} = \overline{b}^\omega$, that is, $\underline{a}$ is obtained by concatenating infinitely many copies of the finite word $\overline{b}$. An ω-word $\underline{a}$ is *eventually periodic* if it has a periodic suffix. An eventually periodic word is of the form $\overline{a} \cdot \overline{b}^\omega$ for two words $\overline{a}$ and $\overline{b}$.

The ω-language $\mathcal{L}$ is *suffix-closed* if for every ω-word $\underline{a}$ in $\mathcal{L}$, all suffixes of $\underline{a}$ are also in $\mathcal{L}$. The ω-language $\mathcal{L}$ is *fusion-closed* if for all symbols a, if $\overline{b} \cdot a \cdot \underline{c}$ and $\overline{b}' \cdot a \cdot \underline{c}'$ are in $\mathcal{L}$, then so is $\overline{b} \cdot a \cdot \underline{c}'$. An ω-word $\underline{a}$ is called a *limit* of the language L if $\mathit{pref}(\underline{a}) \subseteq L$. An ω-word $\underline{a}$ is a *limit* of the ω-language $\mathcal{L}$ if it is a limit of $\mathit{pref}(\mathcal{L})$. In other words, $\underline{a}$ is a limit of $\mathcal{L}$ if every finite prefix of $\underline{a}$ can be extended to an ω-word in $\mathcal{L}$. The ω-language $\mathcal{L}$ is *limit-closed* if it contains all its limits: for all ω-words $\underline{a} \in A^\omega$, if $pref(\underline{a}) \subseteq pref(\mathcal{L})$, then $\underline{a} \in \mathcal{L}$; that is, if every prefix of $\underline{a}$ can be extended to an ω-word in $\mathcal{L}$, then $\underline{a}$ itself is also in $\mathcal{L}$.

Example 8.1 [ω-languages] Let $A = \{a, b\}$. Consider the ω-language $\mathcal{L}_1$ consisting of all ω-words $\underline{a} \in (a+b)^\omega$ that contain infinitely many a symbols: $\mathcal{L}_1 = (b^*a)^\omega$. Then, $pref(\mathcal{L}_1) = (a+b)^*$. The language $\mathcal{L}_1$ is suffix-closed and fusion-closed. However, $\mathcal{L}_1$ is not limit-closed: the ω-word b^ω is a limit of $\mathcal{L}_1$, but is not in $\mathcal{L}_1$.

Consider the ω-language $\mathcal{L}_2$ consisting of all ω-words $\underline{a}$ such that $a_i = a$ for all odd positions i: $\mathcal{L}_2 = ((a+b)a)^\omega$. The language $\mathcal{L}_2$ is limit-closed, but neither suffix-closed nor fusion-closed.

The ω-language $\mathcal{L}_2$ contains the periodic ω-word $(ba)^\omega$. It also contains the eventually periodic ω-word $bababa^\omega$. Not all ω-words in $\mathcal{L}_2$ are eventually periodic, for instance, the ω-word $baba^3ba^5ba^7ba^9\ldots$.

Consider the ω-language $\mathcal{L}_3$ that is the complement of the language $\mathcal{L}_1$ (with respect to A^ω). The ω-word $\underline{a}$ belongs to $\mathcal{L}_3$ iff it contains only finitely many a symbols. Thus, $\mathcal{L}_3 = (a+b)^*b^\omega$ and $pref(\mathcal{L}_3) = (a+b)^*$. The language $\mathcal{L}_3$ is suffix-closed, fusion-closed, but not limit-closed. ■

Remark 8.1 [Limit-closed ω-languages] A limit-closed ω-language $\mathcal{L}$ is completely characterized by its prefix language $pref(\mathcal{L})$: $\mathcal{L} = \{\underline{a} \mid pref(\underline{a}) \subseteq pref(\mathcal{L})\}$. ■

8.1.2 The safety-liveness distinction

Consider an ω-language $\mathcal{L}$. If for every ω-word $\underline{a}$ it can be checked whether $\underline{a}$ belongs to $\mathcal{L}$ by looking only at the finite prefixes of $\underline{a}$, then the ω-language $\mathcal{L}$ is called *safe*. If for every ω-word $\underline{a}$ it cannot be checked whether $\underline{a}$ belongs to $\mathcal{L}$ by looking at any finite prefix of $\underline{a}$, then the ω-language $\mathcal{L}$ is called *live*.

> SAFETY, LIVENESS, AND MACHINE CLOSURE
>
> Let A be a set of symbols, and let $\mathcal{L}$ be an ω-language over the alphabet A. The ω-language $\mathcal{L}$ is *safe* if $\mathcal{L}$ is limit-closed. The ω-language $\mathcal{L}$ is *live* if $pref(\mathcal{L}) = A^*$. Given a safe ω-language $\mathcal{L}_S$ and a live ω-language $\mathcal{L}_L$ over A, the pair $(\mathcal{L}_S, \mathcal{L}_L)$ is *machine-closed* if $pref(\mathcal{L}_S \cap \mathcal{L}_L) = pref(\mathcal{L}_S)$. If the pair $(\mathcal{L}_S, \mathcal{L}_L)$ is machine-closed and the ω-language $\mathcal{L}$ equals $\mathcal{L}_S \cap \mathcal{L}_L$, then the pair $(\mathcal{L}_S, \mathcal{L}_L)$ is said to be a *machine-closed specification* of $\mathcal{L}$.

Remark 8.2 [Safe and live language] The ω-language A^ω is the only ω-language over the alphabet A that is both safe and live. ■

If $\mathcal{L}$ is a safe ω-language, and $\underline{a}$ is an ω-word, then $\underline{a} \in \mathcal{L}$ iff all finite prefixes of $\underline{a}$ can be extended to ω-words in $\mathcal{L}$. If $\mathcal{L}$ is a live ω-language, and $\overline{a}$ is a word, then $\overline{a}$ can be extended to an ω-word in $\mathcal{L}$. If $(\mathcal{L}_S, \mathcal{L}_L)$ is a machine-closed pair of ω-languages, then all finite words that can be extended to ω-words in $\mathcal{L}_S$ can also be extended to ω-words in $\mathcal{L}_S \cap \mathcal{L}_L$.

Example 8.2 [Safety, liveness, and machine closure] Let $A = \{a, b\}$. The ω-language $\mathcal{L}_1 = (b^* a)^\omega$ is not safe, but is live. On the other hand, the ω-language $\mathcal{L}_2 = ((a+b)a)^\omega$ is safe, but not live. The pair $(\mathcal{L}_2, \mathcal{L}_1)$ is machine-closed, since $\mathcal{L}_1 \cap \mathcal{L}_2 = \mathcal{L}_2$. The ω-language $\mathcal{L}_3 = A^* b^\omega$ is live, but not safe. The pair $(\mathcal{L}_2, \mathcal{L}_3)$ is not machine-closed, since $\mathcal{L}_2 \cap \mathcal{L}_3$ is the empty language.

Consider the language $\mathcal{L}_4 = a^\omega + b^\omega$. The language $\mathcal{L}_4$ is safe. The pair $(\mathcal{L}_4, \mathcal{L}_1)$ is not machine-closed: no prefix of b^ω can be extended to an ω-word in $\mathcal{L}_4 \cap \mathcal{L}_1$. ■

As we will see later, the desired set of infinite trajectories of a module will be specified by a machine-closed pair of ω-languages. The safety component is specified by the transition relation, and the liveness component is specified by fairness assumptions about update choices. Machine-closure ensures that the fairness assumptions constrain only what is allowed in the limit, and can be ignored while verifying safety properties of the system. This aspect of machine-closure is captured by the following proposition.

Proposition 8.1 [Safety verification] *Let $(\mathcal{L}_S, \mathcal{L}_L)$ be a machine-closed specification of the ω-language $\mathcal{L}$, and let $\mathcal{L}'$ be a safe language. Then, $\mathcal{L} \subseteq \mathcal{L}'$ iff $\mathcal{L}_S \subseteq \mathcal{L}'$.*

Exercise 8.1 {T2} [Safety verification] Prove Proposition 8.1. ■

Requiring machine-closure is not restrictive since every ω-language can be specified by a machine-closed pair:

Theorem 8.1 [Safety-liveness decomposition] *Let A be a set of symbols. Every ω-language $\mathcal{L}$ over the alphabet A can be specified by a machine-closed pair $(\mathcal{L}_S, \mathcal{L}_L)$ consisting of a safe ω-language $\mathcal{L}_S$ and a live ω-language $\mathcal{L}_L$ over A.*

Proof. Let $\mathcal{L}_S$ be the limit closure of $\mathcal{L}$; that is, $\mathcal{L}_S$ contains all the limits of $\mathcal{L}$. Thus, $\mathcal{L}_S$ is completely characterized by $\mathit{pref}(\mathcal{L})$, and is safe. Let $\mathcal{L}_L$ be $(A^\omega \backslash \mathcal{L}_S) \cup \mathcal{L}$; that is, $\mathcal{L}_L$ contains all ω-words, except the limits of $\mathcal{L}$ not in $\mathcal{L}$. Every word is either a prefix of $\mathcal{L}$, or not a prefix of $\mathcal{L}_S$, and hence, a prefix of $(A^\omega \backslash \mathcal{L}_S)$. It follows that $\mathit{pref}(\mathcal{L}_L) = A^*$, and $\mathcal{L}_L$ is live. Since $\mathcal{L} \subseteq \mathcal{L}_S$, $\mathcal{L}_S \cap \mathcal{L}_L = \mathcal{L}$, and $(\mathcal{L}_s, \mathcal{L}_L)$ is machine-closed. ■

8.1.3 The safety-progress hierarchy

To understand the structure of ω-languages, we consider various ways of building ω-languages from languages over finite words.

Safety languages

Given a language $L \subseteq A^*$, the corresponding safety language consists of all ω-words whose all prefixes belong to L.

> SAFETY
>
> For a language $L \subseteq A^*$ over an alphabet A, $\mathit{safe}(L)$ is the ω-language $\{\underline{a} \mid \forall i \geq 0.\, \overline{a}_{0..i} \in L\}$. The ω-language $\mathcal{L} \subseteq A^\omega$ is a *safety language* if there is a language $L \subseteq A^*$ such that $\mathcal{L} = \mathit{safe}(L)$.

Remark 8.3 [Safety] If $\mathcal{L} = \mathit{safe}(L)$ then $L = \mathit{pref}(\mathcal{L})$. This implies that both definitions of safety coincide: $\mathcal{L}$ is limit-closed iff $\mathcal{L} = \mathit{safe}(L)$ for some language L. ■

While specifying requirements of a reactive module, the alphabet A corresponds to the set of observations. A safe language $\mathit{safe}(L)$, then, can be used to specify that "nothing bad ever happens" as the specification requires every possible finite trace to be in the set L. A classical safety property is the mutual exclusion property of resource allocation algorithms that requires that the same resource is not allocated to two different processes simultaneously.

Example 8.3 [Safe languages] Let $A = \{a, b\}$. The ω-language $\mathcal{L}_2 = (Aa)^\omega$ is safe, and equals $\mathit{safe}((Aa)^* + (Aa)^* A)$. The empty language is safe: $\emptyset = \mathit{safe}(\emptyset)$. The universal language A^ω is safe: $A^\omega = \mathit{safe}(A^*)$. The ω-language consisting of ω-words $\underline{a}$ such that for all $i \geq 0$, if i is a prime number, then $a_i = a$, is safe. The ω-language $\mathcal{L}_1 = (b^* a)^\omega$ is not safe. The ω-language consisting of the single ω-word a^ω is safe; however, its complement $A^* b A^\omega$ consisting of ω-words with at least one b symbol, is not safe. ■

The next proposition asserts that union of two safe languages is safe, and intersection of two safe languages is also safe. The complement of a safe language need not be safe, as illustrated in Example 8.3.

Proposition 8.2 [Closure for safety languages] *Safety languages are closed under union and intersection, but not under complementation.*

Proof. Consider $\mathcal{L}_1 = \mathit{safe}(L_1)$ and $\mathcal{L}_2 = \mathit{safe}(L_2)$. An ω-word $\underline{a}$ is in $\mathcal{L}_1 \cap \mathcal{L}_2$, iff $\underline{a} \in \mathcal{L}_1$ and $\underline{a} \in \mathcal{L}_2$, iff for all $i \geq 0$, $\overline{a}_{0...i} \in L_1$ and $\overline{a}_{0...i} \in L_2$, iff for all $i \geq 0$, $\overline{a}_{0...i} \in L_1 \cap L_2$, iff $\underline{a} \in \mathit{safe}(L_1 \cap L_2)$. This establishes that $\mathcal{L}_1 \cap \mathcal{L}_2 = \mathit{safe}(L_1 \cap L_2)$, and hence, $\mathcal{L}_1 \cap \mathcal{L}_2$ is safe.

To establish closure under union, let L'_1 be the language $\{\overline{a} \mid \mathit{pref}(\overline{a}) \subseteq L_1\}$ consisting of words all of whose prefixes are in L_1. Similarly, let $L'_2 = \{\overline{a} \mid \mathit{pref}(\overline{a}) \subseteq L_2\}$. We prove that $\mathcal{L}_1 \cup \mathcal{L}_2 = \mathit{safe}(L'_1 \cup L'_2)$.

Consider an ω-word $\underline{a} \in \mathcal{L}_1 \cup \mathcal{L}_2$. Without loss of generality, suppose $\underline{a} \in \mathcal{L}_1$. Then, for all $i \geq 0$, $\overline{a}_{0\ldots i} \in L_1$. Hence, for all $i \geq 0$, for all $0 \leq j \leq i$, $\overline{a}_{0\ldots j} \in L_1$. Hence, for all $i \geq 0$, $\overline{a}_{0\ldots i} \in L'_1$. Hence, $\underline{a} \in \mathit{safe}(L'_1 \cup L'_2)$.

Consider an ω-word $\underline{a} \in \mathit{safe}(L'_1 \cup L'_2)$. Then, for all $i \geq 0$, $\overline{a}_{0\ldots i} \in L'_1 \cup L'_2$. Without loss of generality, for infinitely many positions i, $\overline{a}_{0\ldots i} \in L'_1$. This implies for all $i \geq 0$, $\overline{a}_{0\ldots i} \in L_1$ (for, if $\overline{a}_{0\ldots j} \notin L_1$ for some j, then for all $i \geq j$, $\overline{a}_{0\ldots i} \notin L'_1$). Hence, $\underline{a} \in \mathit{safe}(L_1)$. Hence, $\underline{a} \in \mathcal{L}_1 \cup \mathcal{L}_2$. ■

Exercise 8.2 {T2} [*safe* does not distribute over union] Show that $\mathit{safe}(L_1) \cup \mathit{safe}(L_2)$ is not necessarily equal to $\mathit{safe}(L_1 \cup L_2)$. ■

Guarantee languages

Given a language $L \subseteq A^*$, the corresponding guarantee language consists of all ω-words whose *some* prefix belongs to L.

> GUARANTEE
> For a language $L \subseteq A^*$ over an alphabet A, $\mathit{guar}(L)$ is the ω-language $\{\underline{a} \mid \exists i \geq 0.\, \overline{a}_{0..i} \in L\}$. The ω-language $\mathcal{L} \subseteq A^\omega$ is a *guarantee language* if there is a language $L \subseteq A^*$ such that $\mathcal{L} = \mathit{guar}(L)$.

Remark 8.4 [Guarantee] The ω-language $\mathcal{L} \subseteq A^\omega$ is a guarantee language iff there is a language $L \subseteq A^*$ such that $\mathcal{L} = L \cdot A^\omega$. ■

While specifying requirements of a reactive module, a guarantee language $\mathit{guar}(L)$ can be used to specify that "something good eventually happens" as the specification requires the module to produce a trace in L after executing for finitely many steps. A classical guarantee property is the termination requirement that a program eventually produces an output.

Example 8.4 [Guarantee languages] Let $A = \{a, b\}$. The empty language is a guarantee language: $\emptyset = \mathit{guar}(\emptyset)$. The universal language A^ω is also a guarantee language: $A^\omega = \mathit{guar}(A^+)$. The ω-language A^*bA^ω consisting of ω-words with at least one b symbol, is a guarantee language: $A^*bA^\omega = \mathit{guar}(A^*b)$. ■

The safety and guarantee languages are closely related, namely, they are duals of each other: the complement of a safe language is a guarantee language, and the complement of a guarantee language is a safe language.

Proposition 8.3 [Duality of safety and guarantee] *The ω-language $\mathcal{L}$ is a safety language iff the complementary language $A^\omega \backslash \mathcal{L}$ is a guarantee language.*

Exercise 8.3 {T2} [Duality of safety and guarantee] Prove that $\mathcal{L} = \mathit{safe}(L)$ iff $A^\omega \backslash \mathcal{L} = \mathit{guar}(A^+ \backslash L)$. Proposition 8.3 follows. ■

Exercise 8.4 {T2} [Safe and guarantee languages] Characterize the class of ω-languages that are both safety and guarantee languages. ■

Since safe languages are closed under union and intersection, it follows that so are guarantee languages.

Proposition 8.4 [Closure for guarantee languages] *Guarantee languages are closed under union and intersection, but not under complementation.*

Remark 8.5 [Closure for guarantee languages] The closure properties of guarantee languages can be established directly also:

$$\mathit{guar}(L_1) \cup \mathit{guar}(L_2) \;=\; \mathit{guar}(L_1 \cup L_2),$$

and

$$\mathit{guar}(L_1) \cap \mathit{guar}(L_2) \;=\; \mathit{guar}((L_1 \cdot A^*) \cap (L_2 \cdot A^*)).$$

■

Exercise 8.5 {T4} [Obligation languages] Obligation languages are obtained by boolean combinations of safety or guarantee languages. In other words, the set of obligation languages is the least set that contains all safety languages, and is closed under union, intersection, and complementation. Every obligation language can be expressed in a normal form: $\bigcup 0 \le i \le k.\mathit{safe}(L_i) \cap \mathit{guar}(L'_i)$. For example, for $A = \{a, b\}$, the ω-language a^*ba^ω consisting of ω-words with precisely one b symbol, is an obligation language: $\mathit{safe}(a^*ba^* + a^*) \cap \mathit{guar}(a^*b)$. Show that the obligation language a^*ba^ω is neither a safety language nor a guarantee language. ■

Response languages

Given a language $L \subseteq A^*$, the corresponding response language consists of all ω-words whose infinitely many prefixes belong to L.

> RESPONSE
>
> For a language $L \subseteq A^*$ over an alphabet A, $\mathit{recur}(L)$ is the ω-language $\{\underline{a} \mid \forall j \ge 0.\, \exists i \ge j.\, \overline{a}_{0..i} \in L\}$. The ω-language $\mathcal{L} \subseteq A^\omega$ is a *response language* if there is a language $L \subseteq A^*$ such that $\mathcal{L} = \mathit{recur}(L)$.

While specifying requirements of a reactive module, a response language $\mathit{recur}(L)$ is used to specify that "something good happens repeatedly" as the specification requires infinitely many traces to be in the set L. A classical response property is the progress requirement for resource allocation algorithms: if some process is requesting a resource then some process is eventually granted the resource.

Example 8.5 [Response languages] Let $A = \{a, b\}$, and let $L = (a^*b)^*$. Then, $\mathit{recur}(L) = (a^*b)^\omega$ is the corresponding response language, and consists of all ω-words with infinitely many b symbols. Observe that $\mathit{recur}(L)$ is neither a safety language nor a guarantee language. ■

The next proposition asserts that union of two response languages is a response language, and intersection of two response languages is also a response language. However, the response languages are not closed under complementation.

Proposition 8.5 [Closure for response languages] *Response languages are closed under union and intersection, but not under complementation.*

Proof. Consider $\mathcal{L}_1 = \mathit{recur}(L_1)$ and $\mathcal{L}_2 = \mathit{recur}(L_2)$. An ω-word $\underline{a}$ is in $\mathcal{L}_1 \cup \mathcal{L}_2$, iff $\underline{a} \in \mathcal{L}_1$ or $\underline{a} \in \mathcal{L}_2$, iff for infinitely many i, $\overline{a}_{0\ldots i} \in L_1$ or for infinitely many i, $\overline{a}_{0\ldots i} \in L_2$, iff for infinitely many i, $\overline{a}_{0\ldots i} \in L_1 \cup L_2$, iff $\underline{a} \in \mathit{recur}(L_1 \cup L_2)$. This establishes that $\mathcal{L}_1 \cup \mathcal{L}_2 = \mathit{recur}(L_1 \cup L_2)$, and hence, $\mathcal{L}_1 \cup \mathcal{L}_2$ is a response language.

For closure under intersection, consider the language L_{12} that contains a word $\overline{a}_{0\ldots m}$ iff (1) $\overline{a}_{0\ldots m} \in L_2$, and (2) there exits $i < m$ such that $\overline{a}_{0\ldots i} \in L_1$ and $\overline{a}_{0\ldots k} \notin L_2$ for $i < k < m$. We prove that $\mathcal{L}_1 \cap \mathcal{L}_2 = \mathit{recur}(L_{12})$.

Consider an ω-word $\underline{a} \in \mathit{recur}(L_{12})$. There exists an infinite increasing sequence of integers $i_0, i_1 \ldots$ such that for all $j \geq 0$, $\overline{a}_{0\ldots i_j}$ is in L_{12}. By definition of L_{12}, for all $j \geq 0$, (1) $\overline{a}_{0\ldots i_j}$ is in L_2, and (2) there exists i'_j such that $i_{j-1} \leq i'_j < i_j$ and $\overline{a}_{0\ldots i'_j}$ is in L_1. Thus, $\underline{a} \in \mathit{recur}(L_1)$, and $\underline{a} \in \mathit{recur}(L_2)$. Hence, $\underline{a} \in \mathcal{L}_1 \cap \mathcal{L}_2$.

Consider an ω-word $\underline{a} \notin \mathit{recur}(L_{12})$. Then there exists a position i such that for all $j \geq i$, $\overline{a}_{0\ldots j} \notin L_{12}$. We wish to establish $\underline{a} \notin \mathcal{L}_1 \cap \mathcal{L}_2$. Assume to the contrary. Since $\underline{a} \in \mathit{recur}(L_1)$, there exists a position $k \geq i$ such that $\overline{a}_{0\ldots k} \in L_1$. Let k' be the least position such that $k' > k$ and $\overline{a}_{0\ldots k'} \in L_2$ (such a position exists since $\underline{a} \in \mathit{recur}(L_2)$). By definition of L_{12}, it contains $\overline{a}_{0\ldots k'}$, leading to a contradiction.

For non-closure under complement, let $A = \{a, b\}$. Let $L = (a^*b)^*$. The response language $\mathit{recur}(L)$ consists of all ω-words with infinitely many b symbols. Consider the complement of $\mathit{recur}(L)$, that is, the ω-language $\mathcal{L} = (a + b)^*a^\omega$ that contains ω-words with only finitely many b symbols. We prove that $\mathcal{L}$ is not a response language. Suppose, to the contrary, $\mathcal{L} = \mathit{recur}(L')$. We show that there exists a sequence of integers $i_0, i_1, \ldots$ such that for all $j \geq 0$, the word $a^{i_0}ba^{i_1}b\cdots a^{i_j}$ is in L'. The proof is by induction.

The ω-word a^ω is in $\mathcal{L}$. Hence, infinitely many prefixes of it are in L', and hence, there exists an integer i_0 such that $a^{i_0} \in L'$.

Assume that there exist integers $i_0, i_1, \ldots i_k$ such that for all $0 \leq j \leq k$, the word $a^{i_0} b a^{i_1} b \cdots a^{i_j}$ is in L'. Consider the ω-word $a^{i_0} b a^{i_1} b \cdots a^{i_k} b a^\omega$. Since it belongs to $\mathcal{L}$, it has infinitely many prefixes in L', and in particular, there exists an integer i_{k+1} such that $a^{i_0} b a^{i_1} b \cdots a^{i_k} b a^{i_{k+1}}$ is in L'.

Now consider ω-word $a^{i_0} b a^{i_1} b \cdots$. It has infinitely many prefixes in L', but it contains infinitely many b symbols, and is not in $\mathcal{L}$. ■

Exercise 8.6 {T3} [*recur* does not distribute over intersection] Show that $recur(L_1) \cap recur(L_2)$ is not necessarily equal to $recur(L_1 \cap L_2)$. ■

Proposition 8.6 [Hierarchy of languages] *Every safety language and every guarantee language is a response language.*

Proof. To establish that every safety language is a response language, verify that $safe(L) = recur(pref(L))$. To establish that every guarantee language is a response language, verify that $guar(L) = recur(L \cdot A^*)$. ■

It follows that every obligation language is also a response language.

Exercise 8.7 {T4} [From guarantee to response] Prove that the ω-language $\mathcal{L}$ is a response language iff $\mathcal{L}$ is the intersection of countably many guarantee languages. ■

Exercise 8.8 {T3} [ω-repetition and response] Show that for every language $L \subseteq A^*$, the ω-language L^ω is a response language. ■

Persistence languages

Given a language $L \subseteq A^*$, the corresponding persistence language consists of all ω-words whose all, but finitely many, prefixes belong to L.

> PERSISTENCE
>
> For a language $L \subseteq A^*$ over an alphabet A, $persist(L)$ is the ω-language $\{\underline{a} \mid \exists j \geq 0. \forall i \geq j. \overline{a}_{0..i} \in L\}$. The ω-language $\mathcal{L} \subseteq A^\omega$ is a *persistence language* if there is a language $L \subseteq A^*$ such that $\mathcal{L} = persist(L)$.

While specifying requirements of a reactive module, a persistence language $persist(L)$ is used to specify that "something good happens eventually, and stays unchanged" as the specification requires all, but finitely many, traces to be in the set L. A classical persistence property is the eventual stabilization requirement for self-stabilizing algorithms: the system eventually attains the stable state, and stays stable.

Example 8.6 [Persistence languages] Let $A = \{a, b\}$, and let $L = A^*a^*$. Then, $persist(L) = A^*a^\omega$ is the corresponding persistence language, and consists of all ω-words with only finitely many b symbols. Observe that $persist(L)$ is neither a safety language nor a guarantee nor a response language, and its complement is a response language. ■

The response and persistence languages are closely related, namely, they are duals of each other: the complement of a response language is a persistence language, and the complement of a persistence language is a response language.

Proposition 8.7 [Duality of response and persistence] *The ω-language $\mathcal{L}$ is a response language iff the complementary language $A^\omega \backslash \mathcal{L}$ is a persistence language.*

Exercise 8.9 {T2} [Duality of response and persistence] Prove that $\mathcal{L} = recur(L)$ iff $A^\omega \backslash \mathcal{L} = persist(A^* \backslash L)$. Proposition 8.7 follows. ■

It follows that every safety or guarantee language is a persistence language.

Exercise 8.10 {T4} [Response and persistence languages] Show that an ω-language is both a response language and a persistence language iff it is an obligation language. Show that the response language $(a^*b)^\omega$ is not an obligation language, and thus, the persistence language A^*a^ω is not an obligation language. ■

Since response languages are closed under union and intersection, it follows that so are persistence languages.

Proposition 8.8 [Closure for persistence languages] *Persistence languages are closed under union and intersection, but not under complementation.*

Exercise 8.11 {T3} [From safety to persistence] Prove that the ω-language $\mathcal{L}$ is a persistence language iff $\mathcal{L}$ is the union of countably many safety languages. ■

Reactivity languages

Reactivity languages are obtained by boolean combinations of response and persistence languages. In other words, the set of reactivity languages is the least set that contains all response languages, and is closed under union, intersection, and complementation.

REACTIVITY

The ω-langauge $\mathcal{L} \subseteq A^\omega$ is a *1-reactivity language* if there exists a persistence language $\mathcal{L}_1$ and a response language $\mathcal{L}_2$ such that $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$. The ω-langauge $\mathcal{L} \subseteq A^\omega$ is a *k-reactivity language*, for a natural number k, if there exist k 1-reactivity languages $\mathcal{L}_1, \ldots \mathcal{L}_k$ such that $\mathcal{L} = \mathcal{L}_1 \cap \cdots \cap \mathcal{L}_k$. The ω-langauge $\mathcal{L} \subseteq A^\omega$ is a *reactivity language* if it is a k-reactivity language for some natural number k.

Remark 8.6 [Disjunctive form of reactivity] Every reactivity language $\mathcal{L}$ can, alternatively, be expressed in a disjunctive normal form: $\bigcup 0 \leq i \leq k.recur(L_i) \cap persist(L_i')$. ■

A typical 1-reactivity requirement is the conditional repetition: if the symbol a repeats infinitely often, then the symbol b also repeats infinitely often. As we will see shortly, reactivity languages are useful in specification of fairness requirements for reactive modules: an individual strong fairness requirement is a 1-reactivity language.

Example 8.7 [Reactivity languages] Let $A = \{a, b, c\}$. The ω-language consisting of ω-words with infinitely many b symbols or only finitely many a symbols is a 1-reactivity language: $recur((A^*b)^*) \cup persist(A^*(b+c)^*)$. The ω-language consisting of ω-words with infinitely many b symbols and only finitely many a symbols is a 2-reactivity language: $recur((A^*b)^*) \cap persist(A^*(b+c)^*)$. ■

Exercise 8.12 {T5} [Hierarchy of reactivity languages] Show that there is a 1-reactivity language that is neither a response language nor a persistence language. Then, show that, for every natural number k, there is a k-reactivity language that is not a $(k-1)$-reactivity language. ■

By definition, reactivity languages are closed under all boolean operations.

Proposition 8.9 [Closure for reactivity languages] *Reactivity languages are closed under all boolean operations.*

All the ω-languages of interest to us will be reactivity languages. Let us recap the construction of ω-languages starting from languages of words. A safety language is the set of limits of a language over words. Safe languages are closed under union and intersection, but complementing a safe language gives a guarantee language. By considering intersection of infinitely (countable) many guarantee languages, we obtain response languages. Response languages are closed under union and intersection, but complementing a response language gives a persistence language. Equivalently, persistence languages are obtained by infinite unions of safety languages. Boolean combinations of persistence and response languages give reactivity languages. The relationship among these classes is illustrated in Figure 8.1.

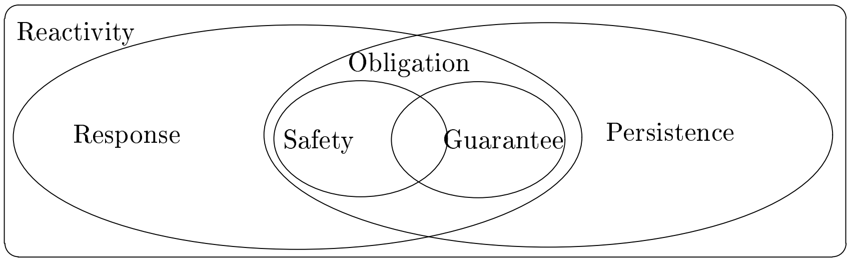

Figure 8.1: Classes of ω-languages

Exercise 8.13 {T5} [Topological characterization] Consider a metric on ω-words such that the distance between two ω-words shrinks exponentially with the length of the longest common prefix. In particular, define the metric d over the set A^ω such that $d(\underline{a}, \underline{b})$ equals 0 if $\underline{a} = \underline{b}$, and 2^{-i} otherwise, where i is the maximum integer j such that $\overline{a}_{0\ldots j} = \overline{b}_{0\ldots j}$. (1) Prove that the safe languages are precisely the closed sets of the resulting topology on ω-words. (2) Prove that the live languages are precisely the dense sets. (3) Which languages correspond to the open sets? ■

Exercise 8.14 {T4} [Machine closure in safety-progress hierarchy] Prove that every C-language is the machine-closed intersection of a safe C-language and a live C-language, where C is one of the following classes: safety; guarantee; obligation; response; persistence; reactivity. ■

8.1.4 ω-Trajectories

The execution of a transition graph G for finitely many steps results in a trajectory of G, which is a finite sequence of states. The execution of a transition graph for infinitely many steps results in a ω-trajectory of G, which is an infinite sequence of states.

> ω-TRAJECTORY
>
> Let $G = (\Sigma, \sigma^I, \rightarrow)$ be a transition graph. An *ω-trajectory* of G is an ω-word $\underline{s} = s_0 s_1 s_2 \ldots$ over the alphabet Σ of states such that for all $i \geq 0$, $s_i \rightarrow s_{i+1}$. The first state s_0 is the source. The ω-trajectory $\underline{s}$ is *initialized* if $s_0 \in \sigma^I$. The *ω-language* $\mathcal{L}_G$ of the transition graph G is the set of initialized ω-trajectories of G.

Remark 8.7 [Seriality] Let G be a serial transition graph. Then, for every state s of G, there is a source-s trajectory of G. The ω-language $\mathcal{L}_G$ is nonempty. ■

Remark 8.8 [Safety of graph languages] The ω-language $\mathcal{L}_G$ of the transition graph G is safe, and equals $safe(L_G)$. ■

Exercise 8.15 {T3} [ω-languages of transition graphs] Prove that the ω-language $\mathcal{L}_G$ of a transition graph G is limit-closed and fusion-closed. Conversely, let $\mathcal{L}$ be a limit-closed and fusion-closed ω-language over the alphabet A. Prove that there exists a transition graph G with states A such that $\mathcal{L}_G = \mathcal{L}$. ■

Example 8.8 [ω-trajectories of mutual exclusion protocol] Let us revisit the asynchronous solution to the mutual exclusion problem (Figure 1.23). The initialized ω-trajectories of *Pete* can be obtained from the reachable subgraph of G_{Pete} (see Figure 2.4). One possible ω-trajectory *Pete* is the periodic trajectory

$$[(o0o0)(r0o0)(i0o0)]^\omega$$

in which process P_1 repeatedly requests and enters its critical section, while process P_2 stays idle. Another possible ω-trajectory is the periodic trajectory

$$[(o0o0)(r0o0)(i0o0)(o0r1)(o0i1)(r1o1)(i1o1)(o1r0)(o1i0)(r0o0)(i0o0)]^\omega$$

in which both processes alternately request and enter thir critical sections. Since all the atoms of *Pete* are lazy, each state has a transition to itself. Consequently, $(o0o0)^\omega$ is also a ω-trajectory of *Pete*. Finally, consider the eventually periodic ω-trajectory $(o0o0)(r0o0)^\omega$ in which process P_1 requests the critical section, but never enters the critical section. ■

8.2 Fairness

Fair modules are obtained from modules by adding two types of fairness requirements.

8.2.1 Weak Fairness

A nondeterministic update command may offer, for a given state, several choices for updating the variables. For instance, consider the module *AsyncCount*:

> **module** *AsyncCount* **is**
>   **interface** $Count\colon \mathbb{N}$
>   **atom controls** *count*
>   **init**
> $[\!]\ true \to count' := 0$
>   **update**
> $[\!]\ true \xrightarrow{\alpha} count' := count + 1$
> $[\!]\ true \xrightarrow{\beta} count' := count$

The counter is initially zero. The update action of the module has two guarded assignments. The guarded assignment α is enabled in every update round, and increments the counter. The guarded assignment β is also enabled in every update round, and leaves the counter unchanged. During the execution of the module, the choice between executing α and executing β is nondeterministic. Thus, for every natural number i, the counter may stay unchanged for the first i update rounds, and get updated to 1 in the round $(i+1)$. This is a convenient abstraction of the assumption that the rate at which the counter is incremented is unknown (or irrelevant). However, consider the limit ω-trajectory $\underline{s}$ in which the counter never gets updated: for every $i \geq 0$, $s_i[count] = 0$. The ω-trajectory $\underline{s}$ is *unfair* to the update choice α; the choice α is enabled in every round, and never executed.

Definition of fair modules provides a way to rule out the ω-trajectory $\underline{s} = 0^\omega$. This is achieved by requiring that the resolution of the update choices be *weakly fair* to the choice α. The module *FairCount* is a fair version of the asynchronous counter *AsyncCount*:

> **module** *FairCount* **is**
>   **interface** $Count\colon \mathbb{N}$
>   **atom controls** *count*
>   **init**
> $[\!]\ true \rightarrow count' := 0$
>   **update weaklyfair** α
> $[\!]\ true \xrightarrow{\alpha} count' := count + 1$
> $[\!]\ true \xrightarrow{\beta} count' := count$

The annotation **weaklyfair** α requires that the guarded command α is executed infinitely often. The ω-trajectories that satisfy this requirement will be called fair trajectories. The ω-trajectory $\underline{s} = 0^\omega$ is not a fair trajectory of *FairCount*. The ω-trajectory $\underline{t} = 012345^\omega$ in which the counter is not incremented beyond 5, is also not a fair trajectory of *FairCount*. On the other hand, consider the ω-trajectory $\underline{u} = 0123\cdots$ in which the counter is incremented in every round. The update choice β is always enabled, but never executed, and yet, the ω-trajectory $\underline{u}$ is a fair trajectory of *FairCount*. This is because *FairCount* makes no assumption about fairness towards the choice β.

In general, an update choice α for an atom U of a module P is a subset of the update action update_U. Consider an ω-trajectory $\underline{s}$ of P. For $i \geq 1$, consider the update round i in which the state s_i is determined from the state s_{i-1}. Recall that the atom U is executed only after the updated values of the variables in $\mathsf{await}X_U$ have been determined. The update choice α is said to be *available* in round i, if the state s_{i-1}, together with the values of the awaited variables in state s_i, satisfies the guard p_γ of some guarded assignment γ in α. The update choice is said to be *executed* in round i, if the values of the controlled variables in state s_i are determined by executing some available guarded assignment γ in α. The ω-trajectory $\underline{s}$ is *weakly fair* with respect to α, if there is no round i such that the choice α is available in every round after i, and is not executed in any round after i. Intuitively, a weakly fair update choice cannot be available forever without being executed.

8.2.2 Strong Fairness

Weak fairness requires that a choice that is continuously available is eventually executed. Suppose a choice is available in all even rounds, and unavailable in all odd rounds, and is never executed. This scenario meets the requirement of weak fairness, but may not be a reasonable scenario in certain cases. For example, consider the module *LossyBuffer*:

> **module** *LossyBuffer* **is**
>   **interface** $y : \mathbb{E}$
>   **external** $x : \mathbb{E}$
>   **passive atom controls** y **reads** x, y **awaits** x
>   **update**
> $[\!]\ x? \xrightarrow{\alpha} y?$
> $[\!]\ x? \xrightarrow{\beta}$

In every round in which the external event x is present, both the update choices α and β are available. If the choice α is executed, then the interface event y is issued, and if the choice β is executed, then the module stutters without issuing the event y. The module *LossyBuffer* can be viewed as an abstraction of a lossy buffer, that either outputs the input event, or loses the input event. Consider the periodic ω-trajectory

$$\underline{s} \;=\; [(y=0, x=0)(y=0, x=1)]^{\omega};$$

in every update round the external event x is present, but the module always executes the update choice β. Requiring weak fairness for the choice α will rule out the ω-trajectory $\underline{s}$. Now consider the periodic ω-trajectory

$$\underline{t} \;=\; [(y=0, x=0)(y=0, x=0)(y=0, x=1)(y=0, x=1)]^{\omega};$$

the external event x is present only in alternate rounds, and the module always executes the update choice β. Note that the ω-trajectory $\underline{t}$ is weakly fair with respect to the choice α, because the choice α is unavailable in infinitely many rounds. If we wish to model the assumption that only some, but not all, messages are lost, then we would like to rule out the ω-trajectory $\underline{t}$ also. This can be achieved by requiring that the resolution of the update choices be ***strongly fair*** to the choice α: if α is available in infinitely many round, then α is executed in infinitely many rounds. The module *FairBuffer* is a fair version of the buffer *LossyBuffer*:

module *FairBuffer* **is**
  interface $y : \mathbb{E}$
  external $x : \mathbb{E}$
  passive atom controls y **reads** x, y **awaits** x
    update stronglyfair α
      $\| \; x? \xrightarrow{\alpha} y?$
      $\| \; x? \xrightarrow{\beta}$

The annotation **stronglyfair** α classifies an ω-trajectory to be fair if either the update choice α is executed infinitely often, or is available only finitely often. Consequently, the ω-trajectory $\underline{t}$ is not a fair trajectory of *FairBuffer*. Intuitively, a strongly fair update choice cannot be available infinitely often without being executed.

A fair module may declare, for each atom, some of its choices to be weakly fair, and some to be strongly fair. An *update choice* of a reactive module P is a subset of the update command update_U, for some atom $U \in \mathsf{atoms}_P$.

> Fair module
>
> A *fair module* $\mathcal{P}$ consists of (1) a reactive module P, (2) [weak fairness] a finite set WeakF_P of update choices of P, and (3) [strong fairness] a finite set $\mathit{StrongF}_P$ of update choices of P.

Since one of the components of a fair module is a reactive module, we freely attribute the properties of a reactive module to a fair module. For instance, every fair module $\mathcal{P} = (P, \mathit{WeakF}_P, \mathit{StrongF}_P)$ defines the transition graph $G_{\mathcal{P}} = G_P$. Different classifications of modules, such as finite versus infinite, closed versus open, apply to fair modules also. For instance, an asynchronous fair module is a fair module all of whose interface variables are controlled by lazy atoms.

8.3 Fair Graphs

We define the semantics of a fair module by associating a fair graph with it. In Chapter 6, we defined automata by augmenting observation structures with accepting regions. The accepting region of an automaton classifies trajectories into accepting and non-accepting, and consequently, automata can define languages that are not necessarily prefix-closed. Now we wish to augment a transition graph with an accepting condition that will classify its ω-trajectories into accepting and non-accepting. By considering only accepting ω-trajectories, we will be able to define live ω-languages.

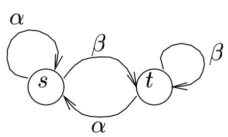

Figure 8.2: Fair graph

Let $G = (\Sigma, \sigma^I, \rightarrow)$ be a transition graph. An *action* of G is a subset of the transition relation $\rightarrow$. For an action α of G, we write $s \xrightarrow{\alpha} t$ if the transition (s,t) belongs to α. A *fairness constraint* f for a transition graph G is a pair (α, β) of actions of G, and a *fairness assumption* F for G is a finite set of fairness constraints for G.

> FAIR GRAPH
>
> A *fair graph* $\mathcal{G}$ consists of (1) a transition graph G, and (2) [fairness] a fairness assumption F for G.

Intuitively, a fairness constraint (α, β) requires that if the action α repeats infinitely often then the action β also repeats infinitely often. Fair trajectories of a fair graph are those ω-trajectories that meet the requirements stipulated by all the fairness constraints.

> FAIR TRAJECTORY
>
> Let G be a transition graph. An ω-trajectory $\underline{s}$ of of G is *α-fair*, for an action α of G, if $s_i \xrightarrow{\alpha} s_{i+1}$ for infinitely many natural numbers i. The ω-trajectory $\underline{s}$ is *f-fair*, for a fairness constraint $f = (\alpha, \beta)$ of G, if either $\underline{s}$ is not α-fair, or $\underline{s}$ is β-fair. The ω-trajectory $\underline{s}$ is *F-fair*, for a fairness assumption F of G, if $\underline{s}$ is f-fair for all fairness constraints f in F. A *fair trajectory* of a fair graph $\mathcal{G} = (G, F)$ is an F-fair ω-trajectory of G. The *fair language* $\mathcal{L}_{\mathcal{G}}$ of a fair graph $\mathcal{G}$ is the set of initialized fair trajectories of $\mathcal{G}$.

Remark 8.9 [Fair graphs] Let $\mathcal{G} = (G, F)$ be a fair graph. The ω-language $\mathcal{L}_{\mathcal{G}}$ is a subset of the safe language $\mathcal{L}_G$. Furthermore, if the fairness assumption F is an empty set then every ω-trajectory is fair, and $\mathcal{L}_{\mathcal{G}}$ equals $\mathcal{L}_G$. ■

Fair languages are not necessarily safe languages, and different fairness assumptions can be used to identify different subsets of the ω-trajectories of a transition graph.

Example 8.9 [Fair graph] Consider the two-state transition graph shown in Figure 8.2. Both states are initial. The actions α and β contain two transitions each, as shown. An ω-trajectory $\underline{t}$ is α-fair if it contains infinitely many visits to the state s, and $\underline{t}$ is β-fair if it contains infinitely many visits to the state t. Consider the fairness constraint $f_1 = (\rightarrow, \beta)$. The ω-trajectory $\underline{t}$ is f_1-fair if it contains infinitely many visits to the state t. Thus, the fair language of the fair graph $(G, \{f_1\})$ is $(s^*t)^\omega$ (note that this is a response language).

Different fairness constraints can define the same language. For instance, for the fairness constraint $f_2 = (\alpha, \beta)$, the ω-trajectory $\underline{t}$ is f_2-fair iff it is f_1-fair. Thus, the fair languages of $(G, \{f_1\})$ and $(G, \{f_2\})$ coincide.

Let $f_3 = (\rightarrow, \alpha)$. Then, an ω-trajectory is f_3-fair if it contains infinitely many visits to the state s. The fair language of the fair graph $(G, \{f_1, f_3\})$ contains ω-trajectories that have infinitely many visits to both the states, and equals $(s^*t)^\omega \cap (t^*s)^\omega$ (note that this is a reactivity language).

Consider the fairness constraint $f_4 = (\alpha, \emptyset)$. Observe that there is no $\emptyset$-fair trajectory. Thus, an ω-trajectory $\underline{t}$ is f_4-fair iff it is not α-fair; that is, if it contains only finitely many visits to the state s. The fair language of the fair graph $(G, \{f_4\})$ is $(s+t)^*t^\omega$ (note that this is a persistence language). ■

The above example shows that fair languages of fair graphs can be reactivity languages. Can fair graphs define more complex languages? The answer is no.

Proposition 8.10 [Languages of fair graphs] *If $\mathcal{G}$ is a fair graph, then the language $\mathcal{L}_\mathcal{G}$ is a reactivity language.*

Proof. Let G be a transition graph. For an action α, let L_α be the set of initialized trajectories $\overline{s}_{0\ldots m}$ of G such that $s_{m-1} \xrightarrow{\alpha} s_m$. Now, an initialized ω-trajectory $\underline{s}$ of G is α-fair iff $\underline{s}$ belongs to the response language $recur(L_\alpha)$. For a fairness constraint $f = (\alpha, \beta)$, an initialized ω-trajectory $\underline{s}$ of G is f-fair iff it is either α-unfair or β-fair; that is, iff it belongs to $(\Sigma^\omega \backslash recur(L_\alpha)) \cup recur(L_\beta)$. Thus, the set of f-fair ω-trajectories is a union of a persistence and a response language, that is, a 1-reactivity language.

Consider the fair graph $\mathcal{G} = (G, F)$. Verify that $\mathcal{L}_\mathcal{G}$ equals

$$\bigcap (\alpha, \beta) \in F.\ (\Sigma^\omega \backslash recur(L_\alpha)) \cup recur(L_\beta).$$

It follows that if F has k fairness constraints then $\mathcal{L}_\mathcal{G}$ is a k-reactivity language. ■

Types of fairness constraints

We consider three types of fairness constraints.

Weak-fair constraint

Let $G = (\Sigma, \sigma^I, \rightarrow)$ be a transition graph. A fairness constraint (α, β) for G is a *weak-fair* constraint if α equals $\rightarrow$. A weak-fair graph is a fair graph (G, F) such that F contains only weak-fair constraints.

While a fairness constraint specifies infinite repetition of an action conditioned upon the repetition of another, a weak-fair constraint specifies unconditional repetition of an action.

Remark 8.10 [Weak-fair constraints] For a weak-fair constraint $f = (\rightarrow, \beta)$, f-fair trajectories are precisely the β-fair trajectories. For a weak-fair graph $\mathcal{G} = (G, F)$, $\mathcal{L}_{\mathcal{G}}$ is the response language

$$\bigcap(\rightarrow, \beta) \in F.\ recur(L_\beta).$$

■

Sometimes we consider actions that are defined by regions. For a region σ of a transition graph G, the action $\alpha_\sigma = \{(s,t) \mid s \in \sigma \text{ and } s \rightarrow t\}$ contains all transitions with source in σ. Consequently, we will use regions in place of actions when there is no cause for confusion. For instance, an ω-trajectory $\underline{s}$ is σ-fair, for a region σ, if it is α_σ-fair, or equivalently, if $s_i \in \sigma$ for infinitely many i. For two regions σ and τ, the fairness constraint $(\alpha_\sigma, \alpha_\tau)$ will be denoted, more concisely, as (σ, τ).

Machine closure

A fair graph $\mathcal{G}$ is said to be machine-closed if every trajectory of $\mathcal{G}$ can be extended to a fair trajectory. Machine closure ensures that a stepwise simulator for fair graphs cannot paint itself into a corner from which the fairness constraints cannot be satisfied. Machine closure for fair graphs is the analog of seriality for transition graphs.

Machine-closed fair graph

A fair graph $\mathcal{G}$ is machine-closed if for every state s of $\mathcal{G}$ there exists a source-s fair trajectory of $\mathcal{G}$.

Remark 8.11 [Machine-closed fair graph] If $\mathcal{G} = (G, F)$ is machine-closed then $pref(\mathcal{L}_{\mathcal{G}}) = pref(\mathcal{L}_G) = L_G$. ■

Remark 8.12 [Machine-closure] Recall the definition of machine-closure from Section 8.1.2: for a safe language $\mathcal{L}_S$ and a live language $\mathcal{L}_L$, the pair $(\mathcal{L}_S, \mathcal{L}_L)$ is machine-closed if every prefix of $\mathcal{L}_S$ can be extended to an ω-word in $\mathcal{L}_S \cap \mathcal{L}_L$.

The above definition of machine-closure has the same spirit. To be precise, consider a fair graph (G, F). Let $\mathcal{L}_S$ be the set of all ω-trajectories of G (this is a superset of $\mathcal{L}_G$ that contains only initialized ω-trajectories). Verify that the set $\mathcal{L}_S$ is safe. Let $\mathcal{L}_L$ be the set of all ω-words $\underline{s}$ over the alphabet Σ_G that are f-fair for every fairness constraint $f \in F$. This set includes all fair trajectories of $\mathcal{G}$, along with ω-words that are not necessarily ω-trajectories of G. Alternatively, the set $\mathcal{L}_L$ is the fair language of a fair graph with state-space Σ_G, initial region Σ_G, transition relation $\Sigma_G \times \Sigma_G$, and fairness assumption F. Verify that the set $\mathcal{L}_L$ is live. Now, the pair $(\mathcal{L}_S, \mathcal{L}_L)$ is machine-closed, that is, $\mathit{pref}(\mathcal{L}_S \cap \mathcal{L}_L) = \mathit{pref}(\mathcal{L}_S)$ iff the fair graph $\mathcal{G}$ is machine-closed, that is, every state is the source of some fair trajectory. ■

Exercise 8.16 $\{$`T3`$\}$ [Intersection and machine-closure] Suppose f_1 and f_2 are two fairness constraints for a transition graph G. Prove or disprove the claim that $(G, \{f_1, f_2\})$ is machine-closed iff both the fair graphs $(G, \{f_1\})$ and $(G, \{f_2\})$ are machine-closed. ■

Local fairness

A local fairness constraint is a type of fairness constraint that, intuitively, stipulates a fair resolution of choice, and nothing more.

> LOCAL FAIRNESS
>
> Let $G = (\Sigma, \sigma^I, \rightarrow)$ be a transition graph. A fairness constraint (α, β) is *local* if for all $s \xrightarrow{\alpha} t$, there is a state $u \in \Sigma$ such that $s \xrightarrow{\beta} u$. A *locally-fair* graph is a fair graph (G, F) such that (1) G is serial, and (2) F contains only local fairness constraints.

For a locally-fair constraint $f = (\alpha, \beta)$, whenever the action α is available, so is β. Consequently, in every α-fair ω-trajectorry, the choice to execute β is also available infinitely often, and thus, an f-unfair trajectory can be produced only by continuously ignoring the choice to execute β.

Figure 8.3 shows an interpreter for producing fair trajectories of a locally-fair graph. The input to the interpreter is a locally-fair graph $\mathcal{G}$, and a state s of $\mathcal{G}$. The algorithm uses two subroutines. The subroutine *Available*(α, s) takes an action α and a state s as input, and returns YES if there is a state u such that the pair (s, u) belongs to the action α. The subroutine *Execute*(α, s) takes an action α and a state s such that *Available*(α, s) holds, and returns a state u such that the pair (s, u) belongs to the action α.

The algorithm maintains a queue of the fairness constraints. The queue is initialized to contain all the fairness constraints of $\mathcal{G}$ in some arbitrary order. Let us say that a fairness constraint $f = (\alpha, \beta)$ is enabled in a state if the action

Algorithm 8.1 [Execution of locally fair graph]

Input: a locally fair graph $\mathcal{G} = (\Sigma, \sigma^I, \rightarrow, F)$ and a state s;
Output: a source-s fair trajectory $\underline{s}$ of $\mathcal{G}$.

Queue: a queue of fairness constraints

Initialization
$s_0 := s$;
Queue contains all the fairness constraints in F in some order.

Update rounds.
for $i := 0$ **to** ∞ **do**
Let *Queue* be $f_1 f_2 \ldots f_n$ with $f_k = (\alpha_k, \beta_k)$ for $1 \leq k \leq n$;
if $\mathit{Available}(\alpha_k, s_i)$ for some $1 \leq k \leq n$ **then**
$j := \mathit{min}\{k \mid \mathit{Available}(\alpha_k, s_i)\}$;
$s_{i+1} := \mathit{Execute}(\beta_j, s_i)$;
$\mathit{Queue} := f_1 \ldots f_{j-1} f_{j+1} \ldots f_n f_j$
else $s_{i+1} := \mathit{Execute}(\rightarrow, s_i)$
fi
od.

Figure 8.3: An interpreter for locally fair graphs

α is available. At every step, the algorithm checks if there is some fairness constraint that is enabled at the current state. If no such constraint exists, an arbitrary successor of the current state is chosen to be the next state (since G is serial, each state has at least one successor). If there are one or more enabled fairness constraints, then the algorithm chooses the constraint $f = (\alpha, \beta)$ such that f is enabled, and all the constraints that appear before f in the queue are disabled. Since f is local, availability of α implies availability of β, and the algorithm extends the trajectory by choosing some β-successor of the current state. Finally, the constraint f is moved from its current position in the queue to the end of the queue so that the other constraints get priorities in the subsequent rounds.

Proposition 8.11 [Execution of locally fair graph] *Given a locally-fair graph $\mathcal{G}$ and a state s, Algorithm 8.1 produces a source-s fair trajectory of $\mathcal{G}$.*

Proof. Let $\underline{s}$ be the ω-trajectory produced by the algorithm of Figure 8.3 in the limit. For every fairness constraint $f = (\alpha, \beta)$ of $\mathcal{G}$, and for every natural number i, let $\mathit{Unfair}(f, i)$ be true iff $(s_i, s_{i+1}) \notin \beta$; let $\mathit{StrongUnfair}(f, i)$ be true iff $\mathit{Unfair}(f, i)$ and $\mathit{Available}(\alpha, s_i)$; and let $\mathit{Rank}(f, i)$ be the position $1 \leq k \leq n$ of the constraint f in the queue at the beginning of round i. Observe that if $\mathit{Unfair}(f, i)$ then, in round i, the constraint f is not moved to the end of the queue, and thus, its rank cannot increase.

(1) For all $f \in F$ and $i \geq 0$, if $\mathit{Unfair}(f, i)$ then
$\mathit{Rank}(f, i + 1) \leq \mathit{Rank}(f, i)$.

If a fairness constraint f is enabled in round i, then it is executed unless some other constraint f' with $\mathit{Rank}(f', i) < \mathit{Rank}(f, i)$ is also enabled, in which case the constraint with least rank among the enabled constraints is executed, and moved to the end of the queue, which decreases the rank of f. This leads to:

(2) For all $f \in F$ and $i \geq 0$, if $\mathit{StrongUnfair}(f, i)$ then
$\mathit{Rank}(f, i + 1) < \mathit{Rank}(f, i)$.

We wish to establish that $\underline{s}$ is a fair trajectory of $\mathcal{G}$. Consider $f = (\alpha, \beta)$. We prove that if $\underline{s}$ is not β-fair then it is not α-fair. Suppose $\underline{s}$ is not β-fair. Then, there exists $i \geq 0$ such that for all $j \geq i$, $\mathit{Unfair}(f, j)$ holds. By (1), for all $j \geq i$, $\mathit{Rank}(f, j + 1) \leq \mathit{Rank}(f, j)$. Since for all $j \geq 0$, $\mathit{Rank}(f, j) \geq 1$, there can be only finitely many rounds j such that $\mathit{Rank}(f, j + 1) < \mathit{Rank}(f, j)$. By (2), there can be only finitely many rounds j such that $\mathit{StrongUnfair}(f, j)$. Hence, the action α is available only in finitely many rounds, and $\underline{s}$ is α-unfair. ■

The execution strategy to produce fair trajectories also implies the following proposition.

Proposition 8.12 [Machine closure of local fairness] *Every locally-fair graph is machine-closed.*

Exercise 8.17 {T3} [Execution of weak-locally-fair graphs] Let $\mathcal{G}$ be a locally-fair weak-fair graph. Show that, to produce a fair trajectory of $\mathcal{G}$, it suffices to maintain a modulo-$|F|$ counter, instead of the queue used by the interpreter of Figure 8.3. ■

8.4 Fair Modules

We associate a fair graph with every fair module by mapping the weak and strong fairness constraints of the module to the fairness constraints for the associated transition graph. Towards this goal, we associate with every update choice a of P, two actions of the graph G_P. The *availability action* of an update choice a contains a transition $s \to_P t$ if the choice a is enabled according to the values of the read variables in s and the awaited variables in t. The *execution action* of an update choice a contains a transition $s \to_P t$ if the values of the controlled variables in t are assigned by executing the choice a.

ACTIONS OF AN UPDATE CHOICE

Let P be a module, and a be an update choice of an atom U of P. The *availability action* $avail_a$ contains a transition $s \to_P t$ of P iff there is a guarded assignment γ in a such that

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

The *execution action* $exec_a$ contains a transition $s \to_P t$ of P iff there is a guarded assignment γ in a such that

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

Remark 8.13 [Actions of an update choice] For every update choice a of a module P, the action $exec_a$ is a subset of the action $avail_a$. ■

Example 8.10 [Actions of an update choice] Consider the update choice α of the module *AsyncCount*. The availability action $avail_\alpha$ contains all transitions of *AsyncCount*. The execution action $exec_\alpha$ contains the transition $s \to t$ if $count[t] = count[s] + 1$.

Consider the update choice α of the module *LossyBuffer*. The availability action $avail_\alpha$ contains the transition $s \to t$ iff $x[t] \neq x[s]$. The execution action $exec_\alpha$ contains the transition $s \to t$ iff $x[t] \neq x[s]$ and $y[t] \neq y[s]$. ■

Weak fairness for a choice a requires that the choice cannot be available forever without being executed, and strong fairness for a choice a requires that if the choice is available infinitely often then it is executed infinitely often.

Fairness constraints of an update choice

Let P be a module, and a be an update choice of P. The *weak-fairness constraint* f_a^W of a is the pair $(\rightarrow_P, exec_a \cup (\rightarrow_P \setminus avail_a))$. The *strong-fairness constraint* f_a^S of a is the pair $(avail_a, exec_a)$.

An ω-trajectory $\underline{s}$ of P is weakly fair to the update choice a precisely when it is f_a^W-fair: for infinitely many rounds $i \geq 0$, $(s_i, s_{i+1}) \in exec_a$ or $(s_i, s_{i+1}) \notin avail_a$. An ω-trajectory $\underline{s}$ of P is strongly fair to the update choice a precisely when it is f_a^S-fair: if for infinitely many rounds $i \geq 0$, $(s_i, s_{i+1}) \in avail_a$, then for infinitely many rounds $j \geq 0$, $(s_j, s_{j+1}) \in exec_a$.

Remark 8.14 [Strong fairness implies weak fairness] Let P be a module, a be an update choice of P, and $\underline{s}$ be an ω-trajectory of P. If $\underline{s}$ is f_a^S-fair then $\underline{s}$ is f_a^W-fair. The converse does not hold. ■

Remark 8.15 [Local fairness] Let P be a module, and a be an update choice of P. Both the fairness constraints f_a^S and f_a^W are local fairness constraints on the transition graph G_P. ■

The fair graph of a reactive module is obtained by adding all the fairness constraints corresponding to the declaration of weak and strong fair update choices.

Fair graph of a fair module

For a fair module $\mathcal{P} = (P, WeakF_P, StrongF_P)$, the fairness assumption $F_\mathcal{P}$ is the set

$$\{f_a^W \mid a \in WeakF_P\} \cup \{f_a^S \mid a \in StrongF_P\}$$

of fairness constraints of G_P. The fair module $\mathcal{P}$ defines the fair graph $\mathcal{G}_\mathcal{P} = (G_P, F_\mathcal{P})$.

A fair trajectory of a fair module $\mathcal{P}$ is a fair trajectory of the fair graph $\mathcal{G}_\mathcal{P}$.

Remark 8.16 [Fair trajectories of a fair module] The set of fair trajectories of a fair module P is a reactivity language. Furthermore, if the module employs only weak fairness, that is, for every atom U, the set $StrongF_U$ is empty, then the graph $\mathcal{G}_P$ is weak-fair, and the set of fair trajectories of a fair module is a response language. ■

8.4.1 Operations on Fair Modules

As in case of reactive modules, we combine fair modules using three operations —parallel composition, variable renaming, and variable hiding.

Parallel Composition

The composition operation combines two fair modules into a single fair module whose behavior captures the interaction between the two component modules. Again, composition is the key operation that allows modular descriptions of complex systems. Two fair modules can be composed only if their modules are compatible. Observe that for compatible reactive modules P and Q, an update choice of P is also an update choice of $P \| Q$. Consequently, to compose two compatible fair modules, we simply compose their reactive modules, and take union of the respective weak and strong fairness constraints.

FAIR MODULE COMPOSITION

If $\mathcal{P} = (P, \mathit{WeakF}_P, \mathit{StrongF}_P)$ and $\mathcal{Q} = (Q, \mathit{WeakF}_Q, \mathit{StrongF}_Q)$ are compatible fair modules, then the *parallel composition* $\mathcal{P} \| \mathcal{Q}$ is the fair module $(P \| Q, \mathit{WeakF}_P \cup \mathit{WeakF}_Q, \mathit{StrongF}_P \cup \mathit{StrongF}_Q)$.

The composition operator over fair modules has all the properties listed for the composition operator over modules in Chapter 1. For instance, the composition operator is commutative and associative.

Variable Renaming

As in modules, we can rename variables to avoid name-conflicts among private variables, and to create multiple copies. To apply a variable renaming to a fair module, we simply apply the renaming to each of its components.

RENAMING OF FAIR MODULE

Given a fair module $\mathcal{P} = (P, \mathit{WeakF}_P, \mathit{StrongF}_P)$, and a renaming ρ for the set X_P of module variables, the *renamed module* $\mathcal{P}[\rho]$ is the fair module with the module $P[\rho]$, the set $\{a[\rho] \mid a \in \mathit{WeakF}_P\}$ of weakly-fair update choices, and the set $\{a[\rho] \mid a \in \mathit{StrongF}_P\}$ of strongly-fair update choices.

Variable Hiding

The hiding of interface variables of a fair module allows abstractions at different levels of details. Hiding of an interface variable of fair modules changes only its variable declaration.

HIDING OF FAIR MODULE

Given a fair module $\mathcal{P} = (P, \mathit{WeakF}_P, \mathit{StrongF}_P)$, and a typed variable x, by *hiding* x in P we obtain the fair module (**hide** x **in** P, $\mathit{WeakF}_P, \mathit{StrongF}_P$), denoted **hide** x **in** $\mathcal{P}$.

8.4.2 Machine Closure and Receptiveness

We know that the transition graph of a module is serial. This means that a step-wise interpreter of a module never gets stuck, and can always extend a trajectory by adding one more step. The analog of seriality in the case of ω-trajectories is machine-closure. It says that every finite trajectory can be extended to a fair trajectory. In particular, there is a strategy to systematically resolve the choices so that the limit ω-trajectory is a fair one. Proposition 8.12 asserts that every locally-fair graph is machine-closed. Since the fairness constraints of an update choice are local, it follows that the fair graph of a fair module is locally-fair, and hence, machine-closed.

Proposition 8.13 [Machine closure of fair modules] *The fair graph $\mathcal{G}_\mathcal{P}$ of a fair module $\mathcal{P}$ is locally-fair, and hence, machine-closed.*

Exercise 8.18 {P2} [Fair module execution] Recall the interpreter for reactive modules from Chapter 2. Using the strategy outlined in Figure 8.3 for executing locally-fair graphs, write an interpreter for fair modules. The input to the interpreter should be a fair module $\mathcal{P}$, and it should produce a fair trajectory of $\mathcal{P}$, if we let it execute forever. ■

For reactive modules, the property of seriality is preserved under parallel composition. It means that every trajectory of a module can be extended by adding another state no matter how the environment updates the external variables. The same applies to machine-closure also. Every finite trajectory of a fair module can be extended to a fair trajectory, no matter how the environment updates the external variables in each round. Thus, the ability to produce a fair trajectory does not demand cooperation from the environment. This fact is captured by the next proposition.

Proposition 8.14 [Receptiveness] *Let $\mathcal{P}$ be a fair module, s be a state of $\mathcal{P}$, and Q be a module that is compatible with $\mathcal{P}$. There exists an ω-trajectory $\underline{t}$ of the composition $\mathcal{P}\|Q$ such that $X_\mathcal{P}[t_0] = s$ and $X_\mathcal{P}[\underline{t}]$ is a fair trajectory of $\mathcal{P}$.*

Proof. By definition of composition, there is a state t of $\mathcal{P}\|Q$ such that $X_\mathcal{P}[t] = s$. By Proposition 8.13, the fair module $\mathcal{P}\|Q$ is machine-closed, and hence, has a source-t fair trajectory. ■

The ability to produce a fair trajectory in the face of an adversarial environment is known as *receptiveness*. Consider a module P and a set $\mathcal{L}$ of ω-trajectories of P. Consider a two-player game in which the *protagonist* attempts to produce an ω-trajectory in $\mathcal{L}$, while the *adversary* tries to prevent this. Initially, the adversary chooses a trajectory $s_{0..m}$ of P. In each round, the adversary chooses the new external state, and then the protagonist extends the current trajectory by choosing the new controlled state. The choices of the protagonist are constrained by the transition relation of the module P. The protagonist wins the game if the resulting infinite ω-trajectory belongs to $\mathcal{L}$. The ω-language $\mathcal{L}$ is *receptive* for the module P if the protagonist has a winning strategy. Now, Proposition 8.14 can reformulated to assert that for every fair module $\mathcal{P} = (P, \mathit{WeakF}_P, \mathit{StrongF}_P)$ the set $\mathcal{L}_\mathcal{P}$ of fair trajectories is receptive for P.

8.5 Examples of Fair Modules

We revisit examples of modules, and add appropriate fairness constraints.

8.5.1 Shared-variables Protocols

Our canonical example of a shared-variable protocol is mutual exclusion. So far we have considered only the mutual exclusion requirement of such protocols, namely, that the two processes are not inside their critical sections simultaneously. When we consider ω-trajectories, the parallel composition of the two processes should also meet the *deadlock-freedom* requirement: if some process requests an entry to its critical section, then some process eventually enters its critical section. Consider a protocol that never allows any process to enter the critical section; that is, none of its guarded assignments assign the value *inC* to *pc*. Such a protocol satisfies the mutual exclusion requirement, but not the deadlock-freedom requirement, and hence, is not an acceptable solution to the problem. An even stronger requirement for the problem is the *starvation-freedom* property that: if a process requests an entry to its critical section, then the same process eventually enters its critical section. Thus, while deadlock-freedom admits solutions that always prefer one process over the other, starvation-freedom requires a fair resolution of the choice.

Both deadlock-freedom and starvation-freedom are liveness properties, and cannot be violated by trajectories. Requiring all ω-trajectories to satisfy such requirements is too strong, as it would rule out asynchronous solutions like *Pete*. For instance, the eventually periodic ω-trajectory $(o0o0)(r0o0)^\omega$ of *Pete* does not meet the deadlock-freedom requirement. Instead, we will add reasonable fairness assumptions, and require that all the fair trajectories satisfy the liveness properties.

Our formulation of the problem allows each process to remain outside the critical section for an arbitrary number of rounds, and to remain inside the critical section for an arbitrary number of rounds. Consequently, the update of *pc* from *outC* to *reqC* is nondeterministic. We do not add any fairness on the resolution of this choice: an ω-trajectory in which some process never requests an entry, is an acceptable scenario. The update of *pc* from *inC* to *outC* is also nondeterministic. To model the assumption that a process may not stay inside its critical section forever, we add weak fairness for the choice to exit the critical section.

Synchronous mutual exclusion

For the synchronous solution *SyncMutex* the update from *reqC* to *inC* is deterministic. Consequently, the only fairness assumption concerns the choice to leave the critical section. The resulting fair modules are shown in Figure 8.4. In Chapter 9, we will present an algorithm to verify that the module *FairSyncMutex* satisfies both the liveness requirements of deadlock-freedom and starvation-freedom.

Asynchronous mutual exclusion

The asynchronous solution *Pete* uses nondeterminism to model the assumption that the two processes execute at independent speeds. We would like to add fairness to rule out executions in which stuttering is always preferred over another available choice. For instance, if a choice to update *pc* from *reqC* to *inC* is available, then it should eventually be executed. The resulting fair modules are shown in Figure 8.5.

8.5.2 Circuits

Synchronous circuits

Recall the definitions of the logic gates *SyncAnd* and *SyncNot* from which all combinational circuits can be built. Both these modules are deterministic: in every round, once the values of the awaited external variables are determined, precisely one guarded assignment is available. Consequently, there is no need to add fairness constraints in the definitions of the logic gates. Our basic unit for memory cell is the module *SyncLatch* (Figure 1.17) with two boolean inputs *set* and *reset*. In every round, if the updated values of both the inputs *set* and *reset* are 1, then the private state of the latch is updated nondeterministically. In this case, both the guarded assignments are available, and the next state may be either 0 or 1. Nondeterminism, in this context, models the fact that the behavior of the latch is unknown, and we expect the latch to be used in an environment that never sets both *set* and *reset* simultaneously to 1. Consequently, we do not add any fairness constraint to *SyncLatch*.

Asynchronous circuits

Recall the modeling of asynchronous logic gates from Section 1.3.4. An asynchronous logic gate is unstable when its output is not according the desired function of the inputs. The gate can stay unstable for an arbitrary number of rounds, and then, it becomes stable by changing its output. Now, we can use fairness to ensure that the gate does not stay unstable forever. The asynchronous AND gate with fairness is shown in Figure 8.6. It requires weak fairness

```
module Q₁ is
  interface pc₁ : {outC, reqC, inC}
  external pc₂ : {outC, reqC, inC}
  atom
    controls pc₁
    reads pc₁, pc₂
    init
      [] true → pc₁' := outC
    update weaklyfair α₁
      [] pc₁ = outC                  →
      [] pc₁ = outC                  → pc₁' := reqC
      [] pc₁ = reqC ∧ pc₂ ≠ inC      → pc₁' := inC
      [] pc₁ = inC                   →
      [] pc₁ = inC                  α₁→ pc₁' := outC

module Q₂ is
  interface pc₂ : {outC, reqC, inC}
  external pc₁ : {outC, reqC, inC}
  atom
    controls pc₂
    reads pc₁, pc₂
    init
      [] true → pc₂' := outC
    update weaklyfair α₂
      [] pc₂ = outC                  →
      [] pc₂ = outC                  → pc₂' := reqC
      [] pc₂ = reqC ∧ pc₁ = outC     → pc₂' := inC
      [] pc₂ = inC                   →
      [] pc₂ = inC                  α₂→ pc₂' := outC
```

$$FairSyncMutex \;=\; \mathcal{Q}_1 \,\|\, \mathcal{Q}_2$$

Figure 8.4: Fair synchronous mutual exclusion

module $\mathcal{P}_1$ **is**
  interface $pc_1 : \{outC, reqC, inC\};\ x_1 : \mathbb{B}$
  external $pc_2 : \{outC, reqC, inC\};\ x_2 : \mathbb{B}$
  lazy atom
    controls pc_1, x_1
    reads pc_1, pc_2, x_1, x_2
    init
      $[\!]\ true \to pc_1' := outC$
    update weaklyfair α_1, β_1
      $[\!]\ pc_1 = outC \to pc_1' := reqC;\ x_1' := x_2$
      $[\!]\ pc_1 = reqC \wedge (pc_2 = outC \vee x_1 \neq x_2) \xrightarrow{\beta_1} pc_1' := inC$
      $[\!]\ pc_1 = inC \xrightarrow{\alpha_1} pc_1' := outC$

module $\mathcal{P}_2$ **is**
  interface $pc_2 : \{outC, reqC, inC\};\ x_2 : \mathbb{B}$
  external $pc_1 : \{outC, reqC, inC\};\ x_1 : \mathbb{B}$
  lazy atom
    controls pc_2, x_2
    reads pc_1, pc_2, x_1, x_2
    init
      $[\!]\ true \to pc_2' := outC$
    update weaklyfair α_2, β_2
      $[\!]\ pc_2 = outC \to pc_2' := reqC;\ x_2' := \neg x_1$
      $[\!]\ pc_2 = reqC \wedge (pc_1 = outC \vee x_1 = x_2) \xrightarrow{\beta_2} pc_2' := inC$
      $[\!]\ pc_2 = inC \xrightarrow{\alpha_2} pc_2' := outC$

$FairPete$ = **hide** x_1, x_2 **in** $\mathcal{P}_1 \,\|\, \mathcal{P}_2$

Figure 8.5: Fair asynchronous mutual exclusion

$$
\begin{array}{l}
\textbf{module } \mathit{FairAnd} \textbf{ is} \\
\quad \textbf{private } pc\colon \{\mathit{stable}, \mathit{unstable}, \mathit{hazard}\} \\
\quad \textbf{interface } \mathit{out}\colon \mathbb{B} \\
\quad \textbf{external } in_1, in_2\colon \mathbb{B} \\
\quad \textbf{lazy atom controls } \mathit{out} \textbf{ reads } pc, \mathit{out} \\
\quad\quad \textbf{update weaklyfair } \alpha \\
\quad\quad\quad [\!]\ pc = \mathit{unstable} \xrightarrow{\alpha} \mathit{out}' := \neg \mathit{out} \\
\quad\quad\quad [\!]\ pc = \mathit{hazard} \quad \rightarrow \mathit{out}' := \neg \mathit{out} \\
\quad \textbf{passive atom controls } pc \textbf{ reads } pc, \mathit{out} \textbf{ awaits } in_1', in_2', \mathit{out}' \\
\quad\quad \textbf{init} \\
\quad\quad\quad [\!]\ \textsc{And}(in_1', in_2', \mathit{out}') \quad \rightarrow pc' := \mathit{stable} \\
\quad\quad\quad [\!]\ \neg\textsc{And}(in_1', in_2', \mathit{out}') \rightarrow pc' := \mathit{unstable} \\
\quad\quad \textbf{update} \\
\quad\quad\quad [\!]\ pc = \mathit{stable} \wedge \neg\textsc{And}(in_1', in_2', \mathit{out}') \qquad\qquad\qquad \rightarrow pc' := \mathit{unstable} \\
\quad\quad\quad [\!]\ pc = \mathit{unstable} \wedge \textsc{And}(in_1', in_2', \mathit{out}') \wedge \mathit{out}' \neq \mathit{out} \rightarrow pc' := \mathit{stable} \\
\quad\quad\quad [\!]\ pc = \mathit{unstable} \wedge \textsc{And}(in_1', in_2', \mathit{out}) \wedge \mathit{out}' = \mathit{out} \ \rightarrow pc' := \mathit{hazard}
\end{array}
$$

Figure 8.6: Fair asynchronous AND gate

for the update choice to toggle the output when the state variable pc equals the unstable value.

8.5.3 Message-passing Protocols

Our canonical example of a distributed system of agents communicating via messages consists of a sender and a receiver.

Synchronous communication

Let us first consider the module *SyncSender* of Figure 1.25. The messages are produced by the asynchronous atom *Producer*, which requires an unknown number of rounds to produce a message. The atom *Producer* has two choices, one to produce a message and another to stutter, available in every round. We do not impose any fairness on the resolution of this choice, because a scenario in which no message is ever produced is acceptable. On the other hand, consider the atom *Consumer* of the receiver that consumes the messages. Again, the atom *Consumer* is asynchronous, and requires an unknown number of rounds to consume a message. We would like to use fairness to ensure that once the message is ready to be consumed ($pc = consume$), the event $done_C$ is eventually issued and the message is copied into msg_C. Weak fairness suffices for this purpose. Similarly, we use fairness to rule out a scenario in which the receiver is ready for reception of the message ($pc = receive$), but delays issuing the

$$
\begin{array}{l}
\textbf{module } \mathit{FairReceiver} \textbf{ is} \\
\quad \textbf{private } \mathit{pc}\colon \{\mathit{receive}, \mathit{consume}\};\ \mathit{msg}_R\colon \mathbb{M};\ \mathit{done}_C\colon \mathbb{E} \\
\quad \textbf{interface } \mathit{ready}\colon \mathbb{E};\ \mathit{msg}_C\colon \mathbb{M} \\
\quad \textbf{external } \mathit{transmit}\colon \mathbb{E};\ \mathit{msg}_S\colon \mathbb{M} \\
\quad \textbf{passive atom} \\
\qquad \textbf{controls } \mathit{pc}, \mathit{msg}_R \\
\qquad \textbf{reads } \mathit{pc}, \mathit{transmit}, \mathit{done}_C \\
\qquad \textbf{awaits } \mathit{transmit}', \mathit{msg}'_S, \mathit{done}'_C \\
\qquad \textbf{init} \\
\qquad\quad [\!] \ \mathit{true} \to \mathit{pc}' := \mathit{receive} \\
\qquad \textbf{update} \\
\qquad\quad [\!] \ \mathit{pc} = \mathit{receive} \wedge \mathit{transmit}? \to \mathit{msg}'_R := \mathit{msg}'_S;\ \mathit{pc}' := \mathit{consume} \\
\qquad\quad [\!] \ \mathit{pc} = \mathit{consume} \wedge \mathit{done}_C? \to \mathit{pc}' := \mathit{receive} \\
\quad \textbf{lazy atom controls } \mathit{ready} \\
\qquad \textbf{update weaklyfair } \alpha \\
\qquad\quad [\!] \ \mathit{pc} = \mathit{receive} \overset{\alpha}{\to} \mathit{ready}! \\
\quad \textbf{lazy atom } \mathit{Consumer} \\
\qquad \textbf{controls } \mathit{done}_C, \mathit{msg}_C \\
\qquad \textbf{reads } \mathit{pc}, \mathit{done}_C, \mathit{msg}_R \\
\qquad \textbf{update weaklyfair } \beta \\
\qquad\quad [\!] \ \mathit{pc} = \mathit{consume} \overset{\beta}{\to} \mathit{done}_C!;\ \mathit{msg}'_C := \mathit{msg}_R
\end{array}
$$

$$\mathit{FairSyncMsg} \ = \ \textbf{hide } \mathit{ready}, \mathit{transmit}, \mathit{msg}_S \textbf{ in } \mathit{SyncSender} \parallel \mathit{FairReceiver}$$

Figure 8.7: Fair synchronous message passing

event *ready* forever. The receiver with these fairness assumptions is shown in Figure 8.7. The fair module *FairSyncMsg* is obtained by composing the fair receiver with *SyncSender* and hiding the variables used for communication.

Exercise 8.19 {P2} [Fair asynchronous sender] What are the appropriate fairness constraints for the module *AsyncSender* of Figure 1.30. Define the fair version of the modules *AsyncSender* and *AsyncMsg*. ■

Exercise 8.20 {P3} [Fair timed message passing] The protocol for timed message passing (Figure 1.32) refers to the external clock *AsyncClock* for measuring time. Define a fair version of the clock *AsyncClock* so that every ω-trajectory contains infinitely many *tick* events. Define a fair version of the module *TimedMsg* by adding appropriate fairness assumptions to *TimedSender* and *TimedReceiver*. ■

$$
\begin{array}{l}
\textbf{module } \mathit{FairStick} \textbf{ is}\\
\quad \textbf{private } \mathit{pc}: \{\mathit{free}, \mathit{left}, \mathit{right}\}\\
\quad \textbf{interface } \mathit{grant}_L, \mathit{grant}_R : \mathbb{E}\\
\quad \textbf{external } \mathit{req}_L, \mathit{release}_L, \mathit{req}_R, \mathit{release}_R : \mathbb{E}\\
\quad \textbf{passive atom}\\
\quad\quad \textbf{controls } \mathit{pc}, \mathit{grant}_L, \mathit{grant}_R\\
\quad\quad \textbf{reads } \mathit{pc}, \mathit{req}_L, \mathit{grant}_L, \mathit{release}_L, \mathit{req}_R, \mathit{grant}_R, \mathit{release}_R\\
\quad\quad \textbf{awaits } \mathit{req}'_L, \mathit{release}'_L, \mathit{req}'_R, \mathit{release}'_R\\
\quad\quad \textbf{init}\\
\quad\quad\quad [\!] \; \mathit{true} \to \mathit{pc}' := \mathit{free}\\
\quad\quad \textbf{update stronglyfair } \alpha_L, \alpha_R\\
\quad\quad\quad [\!] \; \mathit{pc} = \mathit{free} \wedge \mathit{req}_L? \quad \xrightarrow{\alpha_L} \mathit{grant}_L!;\ \mathit{pc}' := \mathit{left}\\
\quad\quad\quad [\!] \; \mathit{pc} = \mathit{free} \wedge \mathit{req}_R? \quad \xrightarrow{\alpha_R} \mathit{grant}_R!;\ \mathit{pc}' := \mathit{right}\\
\quad\quad\quad [\!] \; \mathit{pc} = \mathit{left} \wedge \mathit{release}_L? \quad \to \mathit{pc}' := \mathit{free}\\
\quad\quad\quad [\!] \; \mathit{pc} = \mathit{right} \wedge \mathit{release}_R? \to \mathit{pc}' := \mathit{free}
\end{array}
$$

Figure 8.8: A fair chopstick for the dining philosophers

Dining philosophers

Recall the problem of dining philosophers from Exercise 1.13, and consider the module *Stick*. When both the philosophers on the two sides of a chopstick request the chopstick simultaneously, it is granted to either one of them nondeterministically. To model the assumption that if some philosopher requests a particular chopstick repeatedly, it is eventually granted, we can use strong fairness. Such a fair chopstick is shown in Figure 8.8. Observe that even if the philosopher reissues the request in every round until the chopstick is granted, the choice to grant the request is not always available (due to the conjunct $pc = \mathit{free}$ in the guard). Consequently, the weaker assumption that some philosopher requests a particular chopstick *continuously*, it is eventually granted, is not captured by requiring weak fairness for the choices α_L and α_R.

Exercise 8.21 {P3} [Fair dining philosophers] Consider your definitions of the modules that describe philosophers, together with guards, in Exercise 1.13. Replace the chopstick by the fair chopstick of Figure 8.8. Add fairness assumptions to each philosopher to ensure that once the philosopher has both the chopsticks, (s)he will eventually release both of them. Define the fair module *FairDine4* consisting of four copies of fair chopsticks and fair philosophers. Does your solution satisfy starvation-freedom, that is, is there fair trajectory of *FairDine4* in which some philosopher, after some round, waits forever? ■