FAIR SUPERVISORY CONTROL OF DISCRETE-EVENT SYSTEMS

BY

PEYMAN GOHARI-MOGHADAM

A THESIS SUBMITTED IN CONFORMITY WITH THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY,
GRADUATE DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING,
IN THE UNIVERSITY OF TORONTO.

Fair Supervisory Control of Discrete-Event Systems

Doctor of Philosophy Thesis
Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto

by Peyman Gohari-Moghadam
September 2002

Abstract

Fairness in supervisory control is defined and studied for discrete-event systems when they are represented by finite deterministic automata, or correspondingly by regular languages. The notions of weak and strong fairness are adapted to address the issue of fairness in open- and closed-loop systems. It is argued that since all controllers in practice are finite-state, when synthesizing a controller one need only consider the bounded fair designs. When a bound is specified, the supremal fair restriction of a system exists in both cases and is computed in time polynomial in the state size of the system. Priority relations are reintroduced as a convenient way to describe precedence among events; however, it is shown that under priorities a supremal fair design may not exist in general. Lastly, queues are introduced as an efficient way to implement fairness hierarchically.

Contents

List of Figures

Chapter 1

Introduction

1.1 Elements of supervisory control

1.1.1 Discrete-event systems

A discrete-event system [Won02], [Cas93] is a dynamic system where the state space is represented by a discrete set, and state transition is driven by instantaneous occurrence of events at usually discrete points in time. An event may describe a low-level action (e.g. a micro switch being turned on), or some high-level observation about a physical entity (e.g. water level in a reservoir reaching a certain mark). The modeler decides what level of abstraction is appropriate for the problem at hand. Other examples of events are pressing a button, failure of a device, and sending a message through a communication channel.

Discrete-event systems encompass a wide variety of physical systems. These include [Won02] manufacturing systems, traffic systems, communication protocols, logistic systems, database management systems, and data communication networks.

In the style of Ramadge and Wonham [RW87], we model a discrete-event system as the generator of a formal language over the alphabet of events. At a more conceptual level where the emphasis is on the external behavior, we simply represent a system by a formal language. Thus, formal languages offer a more abstract view of discrete-event systems. However it should be pointed out that any theoretical result may be interpreted only in light of physical implementation.

1.1.2 Supervisory control

Control is introduced into our setup by designating a subset of events as *controllable*. An event is controllable if it can be disabled by some external agent. For example, the event of pressing a button is controllable.

A *supervisor* is an external agent in charge of controlling the system. It observes the sequence of events generated by the system as they unfold, and it can restrict the future behavior at any point by disabling a subset of controllable events. Thus, a supervisor always restricts the behavior, and the closed-loop behavior is a subset of the open-loop behavior as a result.

In supervisory control, the requirement on the system's behavior is often specified by a formal language (or its generator). When the plant and specification are represented by languages L and E, respectively, we say the system—or the plant—satisfies the specification iff $L \subseteq E$, that is, iff every event sequence generated by the plant is acceptable by the specification. When this is not the case, the objective of supervisory control is to design a supervisor that restricts the behavior of the plant in such a way that the supervised system, whose behavior we denote by K, satisfies the specification, i.e. $K \subseteq E$.

An important issue in supervisory control is to identify those sublanguages K of a given language E that can be the behavior of the supervised plant under some supervisory control. We call a language K controllable if a supervisor can be found such that the behavior of the supervised system is equal to K. Ramadge and Wonham

[RW87] show that the controllable sublanguages of a given language form an upper semi-lattice, and therefore a supremal element exists. A supervisor is called optimal if it restricts the behavior of the plant to the supremal controllable sublanguage. The supremal controllable sublanguage is computed in time polynomial in the state sizes of plant and specification when they are modeled by regular languages and the corresponding finite automata.

The reader is referred to [Won02] for a comprehensive study of supervisory control.

1.2 Requirement specification

1.2.1 Safety and liveness

Properties of a discrete-event system can be decomposed into two conceptually different categories. Informally, a *safety* property states that "nothing bad ever happens", while a *liveness* property stipulates that "something good will (eventually) happen". A property may be viewed as a set of strings, which can be represented by the marked language of an automaton. As an example, consider a property specified by the automaton of Figure 1.1. A string satisfies the property if and only if (i) it starts with

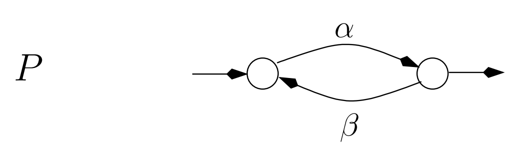


Figure 1.1: Property P.

α, and α and β alternate, and (ii) it terminates with α. The first property is safety. Here "bad things" in the definition are a string where α and β do not alternate, or a string that starts with β. The automaton specifies that bad things described as such never happens. In addition, it specifies a liveness property, namely (ii). It specifies that "good things"—the occurrence of α—will eventually happen in a system that satisfies the property. It is shown in [AS87] that any property can be decomposed into safety and liveness components. In our example, we may specify the components just described by the two automata of Figure 1.2. Observe that in an automaton represen-

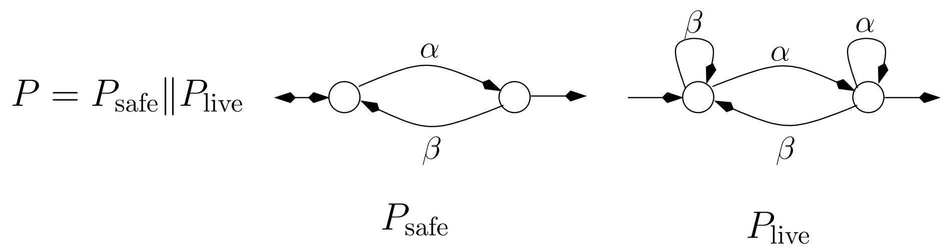


Figure 1.2: Decomposition of P into P_{safe} and P_{live}.

tation of a safety property—"safety automaton" for short—all states are marked, and strings are excluded from the language by disabling a subset of controllable events at different states. On the other hand, at every state of a liveness automaton all events are enabled, and behavior is restricted by marking a subset of states as accepting.

To see yet another distinction between safety and liveness properties, consider the infinite string $s = \alpha\beta\alpha^\omega$. Observe that since the finite prefix $\alpha\beta\alpha^2$ of s is not accepted by P_{safe}, neither is s. The reason is that if a string does not satisfy a safety property, none of its continuations will. Now consider the infinite string $s = \alpha\beta\beta^\omega$. It can be seen that while s is not accepted by P_{live}, all its finite prefixes can be completed to a string in P_{live}. In short, one may conclude that a counterexample to a safety property is finite, whereas a counterexample to a liveness property is infinite.

1.2.2 Liveness in supervisory control: Nonblocking

In supervisory control of discrete-event systems liveness is taken into account by introducing the property of nonblocking. A discrete-event system is called coreachable iff there is a path from any state to a marker state. Ramadge and Wonham call a supervisor nonblocking if the supervised system is coreachable.

For example, consider a supervisor that disables the β event at the right-hand side state of the system shown in Figure 1.3.

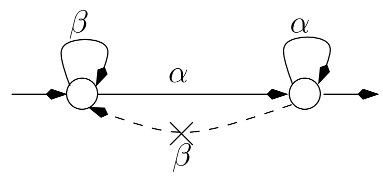


Figure 1.3: A system under control.

The supervisor defined above is nonblocking as the closed-loop system is coreachable. It implements a safety property requiring that an acceptable string must follow the pattern $\beta^m\alpha^n$, for some $m, n \geq 0$. In addition, it implements a liveness property requiring that an acceptable string end with α. Overall, the supervised system can perform any number (possibly zero) of β's, followed by any non-zero number of α's.

Although the closed-loop system is coreachable, a critical analysis of the system reveals what could be a flaw in our design to meet the above objectives. Suppose the system were described by a stochastic model such as a Markov chain [VN92]. Then one could derive probabilistic information about the system, including the mean time to reach the marker state. By a simple mathematical argument it can be seen that the mean time to reach the marker state grows without bound as the probability of taking β at the left-hand side state approaches 1 (and therefore the probability of taking α at the same state approaches 0). Intuitively, because of the dominant nature of β, the system may perform any number of β's without reaching the right-hand side state through the α event. One might conclude that from a practical point of view the system is not coreachable.

Thus, coreachability may not be an adequate measure for liveness of discrete-event systems—especially in the context of distributed systems—when the probabilities are lopsided, or even worse, when no quantitative probabilistic information about the system is available. In this work we introduce a more general class of properties, called *fairness*, which encompasses the already familiar notion of coreachability.

1.2.3 Fairness

The idea of fairness has long been studied in computer science. In its most basic form, fairness means that [Fra86] if a system has to choose repeatedly from a set of alternatives (read "events"), no alternative will be postponed indefinitely; in other words, each alternative will be chosen eventually. Assigning an exact meaning to "repeatedly" in the above description gives rise to the definitions of weak and strong fairness.

Weak fairness requires that the alternative be taken eventually if it is continuously enabled in every choice. Thus, when an alternative is offered only at odd-indexed choices, weak fairness does not require that the alternative be chosen at all. Obviously, we have to strengthen the condition if we are not satisfied with the degree of fairness that the definition provides. This is accomplished by relaxing the enabledness condition: a system is *strongly fair* whenever an alternative is taken eventually if it is enabled infinitely often (i.e. cannot be enabled infinitely often and never taken). To illustrate the contrast between weak and strong fairness, consider a question and answer session at the end of a lecture. Provided that time permits answering all questions, in a strongly fair system a question by an audience member is guaranteed to be answered if he frequently raises and drops his hand. However, in a weakly fair system there is no such guarantee unless a person holds his hand up steadily until his question is answered.

There are several ambiguities surrounding the above definitions, which mainly result from the fact that as often is the case with computer science terminology, no distinction is made between physical plant and controller. For example, when an alternative is said to be enabled, it is not clear if it is enabled in the plant, in the controller, or in both. Therefore, our formal study of the subject begins by giving a concise definition of these terms.

We call an event *eligible* at a state if it is physically possible to be taken at that state of the plant. An eligible event may be *enabled* or *disabled* by a supervisor. From the definition of controllable events it follows that an event is disabled only if it is controllable.

In supervisory control, we consider weak fairness a property of the supervised system; that means the plant under no control is weakly fair by default. A supervised system is weakly fair if along all paths, no event is continuously disabled. That is, if an event is continuously disabled for a number of steps, it must eventually be enabled, or otherwise become ineligible to occur in the plant. This definition appears to be in sharp contrast with the previous one, and the reason, as mentioned earlier, is that previously the distinction between plant and controller was blurred. However we will see that there are enough similarities to justify the same name for the corresponding property in supervisory control. Recall that in [Fra86] weak fairness demands eventual occurrence of an event if it is continuously enabled. We interpret "enabled" as eligible in the *plant*. Suppose an event is continuously eligible but is disabled by the supervisor. While [Fra86] requires its "eventual occurrence", our definition stipulates its eventual enabledness by the *supervisor* if it continues to be eligible. As one may note, our weak fairness property depends on the status of events in *both* plant and controller.

A weakly fair supervisor by definition does not postpone an event forever along *any* infinite run of the system. This may be too restrictive when some infinite runs are so unlikely that we do not mind if along any of those an event is continuously neglected. To accommodate cases like this, we borrow the notion of priorities from [MK99]. Priorities are essentially an abstract measure for the probability of events. Whenever two alternative events compete for occurrence at a state, we may give one a higher priority over the other to emphasize that it is improbable that the low priority event is continuously favored by the system—that is, the combination of the plant and the supervisor—over the other. This implies that when an infinite string of a supervised system consists of low priority events (at the expense of one or more high priority events), that string is improbable, and the system is considered weakly fair even if some events are treated unfairly all along.

When a supervised system is weakly fair, all events (perhaps from different components) are offered a fair *chance* to be taken. The supervisor, however, does not guarantee the eventual occurrence of any particular event. This could lead to a problem if one of the components maliciously steals from the others the chance of being taken on every turn. Against such malevolence, we need an additional guarantee that each event is taken after being frequently offered. We say a system (open- or closed-loop) is strongly fair if an event is taken infinitely often whenever it is eligible infinitely often along a path. It is well to note that whether or not the system is under control is not an issue here. In fact, it could be that the system is already under control by a supervisor that always enables the event whenever it is eligible in the plant, yet the supervisor fails to guarantee its occurrence by enabling other alternatives too. What strong fairness demands is a second supervisor that limits the alternatives for the next move to a single one, namely the event which we want to be treated fairly. As evident from the definition, strong fairness is a property of a system, with no additional requirement placed on the control structure. It is for this reason that our definition is identical to that in [Fra86].

Contexts for fairness

Concern about fairness arises when several groups of processes vie for exclusive access to a shared resource, especially in one of the following three contexts.

1. *Congestion*. When a shared resource is heavily in use by a group of processes, other groups competing for access to the same resource may never get a chance to do so, causing a condition commonly referred to as *starvation*.

2. *Collaboration* and *Malevolence*. Moreover, the competing processes are denied the chance of ever accessing a shared resource if the processes in one group collaborate to continuously keep the resources usage within the group. The processes are said to be *malevolent* if the continuous possession of the resource is intentional.

3. *Imbalance*. When there is imbalance in competitiveness among different groups, it is possible for a shared resource to be in constant usage by groups of more aggressive processes. The other processes, despite having a chance to access the resource, may starve due to their lack of competitiveness.

It is worthwhile to point out that in the first two contexts a group of processes may never get a chance to access a shared resource, which is a violation of the weak fairness condition. In comparison, the situation is less dramatic in the third context, where the system may fail to be strongly fair.

Choice of model

The existing frameworks for modelling discrete-event systems cannot be directly used to represent any of the circumstances described above, namely congestion, collaboration and malevolence, and imbalance. However, it is possible to detect any potential violation of fairness from the existing models. As far as detecting such a potential is concerned, a probabilistic model does not provide any additional insight not derivable from its deterministic counterpart. Moreover, working with a deterministic model will enable us to obtain worst-case guarantees on the eventual occurrence of a desired behavior. (See the discussion of the next section on how it is sometimes more convenient to study the safety version of a liveness property.)

1.2.4 From liveness to safety: Bounded fairness

Consider an unfair multi-agent system in which one of the agents will never participate[1] in a run of the system. We would like to make the system fair by designing a controller that ensures the eventual participation of the agent. In practice since any such controller has a finite number of states it must schedule the agent within a finite number of steps. Let k denote the number of steps within which the agent is scheduled. The system under control satisfies the original specification that the agent always participates eventually by meeting a stronger requirement, namely, that the agent always appears within k steps.

In general, since our controllers are all finite-state, they implement an eventuality requirement within a certain number of steps, or within a *bound*. In this study we investigate the bounded version of the fairness problem, which we refer to as *bounded fairness*. Therefore, we exclude from our study infinite-state controllers. For instance, when a and b represent two agents, a controller might attempt to schedule them always to appear eventually as follows:

$$abbaaabbbbaaaaabbbbbb\ldots$$

[1] Or be offered a chance to participate, depending on whether a strong or a weak version of fairness is meant, respectively.

One can see there is no bound on the eventual occurrence of either a or b, implying that such a design needs infinite memory for implementation and is therefore excluded from our study.

The bounded version of the fairness problem is a stronger requirement than unbounded fairness. If an event appears within a bound in a sequence, it can also be said that it appears in the sequence eventually. Thus, in what we propose, in place of requiring that something good (the agent participating) will eventually happen, we demand a stronger condition requiring that something bad (the agent not participating within k steps) never happens. Therefore placing a bound on fairness has the effect of transforming a liveness property into a safety one.

We end this introduction by noting that in bounded fairness we infer the properties of infinite strings from the properties of a sliding window of length $k+1$ over the string. If an event appears in every sliding window of length $k+1$ over an infinite string, it follows that it always appears within k steps, and therefore it always appears eventually.

1.2.5 Liveness in supervisory control: Fairness

This thesis is dedicated to the study of liveness within the framework of supervisory control, where at any point during the system's evolution a supervisor may restrict possible future behaviors by *disabling* a subset of controllable events. We introduce two new properties for measuring liveness in supervisory control, which we call weak and strong fairness, after the terms used in computer science to describe similar properties.

There is a subtlety in our approach which differentiates it from other works in supervisory control. Suppose we would like to establish the existence of a largest controllable *and* fair restriction of a system. It can be easily shown that such an optimal restriction exists if and only if both a largest controllable restriction, *and* a largest fair restriction, exist. In fact, the optimal controllable and fair restriction can be computed by alternate applications of the procedures for computing the optimal controllable, and for computing the optimal fair restrictions. Throughout this thesis we keep the issues of controllability and fairness separate. Since controllability is discussed extensively in [RW87], we focus almost exclusively on the fairness issue.

1.3 Related work

In [MP92] Manna and Pnueli define the semantics of a concurrent program by interleaving the computations of its parallel processes. Fairness is introduced to limit the interleavings to those that are "fair", i.e. those that represent an actual behavior of the program. For instance, no transition of a process is consistently neglected when it is enabled in the actual program.

Analogous to our work, they define weak and strong fairness—called "justice" and "compassion", respectively—depending on whether a transition is enabled continuously or infinitely often. The open-loop system is a program, while the supervisor is called a *scheduler*. In distinction to our work, they are interested mainly in specifying and *verifying* that a given scheduler is fair, whereas our objective is to *synthesize* a scheduler that is fair. This is a major distinction between research in supervisory control and in formal verification. While the verification community is interested in verifying whether a property (here "fairness") is met by a system, we are interested in designing a supervisor that implements the property. It should not then come as a surprise that our model of a system (namely, finite deterministic automaton) is much more primitive than the high-level models to which system verification methods are commonly applied. The synthesis problem is inherently harder than verification, and it becomes increasingly difficult to treat a synthesis problem using sophisticated models.

In support of working with a bounded version of the fairness problem, [MP92] notes that justice is not associated with any quantitative measure of evenhandedness, and any practical scheduler has a bound on how long a transition can be enabled before activating it. Manna and Pnueli also comment that with no bound fairness is an abstract concept that cannot be physically implemented in its full generality. Yet for the sake of robustness of verification[2], it is common practice to work with the unbounded version of the fairness problem: a property encloses all bounded-fair behaviors if it encloses all unbounded-fair behaviors.

Fairness is a property of infinite behavior. We assume that no matter what choices a system makes it always runs *ad infinitum*. For finite behavior fairness becomes irrelevant except when a supervisor brings a system to a complete halt by not making any choices. Such a situation, called *deadlock*, is clearly undesirable and we regard the resulting finite behavior as unfair. Our decision to declare such finite behavior as unfair is in agreement with Lynch [Lyn96], where fairness in a weak sense is defined for I/O automata.

Jayasimha and Dershowitz formalize bounded fairness in [JD86] by introducing a new binary operator into temporal logic. They call the new logic kTL. While they show that it has the same expressive power as linear temporal logic with the *until* operator, they argue that a bounded fair property can be expressed more succinctly in kTL. The property of k-boundedness suffers from the setback that for any fixed k, it is easy to construct a scheduler that activates a transition $k+1$ events after its enablement, and therefore is not k-fair. A more abstract model of fairness is obtained when one lets the natural number k vary. Alur and Henzinger call a behavior *finitary fair* if it is k-bounded fair for some natural number k [AH98]. Finitary fairness is an abstraction of all "realistic" fair behaviors, i.e. those that are generated by a finite

[2]That is, to use the same procedure to verify all properties.

scheduler of unknown size. It is stated in [AH98] that the verification rules for finitary fairness are simpler than those for absolute fairness.

In [NB98] Negulescu and Brzozowski define finitary fairness on two finitary descriptions of discrete-event systems: finite trace structures and finite automata. Their work has a certain control-theoretic appeal in that the distinction between the closed-loop system (called "implementation") and specification is explicit, and they define liveness of an implementation *relative* to the specification. They call an implementation live if it is *traplock* free, i.e. an output transition in a loop is taken whenever it is enabled by the specification. Finally the authors present an algorithm for verifying their liveness condition.

Fairness as applied to supervisory control of discrete-event systems is related to the more familiar notion of stability, which is explored by Brave and Heymann in [BH89]. A set of states that is invariant under the transition relation is called an *attractor*. A system is stable if from any state there is a path leading to an attractor, and is stabilizable if the closed-loop system is rendered stable by means of some supervisory control. The connection between stability and weak fairness becomes more evident upon declaring an attractor as the set of desired states. However, one should be aware of their different nature: while in a stable system one demands the existence of *some* path to a good behavior, the weak fairness condition requires that *every* path eventually end up in a good behavior.

Thistle and Wonham study the supervisory control of infinite behavior by extending basic results of supervisory control theory to the setting of ω-languages [TW94]. The new setting is expressive enough to specify both safety and liveness properties. A new notion of controllability, called ω-controllability, is defined and a supremal ω-controllable sublanguage is computed in polynomial time for the case where plant and specification are modeled by deterministic Büchi and Rabin automata, respectively. In contrast to [TW94], we aim at keeping our discussion more elementary by inferring the properties of infinite strings from those of a sliding window of a fixed length over strings.

1.4 Structure of the thesis

The rest of the thesis is organized as follows. Chapter 2 motivates the discussion. We study two instances of the same problem, one of which is weakly fair while the other is not. We stress that when a system is not weakly fair, the supervisor may inadvertently disable a component continuously, causing it to starve. Thus weak fairness is equivalent to starvation-freedom. In a fairly long Chapter 3, we introduce weak fairness in supervisory control theory. In particular, we prove that it is a requirement more stringent than the traditional nonblocking requirement. We redefine priorities in discrete-event systems, and demonstrate how they can be utilized to model physical systems. We show that while the class of unboundedly fair sublanguages of a given language is not closed under arbitrary union, the boundedly fair class is, and therefore an optimal element exists. We compute the optimal boundedly fair controller in polynomial time when the (closed-loop) system is regular.

In Chapter 4 we illustrate our findings on a simple example. We design an optimal controller for a closed-loop system that initially is not weakly fair. We see two traditional fixes to the unfairness problem, namely, implementing a turn variable and a first-in-first-out queue, and we show that our design is less restrictive than either. We have devoted all of Chapter 5 to the Dining Philosophers problem, which is famous because a naive implementation of mutual exclusion can lead to starvation. The situation is improved by designing a controller that implements weak fairness. Along the way we generalize fairness to more than one event, and define the scope of behavior which specifies which events are competing against each other when accessing a shared resource.

Strong fairness is introduced in Chapter 6. Many terms and concepts are similar to those in Chapter 3 and we refrain from repeating them. As we did for weak fairness, we construct an automaton for generating the optimal boundedly fair sublanguage of a given language. It turns out that implementing a turn variable, or alternatively a first-in-first-out queue, both yield strongly fair solutions, which as expected are more restrictive than our optimal controller.

Chapter 7 studies queues in detail. Although we showed in the previous chapter that they are not optimal, the conventional wisdom in using queues for implementing strong fairness finally becomes manifest as they lend themselves neatly to hierarchical design. We show this by a formal study of infinite strings generated by a schedule based on the degree to which successive occurrences of different events are separated. Although this separation in queues is more than what is needed to implement strong fairness, it allows just enough space for designing sub-queues within each "team" of agents, thereby a hierarchical queue for scheduling service among all agents. Finally, Chapter 8 concludes the thesis by looking back at what was achieved and pointing out future directions for research.

Chapter 2

Motivation

2.1 Introduction

When several processes compete for exclusive access to a shared resource there is reason to ask whether or not these processes are granted access to the resource in a "fair" way. The simplest situation arises with two teams, each consisting of several processes. For simplicity we call one team "cats" and the other "mice", and the resource that they compete for is access to a common room that can only be shared by them in a mutually exclusive way (cf. [RW87]).

The formal description of the system is as follows. Two different species—cats and mice—are locked in their separate cages. When a cage is opened its inhabitants are free to roam round a common room. A controller is needed to ensure that mutual exclusion is respected, that is, no species of different type occupy the room at the same time.

We study two different versions of the problem: C1M1 and C2M1. As the names suggest, in both we have one mouse, while the number of cats in C1M1 and C2M1 is one and two, respectively. The addition of an extra cat to C1M1 introduces a type of unfair behavior that was not present before. In the process we define *priorities*, a

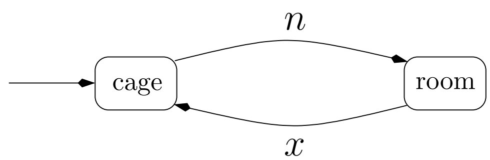


Figure 2.1: Basic model.

notion that later will be dropped due to the difficulties that it introduces to computing a design.

We begin our analysis by a formal statement of the problem. Each animal is modeled using a two-state automaton. Initially, the animal is in its cage. When the cage's door is open, it may enter the room, which is represented by an 'n' (for "enter") transition in the model. The animal may return to its cage at its leisure by taking an 'x' (for "exit") transition.

An event is subscripted with 'c' and 'm' for "cat" and "mouse", respectively, and it may additionally be subscripted with an integer to distinguish different members within the same species.

2.2 C1M1

The open- and closed-loop systems are shown in Figure 2.2. To achieve exclusive access to the room, the controller ensures that when the cat is in the room the mouse is locked in its cage, and vice versa.

Throughout our study, we assume that the system is "live", that is, it may not forever remain idle in a state when a transition out of that state is possible. The above assumption enables us to measure fairness by the number of transitions needed to perform a task. If each transition takes at most t seconds to complete, then instead of requiring the completion of a task in T seconds we may specify $\lfloor \frac{T}{t} \rfloor$ as the maximum number of allowable transitions in the model before the task is complete.

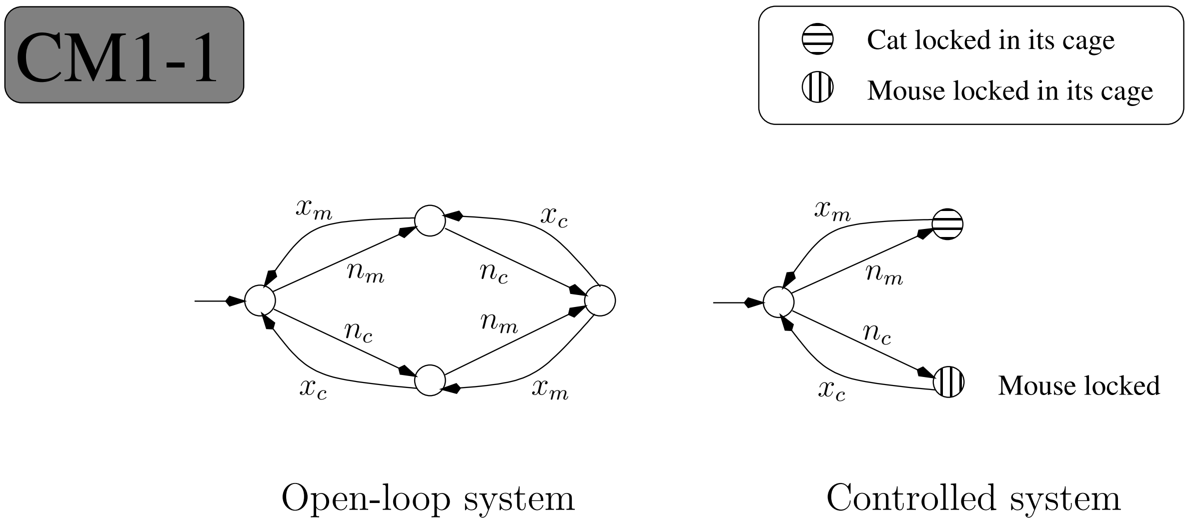


Figure 2.2: Open- and closed-loop systems.

In a broad sense, fairness demands the liveness of components when the system itself is live; in other words, fairness seeks the participation of all components while the system is in progress. Thus, in any fair run of the Cat and Mouse, it is not the case that beyond some point either the cat or the mouse uses the room exclusively.

Consider an unfair run where the room is in exclusive use by the cat. An immediate question is whether the mouse was denied access to the room by the controller, or it simply elected not to take part. This leads to the distinction between *strong* and *weak* fairness. In a strongly fair design, an unfair run does not exist at all. We can see that C1M1 is not strongly fair, as it does not prevent the cat from moving back and forth between the room and its cage while the mouse is sitting idle; in other words, it accepts the unfair infinite sequence $(n_c x_c)^\omega$.

Note that in the above scenario it was not the case that the mouse's cage was locked all the time. As a matter of fact, the cage was open after every second move made by the cat. Yet for reasons not evident from the model the mouse refused to participate—either willingly, or maybe because it was preempted by the cat in each turn. Regardless, we consider the design weakly fair, as even in unfair runs the controller frequently gave both species the chance to enter the room.

2.3 C2M1

The open- and closed-loop systems, together with the controller, are shown in Figure 2.3. To avoid cluttering the picture only a sketch of the open-loop system is depicted. Again, the controller achieves exclusive access to the room by ensuring that the mouse is locked out when the cats are in, and vice versa.

Consider the infinite run $n_{c_1}(n_{c_2}x_{c_2})^\omega$. This run is unfair since after n_{c_1} only the second cat participates. However, while the first cat elects perhaps willingly not to take part, the mouse is *denied* access to the room by the controller; in other words, the event n_m is disabled in all nonempty prefixes of $n_{c_1}(n_{c_2}x_{c_2})^\omega$. For this reason, the system is not regarded as weakly fair. (And therefore is not strongly fair either—see the diagram of Figure 2.4.)

The reader might argue that the above run is not likely to be generated, as the chances of the first cat refusing to participate on every turn are slim to nonexistent. We counter by presenting the run $n_{c_1}(n_{c_2}x_{c_1}n_{c_1}x_{c_2})^\omega$ in which both cats participate, yet the mouse is continuously discriminated against. (Note that n_m is disabled in all nonempty prefixes of the above infinite string.)

Although both cats participate alternately and overlappingly in the latest counterexample, the reader may still find the infinite run an unlikely behavior of the system. Whenever a selection is to be made between $\{x_{c_1}, n_{c_2}\}$ (note that these events are both offered only at the top state of the closed-loop system), n_{c_2} is always favored. A similar observation can be made about the selection between $\{x_{c_2}, n_{c_1}\}$ at the bottom state. We can justify this seemingly unfair selection by introducing the notion of *priorities*. We postulate that an event σ' can overtake σ when they are both

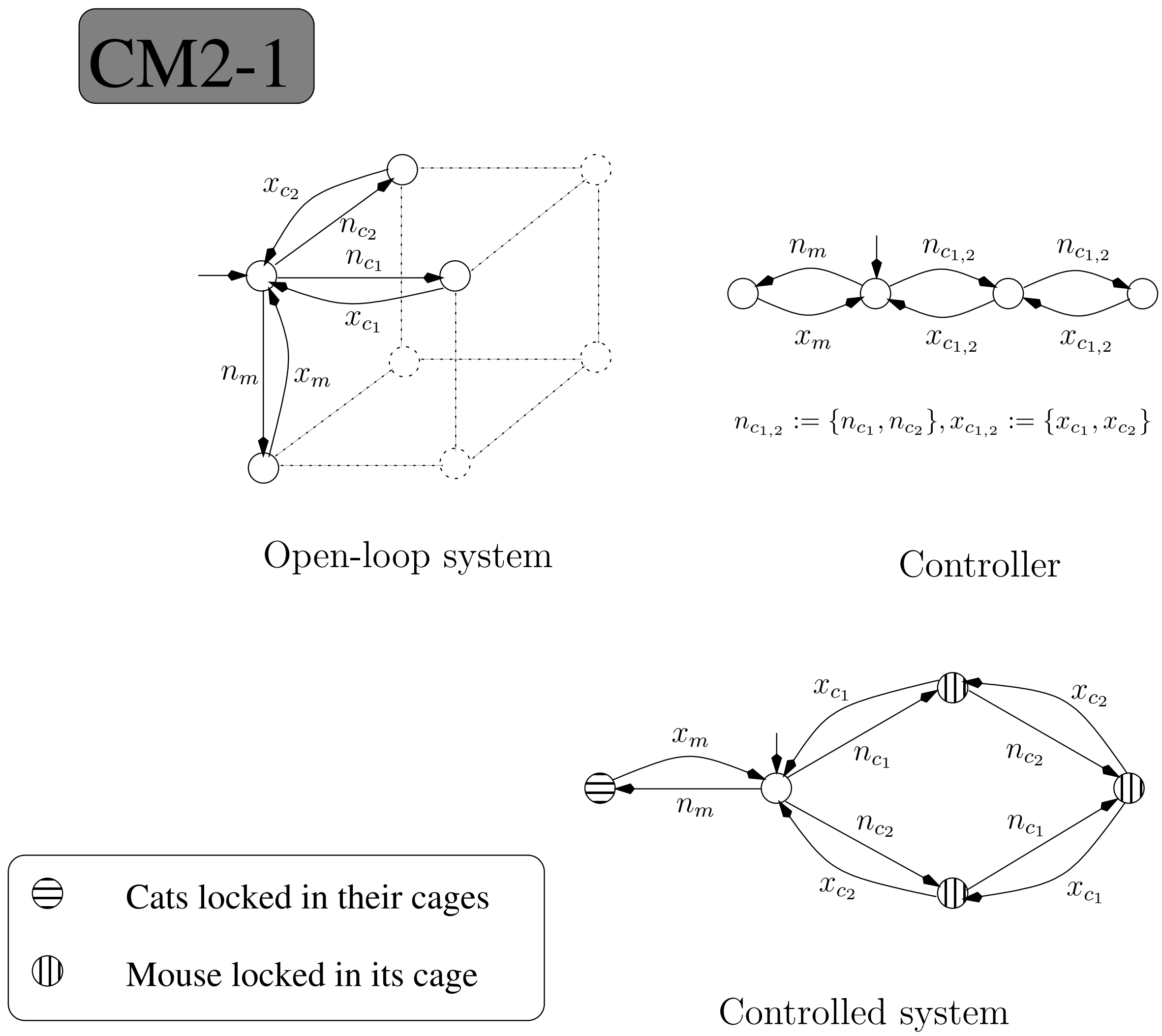


Figure 2.3: Open- and closed-loop systems, together with the controller.

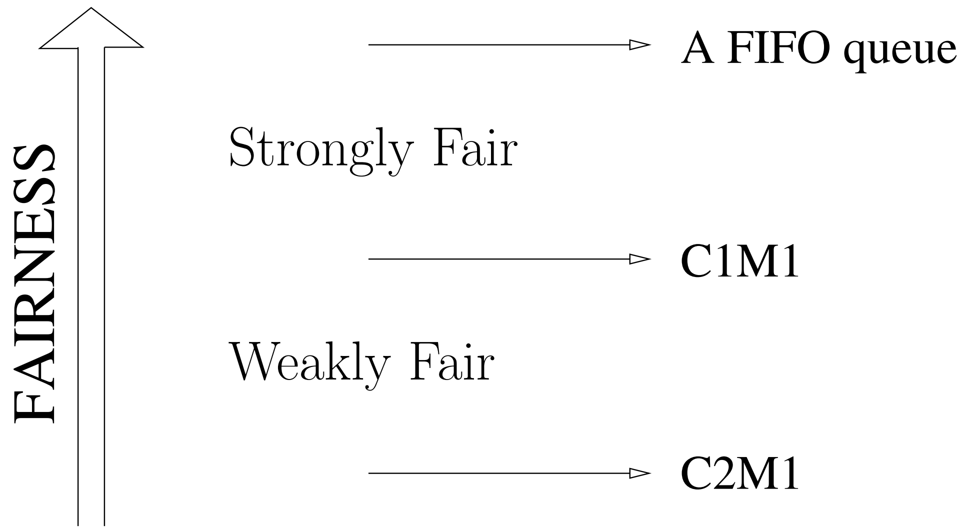


Figure 2.4: Different shades of fairness.

offered by the system if it has a higher priority, denoted by $\sigma \prec \sigma'$, or $(\sigma, \sigma') \in P$. We call P a priority relation on events.

Since we have assumed that the two cats are alike in the real system, it was not reasonable to assign higher priority to either of their exit events over the other. Therefore, the run $n_{c_1}(n_{c_2}x_{c_2})^\omega$ is not typical of the system as x_{c_1} is beaten by x_{c_2} in every turn at the right-hand side state. On the other hand, it may make practical sense to give 'enter' a higher priority over 'exit', at least among the species of the same type, to account for the fact that in a real implementation controllable events, generated by the controller chip, could be a lot faster than events generated by the physical plant. We represent this extra insight into the problem—not derivable in any way from the automaton model—by the following priority relation P:

$$P := \{(x_{c_1}, n_{c_1}), (x_{c_1}, n_{c_2}), (x_{c_2}, n_{c_1}), (x_{c_2}, n_{c_2})\}$$

Addition of priorities weakens the acceptance condition of infinite strings. Ordinarily, an infinite string—or a *run*—must be fair to all components in order to be accepted.

With priorities, a run can be generated by a system if, informally speaking:

1. it can be traced by the automaton model, and

2. it respects the priority relation P, i.e. an event may be continuously ignored only in favor of higher-priority events.

The above set of rules replaces the conventional acceptance condition of a Büchi automaton. C2M1 can be seen to generate the run $n_{c_1}(n_{c_2}x_{c_1}n_{c_1}x_{c_2})^\omega$, which testifies that the mouse is not treated fairly. On the other hand, the infinite sequence $n_{c_1}(n_{c_2}x_{c_2})^\omega$ cannot be generated by the system, and for that it cannot be used as evidence against the fair treatment of the mouse.

As more infinite strings are validated under priorities, it becomes likelier to have fairness violated. When σ' has a higher priority than σ, it is plausible that σ' beats σ in every turn. This may result in unfair treatment of other events which can only be reached through σ. For example, in Figure 2.5, given that β has a higher priority than α, the sequence $\alpha\beta^\omega$ is possible, which clearly treats γ unfairly.

This possibility—that σ' can beat σ in every turn when $\sigma \prec \sigma'$—can be "magnified" by removing σ from the automaton model whenever the two compete against one another. When this removal has been carried out we say that the priorities are *applied* to the automaton.

When priorities are applied, new terminal sets of states may be born (for instance in Figure 2.5, the right-hand side state becomes terminal), and an event is treated unfairly (in a weak sense) if it is disabled in all states of a terminal set. Thus, γ in the example is treated unfairly since it is disabled at the only terminal state of the closed-loop system with priorities applied.

In C2M1, if we apply priorities we get the automaton of Figure 2.6. It is evident that the system is unfair to the mouse, as n_m is disabled in all states of the terminal set formed by the three right-hand side states.

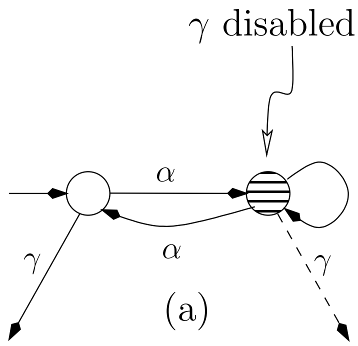


Figure 2.5: (a) A system under control. (b) Assuming that $\alpha \prec \beta$, the resulting system when the priority is applied.

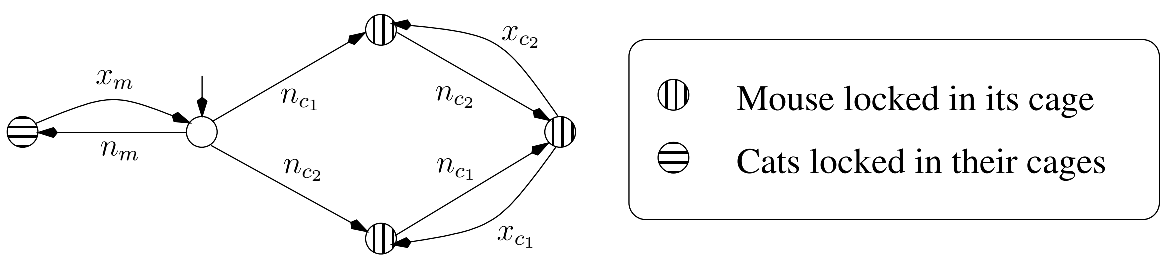


Figure 2.6: C2M1 under control, with priorities applied.

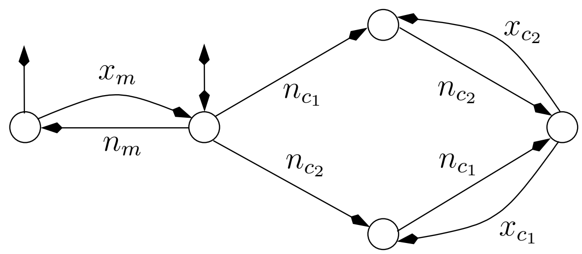


Figure 2.7: The good states are marked in the prioritized automaton. The system is not weakly fair since the resulting automaton is not coreachable.

The above observation can be recast in the RW framework. We regard a state as *good* if n_m is not disabled at that state. A system is weakly fair if with priorities applied, the system can reach good states from any state. But this statement is similar to coreachability of a DES, that is, starting from any state the system can reach marked states. To make the connection precise, we mark good states in the style of RW. Then the claim just made reads: "the system is weakly fair if the prioritized automaton, with good states marked, is coreachable". The claim is demonstrated in Figure 2.7 for C2M1.

In closing, we comment that all through this work we model a system by a *prefix-closed* language. Marking does not enter modeling the plant; rather, it is included in the requirement specification, and the controlled system is expected to show the desired property of coreachability. What is more, our goal is to define different measures of liveness that encompass the notion of coreachability. For instance, no marking of states in C2M1 can detect the unfair behavior addressed in this section: for all possible combinations of state marking, C2M1 is coreachable.

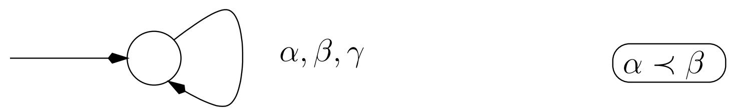


Figure 2.8: Complete model of a system.

2.4 Informal definition

A system is modeled by an automaton and a binary relation P on the set of events. When $(\sigma, \sigma') \in P$, also denoted by $\sigma \prec \sigma'$, the event σ' is assigned a higher *priority* than σ, and in deciding fairness any tie between σ and σ' can be broken in favor of σ'. Thus:

$$\boxed{\textbf{Model} = \textbf{Automaton} + \textbf{Priority Relation}}$$

For example, in the system of Figure 2.8 the automaton is capable of generating any sequence of α's, β's and γ's. However, in the absence of additional information about the system, it will be improbable to have an event neglected all the time. Let us now assume that β can preempt α whenever the two compete against each other. Then it will be tolerated to ignore α at the expense of β in every turn. To take this observation into account, we introduce the following priority relation on the set of events:

$$P = \{(\alpha, \beta)\}$$

The introduction of priorities loosens the acceptance condition on infinite behavior. With no priority relation, the system of the above example is unlikely to generate any "unfair" sequence where an event is continuously neglected. In particular, it cannot perform $(\beta\gamma)^\omega$, as α always has the chance but is never taken. Now suppose we utilize the extra information that $\alpha \prec \beta$. Although α is always ignored in the sequence, we have β—an event with a higher priority—taken often. One might observe

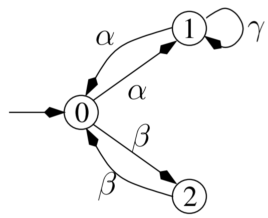


Figure 2.9: The system of Figure 2.8 under control.

that α always gives its turn to β and is excused from being present in the sequence. Therefore, the sequence $(\beta\gamma)^\omega$ becomes probable in the prioritized system, and we should be concerned with how fairly other events are treated along the way. In contrast, γ^ω is unlikely to happen even in the prioritized system since γ has no priority over either of the excluded events α and β, and therefore at least β has to be taken frequently.

Thus, a formerly unlikely infinite string may become likely when a larger priority relation is chosen. In other words, under priorities, *more* infinite strings are probable and it becomes *less* likely as a result for the system to be fair. Consider the system of the previous figure under control, shown in Figure 2.9. The event γ is disabled at states '0' and '2'. Therefore, it would be possible to deny γ the chance of ever being taken only if the system enters the loop consisting of states '0' and '2', which are connected by β transitions, and gets trapped there. This requires always favoring β over α at state '0', which is not likely to happen with no priorities in place. Therefore, the system is deemed fair. However, when $\alpha \prec \beta$, the system can get caught in the loop and still manages to abide by the priority rules. In this case, the system is not fair to γ since γ is disabled at all states of the loop.

Informally speaking, we call a system fair to an event τ if whenever τ is disabled there is an escape to a safe haven—where τ is enabled or becomes ineligible in the plant—through a *maximal priority* string. A string s has maximal priority in a language if at its every prefix, the contributing event has a priority *not* lower than all

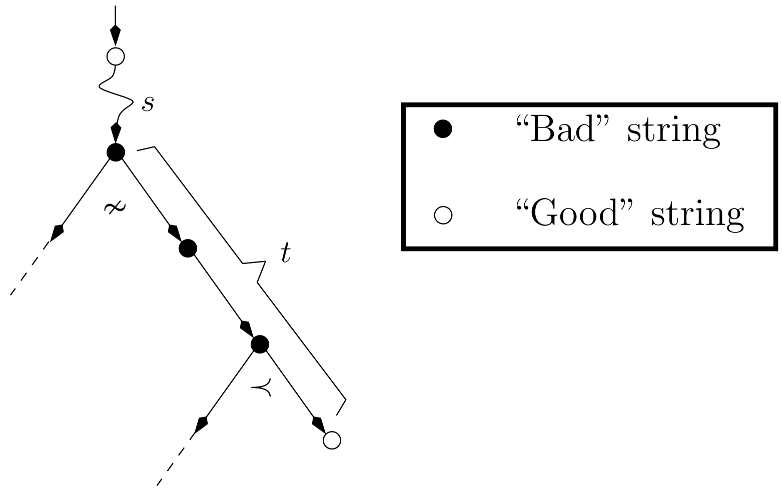


Figure 2.10: A bad string is saved through a maximal priority extension.

its competitors in the language. Priority of σ is *not* lower than σ', i.e. $\neg(\sigma \prec \sigma')$, iff σ has a higher priority than σ' (i.e. $\sigma' \prec \sigma$), or the two priorities are not comparable (i.e. $\neg(\sigma \prec \sigma' \vee \sigma' \prec \sigma)$, denoted by $\sigma \not\sim \sigma'$).

Given an event $\tau \in \Sigma$, we classify the strings of a closed-loop system as *bad* or *good*. A string s is called bad if τ is disabled at that string, and is called good otherwise (i.e. when τ is enabled at s, or it is not possible to follow s with τ in the plant). In Figure 2.10, the bad string s is saved by a maximal priority extension t. Observe that at all prefixes of t, the contributing event's priority is not less than that of any of its competitors.

Back to the example of Figure 2.9, we observe that γ is disabled at $s := \epsilon$. In the absence of priorities, any string has maximal priority. In particular, $t := \alpha$ is a maximal priority extension of s and γ is enabled at st. On the other hand, when $\alpha \prec \beta$, the above extension fails to have maximal priority. As a matter of fact, there is no maximal priority extension t of s such that γ is enabled at st. Any such t must lead the system to state '1' since it is the only state where γ is not disabled. But then we must have a $t' < t$ such that t' leads the system to state '0', where $t'\alpha \leq t$.

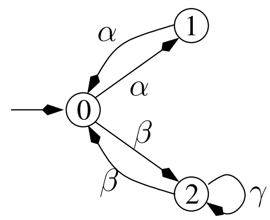


Figure 2.11: Another controller for the system of Figure 2.8.

Consider the above prefix (t') of t. Since we have $t'\alpha \leq t$, $t'\beta$ is in the language, and $\alpha \prec \beta$, t does not have maximal priority.

To consolidate the idea we consider the next example.

Example 1 The open-loop system of Figure 2.8 is supervised by the automaton of Figure 2.11.

When β has a higher priority than α, i.e. when $P := \{(\alpha, \beta)\}$, the system is fair to γ. To keep γ always disabled the controller has to enter and remain in the loop formed by states '0' and '1'. But this requires consistently favoring α over β at state '0' beyond some point, which is not allowed by the priority relation.

Now consider the largest priority relation of all, that is, $P := Pwr(\{\alpha, \beta, \gamma\})$. Thus, every event has a higher priority over every other event, and every infinite string is deemed likely. In particular, the controller can legally perform an α^ω loop, at all states of which γ is disabled, and is therefore unfair to γ. Observe once again that the addition of more priorities has cost us the fairness of the closed-loop system. $\square$

Contrary to the second part of the above example where a symmetric relation P was chosen, often it makes more practical sense to require that P be *anti-symmetric*: it does not make sense to assign α a higher priority than β and vice versa, especially *in the same state*; there ought to be a clear winner if the priorities are comparable at all. In the next chapter we make a case for P being transitive. Finally, the reflexivity

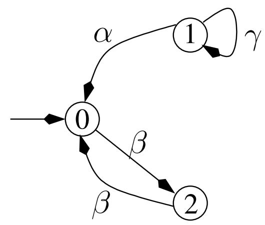


Figure 2.12: The automaton of Figure 2.9 with priorities applied.

of P is not an issue since an event can never compete against itself in a deterministic automaton.

The following remarks are due to be made. Fairness can be measured with respect to an event or a set of events. We call a system fair with respect to a set of events iff it is fair with respect to at least one event in the set. For simplicity here we study fairness with respect to a single event. We name that event τ unless stated otherwise. Another important assumption is that we require fairness of a controller whose job normally is to implement a safety requirement. Thus, *(weak) fairness is a property of the closed-loop system.*

The procedure of verifying fairness for an anti-symmetric and transitive priority relation can be automated as shown below on the example of Figure 2.9. We assume that the system is modeled by an automaton and a priority relation. First, as shown in Figure 2.12, we resolve any non-determinism between α and β by keeping only the event that has *maximal* priority[1], i.e. β. In other words, we apply priorities to the graph *but we do not remove any unreachable states.*

Next, for $i = 0, 1, 2$, when γ is disabled at state i, we find the sets of *terminal* states for the *reachable* automaton initialized at state i. If γ is disabled in all states of at least one terminal set[2], the system is unfair to γ. In the example, let T_i denote the set of terminal sets for a given i, and B be the set of states at which γ is disabled.

[1]Note that if priorities were not comparable, α and β would both have maximal priority.

[2]Note that we have one automaton per bad state, which may have more than one terminal set.

$$i = 0 \qquad T_0 = \big\{\{0,2\}\big\} \qquad B = \{0,2\}$$

$$i = 2 \qquad T_2 = \big\{\{0,2\}\big\} \qquad B = \{0,2\}$$

The system is unfair if for some i there is a subset S of B that is in T_i, i.e.:

$$\exists i, S.\ S \in T_i \wedge S \subseteq B$$

This happens to be the case here, therefore the system of Figure 2.9 is unfair to γ.

Example 2 The open-loop system is specified below. Compared to the previous example, while the state-machine is unchanged, no event has priority over the others:

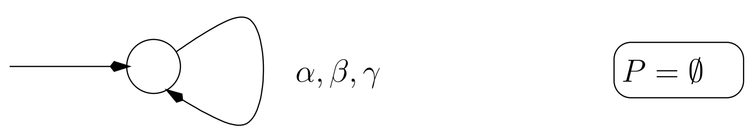


Figure 2.13: The open-loop system of Example 2.

(a) and (b) are two closed-loop behaviors where γ is disabled at the right-hand side state by the controller. In addition, the (implicit) controller in (b) disables β at the same state.

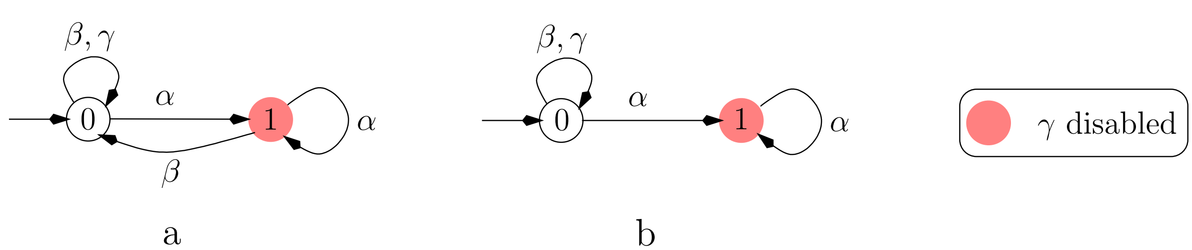


Figure 2.14: The system of Figure 2.13 under two different controllers.

We check fairness to γ algorithmically. Since no priority is specified, the decision is based solely on the above state machines.

For machine (a), only state 1 (the only bad state) need be checked.

$$i = 1: \qquad T_1 = \big\{\{0,1\}\big\} \qquad B = \{1\}$$

Since γ is not disabled at all states of the (only) terminal set (in other words, since $\{0,1\} \nsubseteq \{1\}$), the system is fair.

For machine (b), again state 1 is the only bad state that has to be checked.

$$i = 1: \qquad T_1 = \big\{\{1\}\big\} \qquad B = \{1\}$$

Since $S = \{1\} \subseteq B$ and $S \in T_1$, the system is unfair to γ. In plain English, the system is unfair to γ since γ is disabled at all states of the (only) terminal set. $\square$

It should not come as a surprise that (b), the restriction of a fair behavior (namely, (a)) is unfair. After all, any open-loop system, seen as a system under no control, is fair.

In a slight deviation from the modeling formula at the beginning of this section, we use a formal language in place of an automaton as the first part of a system model. It is our belief that the fairness of a design should not be affected by changes to the underlying *finite* state-machine as long as the generated language[3] remains the same. A string is fair in one iff it is fair in the others. To make our study of fairness robust, we take the language generated by a system as the first component of the model. Thus:

$$\boxed{\textbf{Model} = \textbf{Formal Language} + \textbf{Priority Relation}}$$

[3]Note that here we consider prefix-closed languages only. We will show later that marking can be incorporated into the setup as a system requirement.

Chapter 3

Weak Fairness

In this chapter weak fairness in the context of supervisory control will be studied in detail. To enhance our modelling flexibility the idea of *priority relation* is borrowed from Magee and Kramer [MK99]. After formally defining fairness in Section 3.2, we show in Section 3.3 that nonblocking is a special case of weak fairness where all events have equal priorities (in other words, the priority relation is empty). In Section 3.4 we study the existence of an optimally fair sublanguage of a given language, and find that our definition of fairness is too general to allow the existence of such an optimal element. That motivates us to strengthen our weak fairness condition in two directions. First, in Section 3.5 we limit ourselves to working with a bounded version of the fairness property, and later in Section 3.6 we drop the priority relation in favor of a stronger condition, which requires avoiding all undesirable behaviors independent of the likelihood of their occurrence. Section 3.7 confirms that weak fairness defined as such is stronger than the previous version; however, as now established in Section 3.8 we achieve our desired goal, namely, the existence of an optimally fair sublanguage of a given language.

3.1 Introduction

Weak fairness is a property of the supervised system. A supervisor is less likely to be weakly fair as it continues to disable an event. For convenience, we call a string *bad* if the event with respect to which fairness is gauged (denoted by τ unless stated otherwise) is disabled by the supervisor. A string is called *good* otherwise.

Let K be the closed language of the supervised system, and $s \in K$ be a bad string. Consider the computation tree rooted at s, that is, all possible continuations of s in the language K. Then weak fairness requires that eventually a good string be reached. In Ramadge and Wonham supervisory control of discrete-event systems, a nonblocking supervisor ensures that there exists a path along which this is the case. In the logic of CTL [CES86] this can be expressed succinctly by $E\Diamond g$, where g is an atomic formula true of a good state. It reads "there exists a path (E) along which g is eventually ($\Diamond$) true."

At the other end of the spectrum, in Section 3.6 we state a much stronger property requiring that good states be reached eventually along all paths, i.e. $A\Diamond g$.

The first four sections of this chapter are devoted to the study of cases in between ("there exists" and "for all"), where only some continuations of a string are of interest. For example, in a real application we might have a continuation from a bad string to a good one, but the high cost of taking that continuation could render it useless. Or we might want to warn others of its slowness, which may result in its being preempted by faster but undesirable continuations. To accommodate this, we introduce a binary relation P on the set of events. We denote:

$$\alpha \prec \beta \text{ iff } (\alpha, \beta) \in P$$

$\alpha \prec \beta$ may read "β is faster than α", or simply "α is dominated by β". We assume that P is irreflexive (α cannot be faster than itself), anti-symmetric (if β is faster than α then α cannot be faster than β), and transitive (γ is faster than β and β is faster than α imply γ is faster than α).

The relation is considered transitive first because it makes more practical sense to have it defined that way, and second, it resolves any ambiguity as to how to interpret the priorities between more than two competing events, as the following example suggests.

Example 3 Suppose that we know about the system of Figure 3.1 that $\alpha \prec \beta$ and $\beta \prec \gamma$, and we would like to compare the priorities of α and γ. Specifically, we would like to know which continuation of $s = \epsilon$ has maximal priority (or is faster than the rest).

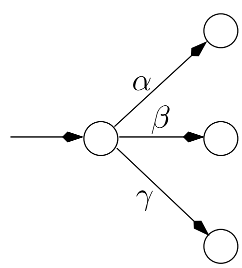


Figure 3.1: Determining maximal priority between three events is ambiguous if P is not transitive.

With a non-transitive relation the figure can be interpreted in two different ways:

1. First β beats α, becoming the winner between the two. Then in the next round it loses the competition to γ since $\beta \prec \gamma$. γ is the ultimate winner and the event with maximal priority at $s = \epsilon$.

2. First β is beaten by γ. The winner, γ, is not comparable to α, and we have no justification to break the tie between them. Both α and γ provide maximal priority exits from $s = \epsilon$.

Although normally α is dominated by β, in the second reading β is not present to dominate α since it gives way to γ right at the beginning. What we are highlighting is the well known fact that only when $\prec$ is transitive is the max operator associative:

$$\max(\alpha,\beta,\gamma) = \max\big(\alpha, \max(\beta,\gamma)\big) = \max\big(\max(\alpha,\beta),\gamma\big)$$

$\square$

3.1.1 Priority relation as an acceptance condition

We now recast the priority relation as a condition for accepting infinite strings generated by an automaton.

First we extend priorities to strings. Let s_1 and s_2 be two nonempty strings in Σ^+. s_2 has a higher priority than s_1, denoted by $s_1 \prec s_2$, iff: (see Figure 3.2)

$$\exists u, \sigma_1, \sigma_2.\ u\sigma_1 \leq s_1 \wedge u\sigma_2 \leq s_2 \wedge \sigma_1 \prec \sigma_2$$

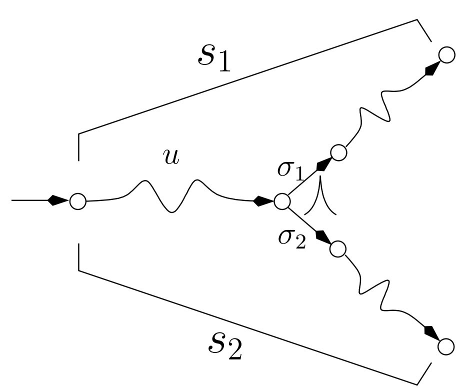


Figure 3.2: s_2 has a higher priority than s_1.

Thus, s_2 has a higher priority than s_1 if at the point of departure (u in the definition) it has a higher priority than s_1. The definition is handy in comparing the priorities of different continuations of a string in a language. As an example, consider a language L defined as follows:

$$L = \overline{\alpha(\beta\alpha^2 + \alpha\beta^2)}$$

Assuming that $\alpha \prec \beta$, it follows from the definition that:

$$\alpha\beta^2 \prec \beta\alpha^2$$

which is a useful piece of information about the system since the above strings both extend $s = \alpha$ in L. On the other hand, in the context of language L the comparison

$$\alpha^2 \prec \alpha\beta^2$$

is meaningless since α^2 extends $s = \alpha\beta$ and $s = \epsilon$, whereas $\alpha\beta^2$ is a continuation of $s = \alpha$ in L.

We now revisit the C2M1 example, where two cats and a mouse can roam round a room, while a supervisor ensures that the two species do not occupy the room at the same time.

Consider the sequence $n_{c_1}(n_{c_2}x_{c_2})^\omega$. Given the above automaton model and assuming that the system runs *ad infinitum*, the sequence is not likely to occur as beyond $s = n_{c_1}$, the event x_{c_1} is repeatedly overtaken by n_{c_2} and x_{c_2} at states 1 and 3, respectively. If there is the possibility of being trapped in the $(n_{c_2}x_{c_2})^\omega$ loop, we need to emphasize the likelihood of x_{c_1} being overtaken by n_{c_2} at state 1, and by x_{c_2} at state 3. In a priority setting, this is stated by declaring that $x_{c_1} \prec n_{c_2}$ and $x_{c_1} \prec x_{c_2}$. Then it is justifiable that x_{c_1} be overtaken in every turn by the competing

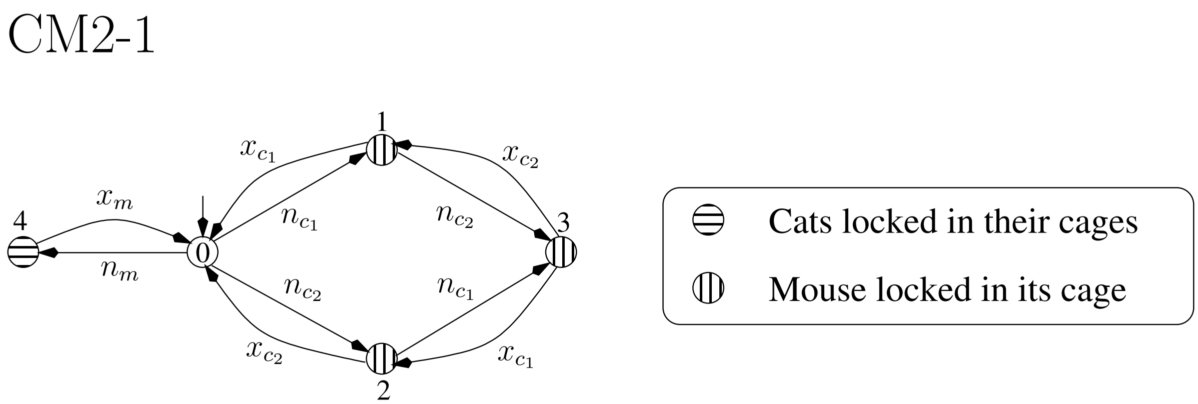


Figure 3.3: C2M1 revisited.

events and the infinite sequence becomes a possible behavior of the system.

In the original priority assignment, we gave higher priority to cats entering the chamber over their exiting, i.e.:

$$x_{c_{1,2}} \prec n_{c_{1,2}}$$

With this assignment, the sequence $s = n_{c_1}(n_{c_2}x_{c_1}n_{c_1}x_{c_2})^\omega$ is "fair" to all enabled events; although x_{c_1} is preempted consistently by n_{c_2} at state 1, the preemption is justifiable since $x_{c_1} \prec n_{c_2}$. A similar argument explains why x_{c_2} can be consistently neglected at state 2.

The infinite string $s = n_{c_1}(n_{c_2}x_{c_2})^\omega$, on the other hand, does not treat x_{c_1} fairly. At state 3, x_{c_2} is always preferred over x_{c_1} without having priority over it.

3.2 Formal definition

Let $K \subseteq L \subseteq \Sigma^*$ be prefix-closed languages over Σ. L and K model the open- and closed-loop behavior of a system, respectively. An event $\tau \in \Sigma$ is *disabled* by K at $s \in K$ iff

$$s\tau \in L \text{ but } s\tau \notin K, \quad \text{or equivalently } s\tau \in L - K.$$

τ is said to be *enabled* at s by K iff $s\tau \in K$, and is eligible to occur in L (or simply *eligible*) iff $s\tau \in L$. When clear from the context we drop references to K and L. Observe that when an event is not disabled it could either be enabled or ineligible, so "disabled" and "enabled" do not convey opposite meanings. In other words:

$$\text{not (disabled)} \equiv \text{ (enabled or not eligible)} \equiv \text{(eligible}\Rightarrow\text{enabled)}$$

or:

$$s\tau \notin L - K \leftrightarrow (s\tau \in L \rightarrow s\tau \in K)$$

The objective of control is to design controllers that are fair to a set of events. To simplify the discussion we assume that the set in question is a singleton $\{\tau\}$.

We partition K into the set of *good* and *bad* strings. With τ fixed, a string s is called *bad* iff τ is disabled at s, and is called *good* otherwise. Thus, each event τ partitions K into the set of good and bad strings according to:

$$K_b := \big\{s \in K |\ s\tau \in L - K\big\},\ K_g := K - K_b$$

Therefore we have:

$$K = K_g \dot{\cup} K_b$$

We are now ready to define a fair sublanguage of L given a priority relation P.

$$\boxed{\begin{array}{l}\text{A controlled language } K \text{ is fair iff for all } s \in K_b,\\ \exists t.\ st \in K_g \wedge \big(\forall t' \in \Sigma^*.\ \ t \prec t' \rightarrow st' \notin K\big)\end{array}}$$

The definition calls for the existence, for each bad string s, of a maximal-priority continuation t leading to a good string. We call such a continuation a *savior*. It is immediate from the definition that a savior must have length at least equal to one.

As the first part of the definition suggests, a savior t has to lead to a good string, i.e. $st \in K_g$. The second part demands that event-by-event, the continuation should have a *maximal* priority: while s is evolving to st, at any juncture the event that has just become part of t must have a higher priority than its competitors, or be at least incomparable to them.

When $P = \emptyset$, the condition in the definition of fairness reduces to:

$$\forall s \in K.\ s \in K_b \rightarrow \exists t.\ st \in K_g$$

3.3 Nonblocking incorporated

Let E and M be languages over Σ. (E, M) is a *pair of machine-generated behaviors* iff $\overline{M} = M$ and $E \subseteq M$ [Rut00]. E and M respectively model the marked- and closed-behavior of a DES, and together they provide a complete description of the system behavior. A string of M is *coreachable* if it can be completed to a string of E. A system is coreachable if all strings of M are coreachable, in other words:

$$\overline{E} = M$$

When (E, M) represents a system under supervision, (E, M) not coreachable could mean that the supervisor may have inadvertently blocked the system by disabling all possible choices leading to E at a string of $M - \overline{E}$. In that case the supervisor is called *blocking*.

We recall that a system is coreachable iff $\overline{E} = M$. Since the inclusion from left to right is automatic, the condition is equivalent to $\overline{E} \supseteq M$. The condition can be rewritten as:

$$\begin{aligned} & \forall s.\ s \in M \rightarrow s \in \overline{E} \\ = \ & \forall s \in M.\ \exists t.\ st \in E \\ = \ & \forall s \in M.\ (s \in E \rightarrow \exists t.\ st \in E) \wedge (s \notin E \rightarrow \exists t.\ st \in E) \end{aligned}$$

Note that the first conjunct is always true upon choosing $t = \epsilon$. Thus:

$$\begin{aligned} = \ & \forall s \in M.\ s \notin E \rightarrow \exists t.\ st \in E \\ = \ & \forall s \in M.\ s \in M - E \rightarrow \exists t.\ st \in E \end{aligned}$$

Denoting $M_g := E$ and $M_b := M - E$, the condition for coreachability becomes:

$$\forall s \in M.\ s \in M_b \rightarrow \exists t.\ st \in M_g$$

which coincides with the definition of fairness when $P = \emptyset$. Thus, checking (E, M) for coreachability is equivalent to checking M for fairness with no priorities, where the strings in E are marked as good. Therefore the coreachability problem can be viewed as a special case of weak fairness where no priority is given.

As an example, consider the marked closed-loop system shown in Figure 3.4. The system is considered coreachable as it is always possible to extend an unmarked string to a marked one. However, if β had a higher priority than α, then it would be possible never to take the transition to the right. Thus, with $\alpha \prec \beta$ the system is not "truly" coreachable.

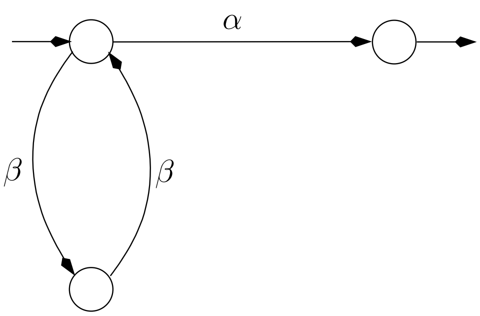


Figure 3.4: A coreachable closed-loop system. Note that in reality the system may never reach the marker state if $\alpha \prec \beta$.

3.4 Fairness under union

3.4.1 No priority ($P = \emptyset$)

In this subsection we show that with no priority relation given, fairness is preserved under arbitrary union. This is consistent with the Ramadge and Wonham [RW87] result that an optimal nonblocking supervisory control exists.

The following result will be used frequently.

Lemma 3.1 *Let I be an index set and $K := \bigcup_{i \in I} K_i$. Then we have:*

$$K_b \subseteq \bigcup_{i \in I} K_{i_b} \tag{3.1}$$

$$K_g = \bigcup_{i \in I} K_{i_g} \tag{3.2}$$

Proof.

(3.1). Let $s \in K_b$. We show that s must belong to K_{i_b} for some $i \in I$.

By definition $s \in K_b$ is equivalent to $s \in K$ and $s\tau \in L - K$, where τ is the event with respect to which fairness is measured. Since $s \in K$ we must have $s \in K_i$ for some $i \in I$. On the other hand, $s\tau \notin K$ implies $s\tau \notin K_j$ for all $j \in I$. In particular, $s\tau \in L - K_i$, i.e. $s \in K_{i_b}$.

(3.2,$\supseteq$). Let s be in K_{j_g} for some $j \in I$. We must show that $s \in K_g$.

We have $s \in K_{j_g}$, i.e. $s \in K_j$ and

$$s\tau \in L \rightarrow s\tau \in K_j$$

Since $K_j \subseteq K$ it follows that $s \in K$ and

$$s\tau \in L \rightarrow s\tau \in K$$

i.e. $s \in K_g$.

(3.2,$\subseteq$). Let s be in K_g. We must show $s \in K_{i_g}$ for some $i \in I$.

We have that $s \in K_g$, implying $s \in K$ and

$$s\tau \in L \rightarrow s\tau \in K$$

If $s\tau \notin L$, then from $s \in K$ it follows that $s \in K_i$ for some $i \in I$. In this case it is trivially true that

$$s\tau \in L \rightarrow s\tau \in K_i$$

and therefore $s \in K_{i_g}$. Otherwise, if $s\tau \in L$ we must have $s\tau \in K$ by the assumption, which implies that $s\tau \in K_i$ for some $i \in I$. Since K_i is closed it follows that $s \in K_i$. Hence we have found $s \in K_i$ such that

$$s\tau \in L \rightarrow s\tau \in K_i$$

i.e. $s \in K_{i_g}$. ∎

Note. (3.1) can be expressed string-wise as:

$$\forall s.\ s \in K_b \rightarrow (\exists j.\ s \in K_{j_b})$$

With a slight modification in the proof we can actually establish a stronger result, namely:

$$\forall s.\ s \in K_b \rightarrow \big((\exists j.\ s \in K_{j_b}) \wedge (\forall i.\ s \in K_i \rightarrow s \in K_{i_b})\big)$$

or equivalently,

$$\forall s.\ s \in K_b \rightarrow \big((\exists j.\ s \in K_j) \wedge (\forall i.\ s \in K_i \rightarrow s \in K_{i_b})\big)$$

The first conjunct in the right-hand side is immediate from the fact that $K_j \subseteq K$. Therefore, the more informative part of the above statement is:

$$\forall s \in K.\ s \in K_b \rightarrow (\forall i.\ s \in K_i \rightarrow s \in K_{i_b})$$

which states that "if a string is bad in a union then it must be bad in any participating component that contributes that string." It is readily seen that the statement is also valid in the other direction, that is:

$$\forall s \in K.\ s \in K_b \leftarrow (\forall i.\ s \in K_i \rightarrow s \in K_{i_b})$$

while as the following example suggests, the statement of the theorem cannot be reversed.

Example 4 In this example we show that in general it is not true that $K_b \supseteq \bigcup_{i \in I} K_{i_b}$. In Figure 3.5, s is defined to be bad if β is disabled at s, and L is taken to be equal to K_2. We have:

$$K_1 \subseteq K_2 = K_1 \cup K_2$$

The string $s = \epsilon$ is bad in K_1 but good in $K_2 = K_1 \cup K_2$, as β is enabled at $s = \epsilon$ in K_2.

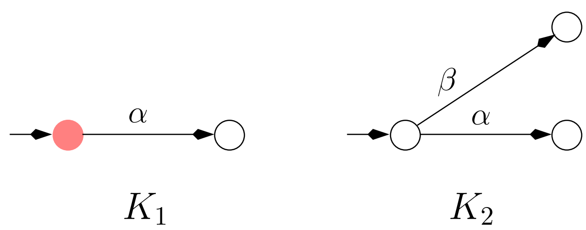


Figure 3.5: $K_b \not\supseteq K_{1_b} \cup K_{2_b}$, where $K = K_1 \cup K_2$.

$\square$

We now prove that fairness is preserved under arbitrary union when $P = \emptyset$.

Theorem 3.1 *Let $P = \emptyset$ and I be an index set. Then $K := \bigcup_{i \in I} K_i$ is fair if K_i is fair for all $i \in I$.*

Proof. Let $s \in K_b$ and K_i be fair for all $i \in I$. Then by the previous result s must belong to K_{j_b} for some $j \in I$. Since K_j is by assumption fair we must have a $t \in \Sigma^*$ such that $st \in K_{j_g}$. It follows that $st \in K_g$, implying that K is fair. $\blacksquare$

3.4.2 General case

In this section we aim at proving a similar result when P is nonempty. First we show that fairness is preserved under finite union.

Lemma 3.2 *Let K_1 and K_2 be fair. Then $K := K_1 \cup K_2$ is fair.*

Proof. Let $s \in K_b$. Then we must have, for example, $s \in K_{1_b}$. Let t_1 be a savior of s in K_1; that is, $st_1 \in K_{1_g}$ and t_1 has maximal priority among all continuations of s in K_1. Note that since K_1 is fair such a savior must exist. From (3.2) it follows that $st_1 \in K_g$. If t_1 is not a savior of s in K, we must have (Figure):

$$\exists \sigma_1, \sigma_2 \in \Sigma, u \in \Sigma^*.\ u\sigma_1 \le t_1 \wedge su\sigma_2 \in K \wedge (\sigma_1, \sigma_2) \in P$$

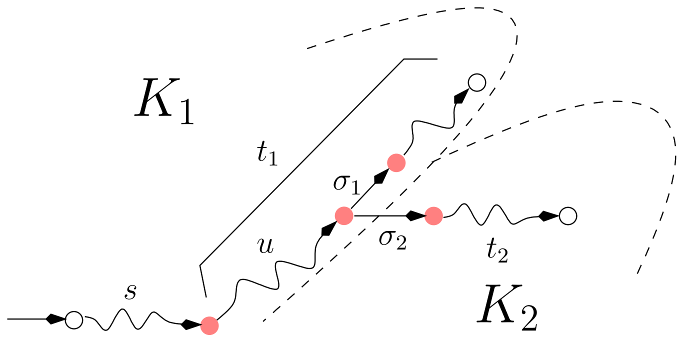


Figure 3.6: (Unbounded) fairness is preserved under finite union.

In other words, σ_2, with its higher priority, "derails" t_1 in its attempt to save s in K. Pick u with the smallest length then σ_2 with maximal priority [1] satisfying the above. Observe that $su\sigma_2$ cannot belong to K_1, for otherwise t_1 would not have maximal priority. Therefore it must be in K_2. If $su\sigma_2$ is good, $u\sigma_2$ is a savior of s in K and the proof is complete. Otherwise, if $su\sigma_2$ is a bad string in K, by Lemma 3.1 it must also be bad in K_2. But K_2 is fair and therefore $su\sigma_2$ is saved in K_2 by a continuation, say t_2, of maximal priority. The fact that $su\sigma_2 t_2$ is good in K_2 implies that it is also good in K. Moreover, we observe that the continuation t_2 lies entirely outside K_1 and therefore it cannot be preempted when adding K_1 to K_2 to form the union. Hence, t_2 is a savior of $su\sigma_2$ in K and therefore $u\sigma_2 t_2$ is a savior of s in K. ■

But as the following example demonstrates, fairness is not preserved under countable union, even for chains of fair languages.

Example 5 The open-loop system is shown in Figure 3.7.

We assume that fairness is measured with respect to γ. Then the sublanguages $K_i \subseteq L$, $i \in \mathbb{N}$, shown in Figure 3.8, are all fair, whereas the limit

$$K := \bigcup_{i \in \mathbb{N}} K_i$$

[1] $\sigma \in \Sigma$ has maximal priority at s in K iff $\forall \sigma' \in \Sigma.\ s\sigma' \in K \rightarrow \neg\sigma \prec \sigma'$.

Figure 3.7: The open-loop system.

shown in the same figure, is not. Note that

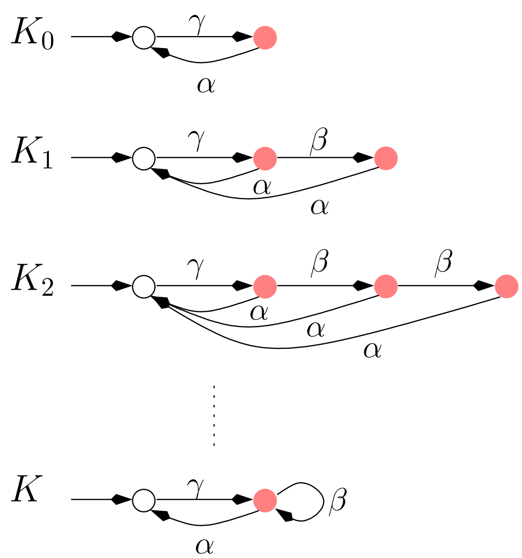


Figure 3.8: A chain of fair restrictions of 3.7 and its unfair limit.

$$K_0 \subseteq K_1 \subseteq K_2 \subseteq \cdots$$

To obtain K_i from K, the β selfloop is unwound i times and then β is disabled. That gives α a chance to become the event with maximal priority and to save all bad strings of K_i.

On the other hand, the bad string $s = \gamma$ does not have a savior in K. Observe that any such candidate should terminate with α; without loss of generality assume $t = \beta^m \alpha$ for some $m \geq 0$. Since $\gamma\beta^m\beta \in K$ and $\alpha \prec \beta$, it follows that the $\beta^m\alpha$

continuation of $s = \gamma$ does not have maximal priority. □

3.5 Bounded fairness

The result of the previous section, that fairness is not preserved under countable union, is a major setback in computing a fair design since the fair sublanguages of a given language do not in general form a semi-lattice with respect to union, and a "largest" (i.e. one that contains the others) fair sublanguage may not exist as a consequence.

When this property is lost, it is not intuitively obvious what is a best fair solution. As the previous example shows, the answer largely depends on how long (that is, how many transitions) we can tolerate before γ is enabled. For instance, if it is acceptable to keep γ disabled along as many as two transitions, then K_2 seems a reasonable "optimal" solution. The problem arises when one is less specific, and is content with a general guarantee like "γ can be disabled for *any* finite number of steps", or equivalently, "γ is enabled *eventually*". A property cannot be met optimally in time when no deadline for its satisfaction is specified.

It is our contention that a suboptimal solution is indeed optimal under a more stringent set of conditions. As suggested above, we propose to place a bound on the length of the savior.

In the example, if we specify that a bad string must be saved by a continuation of length at most, say 5, then all designs $K_5, K_6, \ldots$ are unfair to γ since $\beta^i\alpha$, the savior of minimum length in K_i of the bad string $s = \gamma$, has length $i + 1$. Therefore, only designs indexed with

$$i + 1 \leq 5 \quad \text{or} \quad i \leq 4$$

are fair.

Definition 3.1 *Let* $\sigma_i \in \Sigma$ *for* $i \in \mathbb{N}$*,* $n \in \mathbb{N}$ *be a constant, and* $s = \sigma_1\sigma_2 \dots \sigma_n$*. The length of* s*, denoted by* $|s|$*, is defined to be equal to* n*.*

Definition 3.2 *Let* $L \subseteq \Sigma^*$*,* P *be a priority relation on events, and* $N \in \mathbb{N}$ *be a constant (bound). A controlled language* $K \subseteq L$ *is* N*-fair if for every* $s \in K_b$ *we have:*

$$\boxed{\exists t.\ |t| \leq N \wedge st \in K_g \wedge \big(\forall t' \in \Sigma^*.\ t \prec t' \rightarrow st' \notin K\big)}$$

The following result states that the finite union of bounded-fair languages is also bounded-fair, where the length of the savior is bounded by the sum of the constituting bounds.

Theorem 3.2 *Let* K_1 *and* K_2 *be* N_1*- and* N_2*-fair, respectively. Then* $K = K_1 \cup K_2$ *is* $(N_1 + N_2)$*-fair.*

Proof. Let s be a bad string of K. Recalling the proof of Lemma 3.2, we observe that s is led to a good string by a maximal priority continuation, which is either:

1. t_1, where t_1 is a savior of s in K_1. Since K_1 is N_1-fair, we pick a t_1 that satisfies the length criterion, i.e. $|t_1| \leq N_1$. We have:

$$N_1 \leq N_1 + N_2$$

2. $u\sigma_2$, where $su\sigma_2$ is good in K. In this case we have:

$$\begin{aligned} |u\sigma_2| = |u| + |\sigma_2| &\leq (N_1 - 1) + 1 \qquad \text{(since } u < t_1\text{)} \\ &= N_1 \\ &\leq N_1 + N_2 \end{aligned}$$

3. $u\sigma_2 t_2$, where $su\sigma_2$ is bad in K and K_2, but is saved in K_2 and therefore in K by t_2. Since K_2 is N_2-fair, we pick a t_2 that satisfies the length criterion. We have:

$$|u\sigma_2 t_2| = |u| + |\sigma_2| + |t_2| \leq (N_1 - 1) + 1 + N_2 = N_1 + N_2$$

Hence, in all cases, the length of a savior is bounded by $N_1 + N_2$. ■

The previous result offers a worst case estimate on the bound of a union. Concern is raised about the existence of a bound for arbitrary unions if the constituting bounds stack up as predicted by the lemma. Unfortunately, as the following example suggests, this could be the case.

Example 6 Consider a plant that is always capable of doing any event from the set $\Sigma = \{\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma, \lambda\}$. K_1 and K_2 are two restrictions which are 1-fair to λ. In particular, $s = \epsilon$ is saved by α_1 in K_1 and by α_2 in K_2, both having maximal priorities within their respective sublanguages. Note that α_1 can only be preempted by β_2 which is not present in K_1, and vice versa.

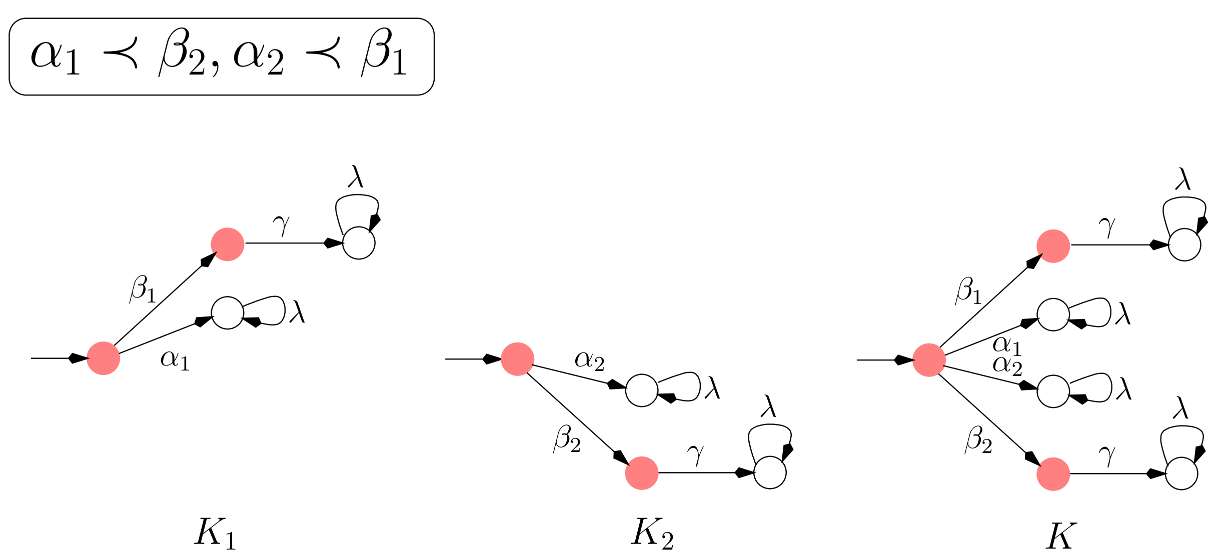


Figure 3.9: N-fairness is not preserved under finite union.

On the other hand, $K = K_1 \cup K_2$ is not 1-fair to λ (although it is 2-fair). The string $s = \epsilon$ cannot be saved by either of α_1 or α_2 anymore, as they lose competition to β_2 and β_1, respectively. Thus, the bad string $s = \epsilon$ is saved in K by either of $\beta_1\gamma$ or $\beta_2\gamma$, and therefore K is a 2-fair[2] sublanguage. □

Thus, we have the following result.

Theorem 3.3 *For $N \geq 1$, the subclass of N-fair sublanguages is not closed under union.*

Note that a language is 0-fair if all its strings are good, and as previously shown a good string remains good in the union. Thus, the class of 0-fair sublanguages is closed under arbitrary union.

We conclude that an optimal N-fair sublanguage of a given language does not exist in general.

Example 7 In Figure 3.10, K_{max_1} and K_{max_2} are both 1-fair sublanguages of K of the previous example, while their union, which happens to be K itself, is not.

We note that both K_{max_1} and K_{max_2} are *maximal* sublanguages satisfying the 1-fairness criterion, in the sense that, take K_{max_1} for example, if one added the string β_2, which is the common prefix of all strings in $K - K_{max_1}$, the 1-fairness condition would be violated. □

N-fairness, however, is preserved under union in infinite chains, a result that will lend itself in Appendix A to the existence of maximal elements in the poset of N-fair

[2] In the current definition of bounded-fairness, if a language is k-fair, $k \in \mathbb{N}$, it is also $(k+i)$-fair for any non-negative integer i. Thus we have the following chain of subclasses of bounded-fair languages:

$$0-\text{fair} \subseteq 1-\text{fair} \subseteq 2-\text{fair} \subseteq \cdots$$

One could alternatively consider a language N-fair if every bad string has a savior of length at most N, and at least one bad string exists whose savior has length *exactly* equal to N. Then the bounded-fair subclasses would be disjoint.

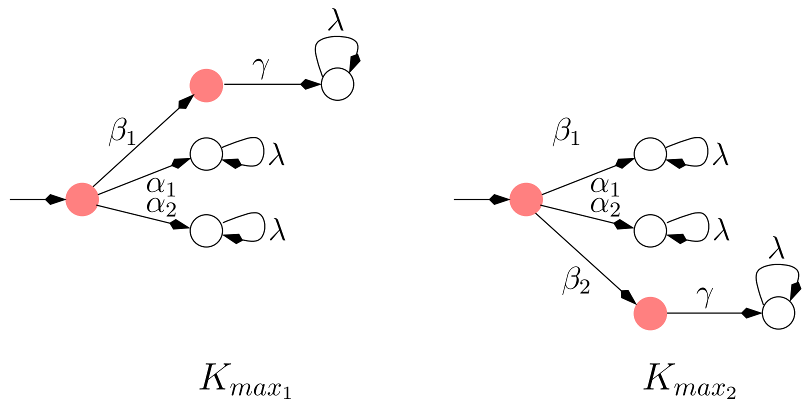


Figure 3.10: Two maximally 1-fair sublanguages of K of the previous example.

sublanguages of a given language. The proof of the following result relies on the fact that Σ, the alphabet of events, is a finite set.

Lemma 3.3 *Let*

$$K_0 \subseteq K_1 \subseteq K_2 \subseteq \cdots$$

be an increasing chain of N-fair languages, $N \in \mathbb{N}$. Then the limit

$$K := \bigcup_{i \in \mathbb{N}} K_i$$

is N-fair.

Proof. Let $s \in K_b$ be a bad string of K. Then $s \in K_{i_{0b}}$ for some $i_0 \in \mathbb{N}$. Since K_{i_0} is N-fair, s must have a savior in K_{i_0}, denoted by t_0, of length at most N. If t_0 is not a savior of s in K, then it must be preempted by an event σ_1 with higher priority. The string ended with σ_1 cannot belong to K_i for any $i \leq i_0$, because otherwise t_0 will not be a savior in K_{i_0}. Thus it must belong to K_{i_1}, for some $i_1 > i_0$. Since K_{i_1} is N-fair, s must have a savior of length bounded by N in K_{i_1}. Call this continuation t_1. We note that since $\{K_i\}_{i \in \mathbb{N}}$ is an increasing chain, the continuation t_0 is also in K_{i_1}, but it no longer has maximal priority. If t_1 is not a savior of s in K, we repeat the procedure and form a sequence (cf. Figure 3.11)

$$K_{i_0} \subset K_{i_1} \subset K_{i_2} \subset \cdots$$

where $i_0 < i_1 < i_2 < \cdots$ and s has a savior t_i in K_i of length bounded by N with an increasing order of priority:

$$t_0 \prec t_1 \prec t_2 \prec \cdots$$

Since s can be continued only by $|\Sigma|\frac{|\Sigma|^N-1}{|\Sigma|-1}$ different nonempty strings of length at most N, this procedure must terminate after a finite number of applications, say, $n = n^*$. This means that t_{n^*} is a savior of s in K of length bounded by N. ■

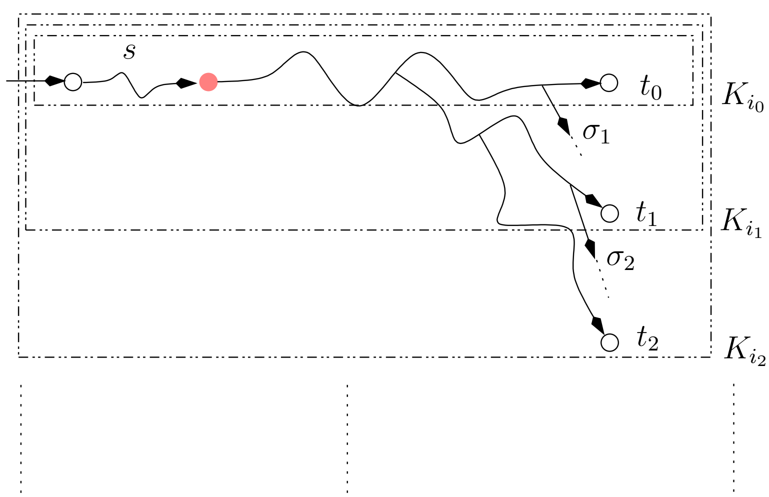


Figure 3.11: Illustration of Lemma 3.3.

3.6 Another stab at fairness: optimally fair

3.6.1 Introduction

It appears that although allowing priorities among different events provides the designer with more flexibility in modeling a physical system, it makes the construction of a “best” solution rather hard, if not impossible. Returning to Example 7, Page 51, we observe that we have two maximally 1-fair sublanguages of K, but a supremum or largest 1-fair solution does not exist. Neither is it intuitively clear which one is the more “natural” of the two and should be preferred.

It seems to us that a systematic approach to computing a maximum within a class should involve imposing additional constraints to be satisfied by candidate solutions, so that the restricted class of solutions *does* have a unique maximum, or even more desirable, a supremum.

Towards this end we propose a more stringent fairness condition. The new condition is not off the target in that it has a reasonable physical appeal. On the other hand, it is easily verifiable that a language is (bounded) fair in the old regime if it is so under the new one; thus the new class of solutions is contained in the old one. It turns out that arbitrary union is preserved in the new class, and therefore a supremal element exists. Through an example we will see that under the new condition the supremal element does not necessarily coincide with any of the old maxima; however, since the fairness condition is strengthened it is immediate that the new supremum is smaller than or equal to one of the old maxima.

3.6.2 Fairness revisited

Previously, in our treatment of fairness we allowed priorities, meaning that when we are caught in a loop, we are safe if we can escape through an exit of maximal priority.

This gave us extra modeling flexibility, in that the mere existence of a bad loop (or a sequence of bad strings of length greater than N when studying bounded fairness) is not sufficient to make a system unfair. To be considered unfair, in addition, the loop must have maximal priority, so that all exits to safety lose to their competitors participating from the loop.

As we have argued above, the flexibility provided in modeling exacts a cost: namely, it comes at the expense of losing optimality. We now investigate the other side of the tradeoff between expressibility and computability, namely, we let the mere existence of a bad loop constitute unfair behavior. In effect, we adopt a pessimistic strategy, where transitions leading to longer disablement of an event will always prevail. In a way, we have a "dynamic" priority relation in place. It can foresee in which path τ is disabled the longest, and assigns higher priorities to the constituting events accordingly. Thus, from a bad string, a transition leading to a bad behavior is given priority over one leading to a good one, independent of their labels[3].

Remark: To keep our terminology simple we continue to refer to the new property as "N-fairness". Despite their identical names, the new property should not be confused with the one introduced earlier in Definition 3.2, Page 49. Our reference to different types of fairness should be clear from the context, and is kept implicit except in the next section where a formal comparison between different types of fairness is made. However, in the remainder of this thesis and unless stated otherwise, the property defined next is the one referred to as "N-fairness".

A controlled language K is N-fair, $N \in \mathbb{N}$, iff for all $s \in K_b$,

$$(\exists t \in \Sigma^*.\ t \neq \epsilon \wedge st \in K) \wedge (\forall t \in \Sigma^*.\ |t| = N \wedge st \in K \rightarrow \exists u \leq t.\ su \in K_g)$$

The first conjunct states that a bad string cannot be *terminal*—a string is terminal

[3] So it is possible to have β beating α in one place and α beating β in the other.

in a prefix-closed language K if it cannot be continued in K—which is a reasonable demand that any fairness criterion should make: otherwise a controller can bring the system to a complete halt by disabling the only choice for the next move, which happens to be τ.

Before interpreting the second conjunct we expand our terminology with the following definition. A *run* of s of length $M \in \mathbb{N}$ is any sequence

$$\{s, s\sigma_1, \ldots, s\sigma_1 \ldots \sigma_M\}$$

where $\sigma_1, \ldots, \sigma_M \in \Sigma$. A run of s of length zero is $\{s\}$. A run is bad if every string in the run is bad, i.e. if τ is disabled at all strings of the run.

The second conjunct ensures that no bad run of length greater than N exists in the system. The event of interest must be enabled (or otherwise must become ineligible to occur in the plant) after being disabled for N consecutive transitions[4] if the system is N-fair.

We did not bother to define fairness for the unbounded case. With bound removed, a language is unfair if it contains a bad run of length infinity; that is to say, when the language is generated by a finite automaton, the language is unfair if the automaton contains a loop of bad states, or which we refer to as a bad loop.

To justify our exclusion, we revisit the figure of Page 47, repeated below for convenience.

Observe that K is not fair since we have a bad (self-)loop. On the other hand, each K_i is fair as no bad loop exists in its generating automaton. Observe that:

$$K = \lim\{K_i\}_{i \in N} = \bigcup_{i \in N} K_i$$

[4] An event is disabled after taking a transition if it is disabled at the string that terminates with that transition.

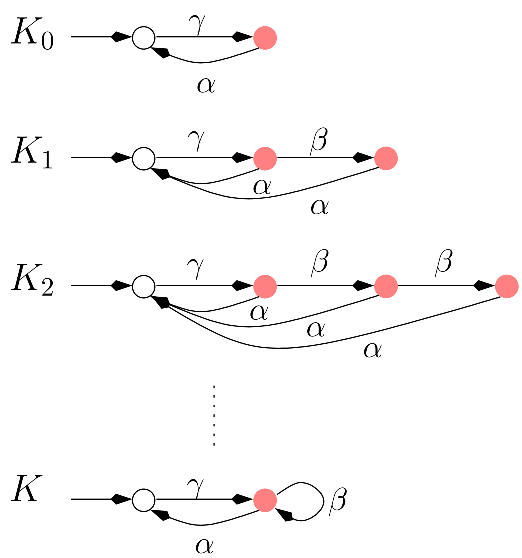


Figure 3.12: With no bound the new class is not closed under union.

Thus, in spite of strengthening the fairness condition in the hope of getting a subclass that is closed under union, the newly defined unbounded fairness is not preserved under countable union.

3.6.3 Alternative definition

We have the following equivalent statement of the fairness condition and hence the alternative definition below:

A controlled language K is N-fair, $N \in \mathbb{N}$, iff for all $s \in K_b$,

$$(\exists t \in \Sigma^*.\ |t| \leq N \wedge st \in K_g) \wedge (\forall t \in \Sigma^*.\ |t| = N \wedge st \in K \rightarrow \exists u \leq t.\ su \in K_g)$$

We prove that the two alternatives are equivalent.

Lemma 3.4 *The fairness condition above is equivalent to the condition used in the definition of the previous section.*

Proof. For convenience we repeat the two conditions below:

$$(\exists t \in \Sigma^*.\ t \neq \epsilon \wedge st \in K) \wedge (\forall t \in \Sigma^*.\ |t| = N \wedge st \in K \rightarrow \exists u \leq t.\ su \in K_g) \tag{3.3}$$

$$(\exists t \in \Sigma^*.\ |t| \leq N \wedge st \in K_g) \wedge (\forall t \in \Sigma^*.\ |t| = N \wedge st \in K \rightarrow \exists u \leq t.\ su \in K_g) \tag{3.4}$$

We notice that only the first conjuncts are different. It is immediate that when $s \in K_b$ we have (3.4) $\rightarrow$ (3.3) since:

$$(s \in K_b \wedge \exists t \in \Sigma^*.\ |t| \leq N \wedge st \in K_g) \rightarrow (\exists t \in \Sigma^*.\ t \neq \epsilon \wedge st \in K)$$

Note that $t = \epsilon$ cannot satisfy the existential statement in the antecedent.

Now suppose s is a bad string and (3.3) is true. In particular, for some $t_1 \in \Sigma^*$ we have $t_1 \neq \epsilon$ and $st_1 \in K$. Set $t := t_1$. If $|t| \geq N$ we quit the loop. Otherwise, i.e. when $|t| < N$, if $st \in K_g$ the proof is complete. Else, we have $st \in K_b$ and from (3.3) it follows that for some $t_2 \in \Sigma^*$ we have $t_2 \neq \epsilon$ and $st_1t_2 \in K$. Set $t := t_1t_2$ and repeat the above.

The procedure must terminate at most after N steps since

$$|t_1t_2 \cdots t_N| \geq N$$

After termination, write $t =: vw$ where $|v| = N$. Since K is prefix-closed we have that $sv \in K$, and therefore it follows from the second clause of (3.3) that

$$\exists u \leq v.\ su \in K_g$$

Since $|u| \leq |v| = N$ and $su \in K_g$, u satisfies the first clause of (3.4) and therefore we have:

$$\forall s \in K_b.\ (3.3) \rightarrow (3.4)$$

That completes the proof. ■

3.7 Comparing different measures of fairness

Aside from the issue of boundedness, up until now we have used three different criteria to measure fairness. First we have the familiar notion of coreachability, which is the only type of liveness studied in the traditional supervisory control theory. According to this methodology, a live behavior is specified by marking a subset of strings generated by the plant. The system is live if any string generated by the closed-loop system can be completed to a marked string. As seen before, on partitioning strings into two disjoint subsets "good" and "bad", this is equivalent to saying that in a fair system any bad string can be completed to a good one. There is no need to place a bound on the length of a bad run in this case: we proved that with no bound the property is preserved under arbitrary union. That property, once again, states that there is a completion to a good string from any bad string.

In the second type, we provided more flexibility by allowing a priority relation among events. Now not every completion to a good string is acceptable: that completion, called a savior, has to have a maximal priority. In passing we showed that the first type is a special case of the second when no priority is given.

The third type of fairness does not allow any bad loop, or in the bounded case, any bad run longer than a prescribed length. We showed that except for the first type, fairness is not robust under arbitrary union when no bound is specified. Even with bound, the second class is not closed under union. It is yet to be determined in Section 3.8 if a supremal element in the bounded case exists for the third type.

For the purpose of comparison here we list all types we have encountered so far:

Let $K \subseteq L \subseteq \Sigma^*$, P be a priority relation on events, and $N \in \mathbb{N}$ be a constant (bound). K is called N-fair of type indicated if for every $s \in K_b$ we have:

I. $\exists t.\ |t| \leq N \wedge st \in K_g$

II. $\exists t.\ |t| \leq N \wedge st \in K_g \wedge \big(\forall t' \in \Sigma^*.\ t \prec t' \rightarrow st' \notin K\big)$

III. $(\exists t.\ |t| \leq N \wedge st \in K_g) \wedge (\forall t \in \Sigma^*.\ |t| = N \wedge st \in K \rightarrow \exists u \leq t.\ su \in K_g)$

Graphically, for $N = 2$, sample members of $N-$fair languages of each type are shown in Figure 3.13.

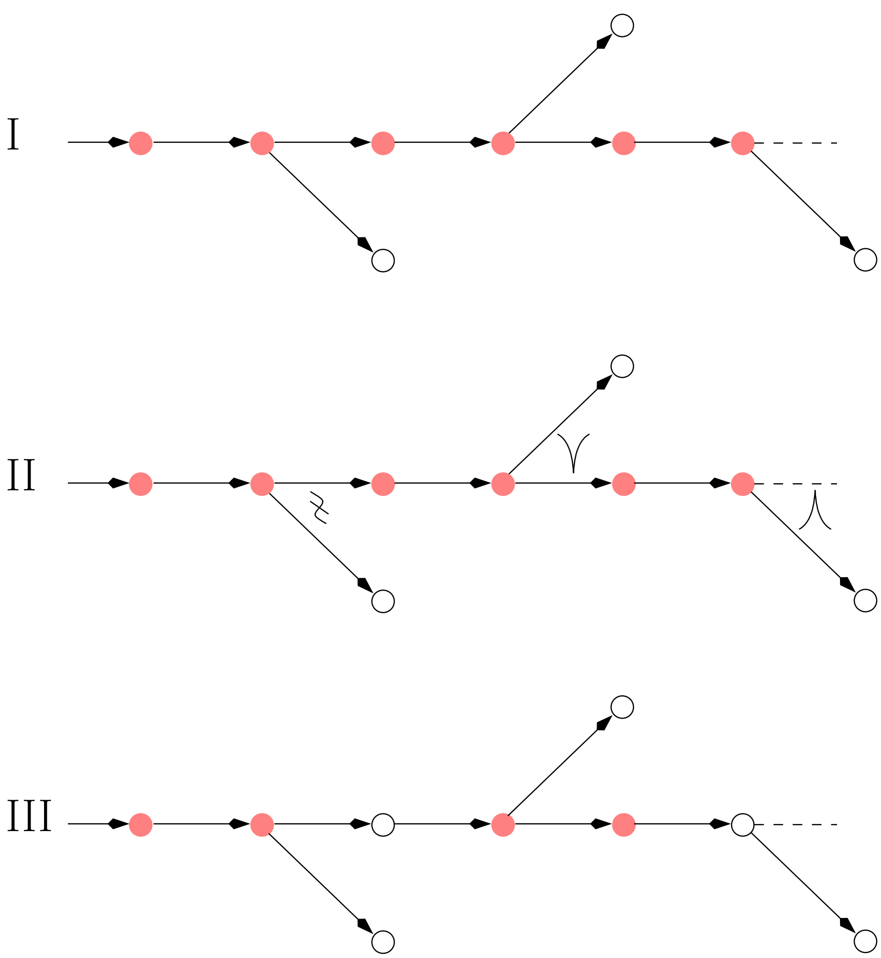


Figure 3.13: Fairness of different types ($N = 2$).

It is immediate from the definition that any language that is fair by (II) is also fair by (I), that is, denoting the three classes of fair languages by $\mathcal{F}_{\mathrm{I}}$, $\mathcal{F}_{\mathrm{II}}$, and $\mathcal{F}_{\mathrm{III}}$, respectively, we have:

$$\mathcal{F}_{\mathrm{II}} \subseteq \mathcal{F}_{\mathrm{I}}$$

In what follows we show that $\mathcal{F}_{\mathrm{III}} \subseteq \mathcal{F}_{\mathrm{II}}$.

3.7.1 Some results on maximal priority

The results presented here are used in the proof of the next section. They have rather clear intuitive meanings which for the most part follow from the fact that with a finite alphabet and a bound placed on the length of strings, we have a finite poset of strings which always has a maximal element. Nevertheless, for completeness we supply all proofs, especially since our irreflexive priority relation is not a partial order.

Our first result states that at every move we have an action of maximal priority to take.

Lemma 3.5 *Let P be an irreflexive, anti-symmetric and transitive relation on Σ, and $\emptyset \neq \mathcal{T} \subseteq \Sigma$. Then $\mathcal{T}$ has maxima, i.e.*

$$\exists \sigma \in \mathcal{T}.\ \forall \sigma' \in \mathcal{T}.\ \neg(\sigma, \sigma') \in P$$

Proof. By contradiction. Suppose:

$$\forall \sigma \in \mathcal{T}.\ \exists \sigma' \in \mathcal{T}.\ (\sigma, \sigma') \in P$$

Pick any $\sigma_1 \in \mathcal{T}$. From the above there must exist $\sigma_2 \in \mathcal{T}$ such that $(\sigma_1, \sigma_2) \in P$. If $\sigma_2 = \sigma_1$ quit the loop. Otherwise, since $\sigma_2 \in \mathcal{T}$ we must have a $\sigma_3 \in \mathcal{T}$ such that

$(\sigma_2, \sigma_3) \in P$. We continue this procedure n times and construct the sequence

$$(\sigma_1, \sigma_2), (\sigma_2, \sigma_3), \cdots, (\sigma_{n-1}, \sigma_n)$$

in P until $\sigma_n = \sigma_k$ for some $k < n$. This should eventually happen since we have a finite alphabet. From the transitivity of P it follows that $(\sigma_k, \sigma_k) \in P$, which contradicts the fact that P is irreflexive. ∎

The next definition extends the maximality of events to strings. More specifically, we identify maximal continuations of a string in K as those continuations at whose every prefix an event with maximal priority is taken out of possible choices for the next move.

Definition 3.3 *Let $s \in K$. t is said to be a maximal continuation of s in K iff*

$$st \in K \wedge (\forall t' \in \Sigma^*.\ t \prec t' \rightarrow st' \notin K)$$

Maximality on strings is of course induced by that on events. It is noteworthy that strings can now be compared based not only on their lengths, but also on their priorities. We reserve the word "maximal" to mean maximal priority[5], whereas "greatest" always refers to the length.

Since we mostly deal with bounded sequences, we strengthen the above condition by placing a bound on the length of a maximal continuation. In addition, we demand the supremality of length, that is, we stop before reaching the bound only if there is no choice for the next move in K.

Definition 3.4 *Let $s \in K$ and $N \in \mathbb{N}$ be a bound. t is a maximal continuation of s*

[5]Note that a highest priority may not exist.

in K of greatest length bounded by N iff

$$t \text{ is a maximal continuation of } s \text{ in } K \wedge |t| \leq N \wedge \big(|t| < N \rightarrow \forall \sigma \in \Sigma.\ st\sigma \notin K\big)$$

Using the result stated in the beginning of this section, one can constructively prove the existence of a t satisfying the above condition. Starting at $t = \epsilon$, inductively extend t with a maximum of the set:

$$\mathcal{T}_{st} = \{\sigma |\ st\sigma \in K\}$$

Terminate when $|t| = N$ or $\mathcal{T}_{st} = \emptyset$, whichever happens first. Then t will satisfy all clauses of the definition.

Our last result of this section, which will be used in the proof of the next, states that a prefix of a maximal string is also maximal.

Lemma 3.6 *Let $t_1 \leq t_2$. If t_2 is a maximal continuation of s in K, so is t_1.*

Proof. Let t_2 be a maximal continuation of s in K, i.e.:

$$st_2 \in K \wedge (\forall t' \in \Sigma^*.\ t_2 \prec t' \rightarrow st' \notin K)$$

Since K is closed, $st_2 \in K$ and $t_1 \leq t_2$ imply $st_1 \in K$. Let $t' \in \Sigma^*$ be such that $t_1 \prec t'$, that is:

$$\exists u, \sigma, \sigma'.\ u\sigma \leq t_1 \wedge u\sigma' \leq t' \wedge \sigma \prec \sigma'$$

For any $u \in \Sigma^*$, it follows from $u\sigma \leq t_1$ and $t_1 \leq t_2$ that $u\sigma \leq t_2$, therefore the above implies:

$$\exists u, \sigma, \sigma'.\ u\sigma \leq t_2 \wedge u\sigma' \leq t' \wedge \sigma \prec \sigma'$$

i.e. $t_2 \prec t'$. But since t_2 is maximal this implies $st' \notin K$. Thus, t_1 is a maximal continuation of s in K. ■

3.7.2 The new type is the most restrictive

In this section we show that a language is fair by (II) if it is fair by (III).

Theorem 3.4 *Let K be fair by (III). Then it is fair by (II), i.e. $\mathcal{F}_{III} \subseteq \mathcal{F}_{II}$.*

Proof. By contradiction. Suppose K is fair by (III) but not by (II), i.e. there exists an $s \in K_b$ such that:

$$\neg\exists t.\ |t| \leq N \wedge st \in K_g \wedge \left(\forall t' \in \Sigma^*.\ t \prec t' \rightarrow st' \notin K\right)$$

which can be restated equivalently as:

$$\forall t.\ |t| \leq N \wedge \left(\forall t' \in \Sigma^*.\ t \prec t' \rightarrow st' \notin K\right) \rightarrow st \notin K_g$$

Let t be a maximal continuation of s in K of greatest length bounded by N. Since $st \in K$ it follows from the above implication that $st \in K - K_g = K_b$.

We argue that $|t| = N$. Otherwise, since K is fair by (III) it follows from the first clause of (3.3), Page 58 that:

$$\exists v.\ v \neq \epsilon \wedge stv \in K$$

which contradicts the fact that t has greatest length bounded by N unless $|t| = N$. Since

$$|t| = N \wedge st \in K$$

it follows from the second clause of the definition that:

$$\exists u \leq t.\ su \in K_g.$$

Since t is a maximal continuation of s in K, so is u, that is:

$$\forall t' \in \Sigma^*.\ u \prec t' \to st' \notin K$$

Also, we have $|u| \leq |t| = N$. Therefore, from the contradiction assumption it follows that $su \notin K_g$, which is a contradiction. ■

Thus we have shown that:

$$\mathcal{F}_{\mathrm{III}} \subseteq \mathcal{F}_{\mathrm{II}} \subseteq \mathcal{F}_{\mathrm{I}}$$

Figure 3.14 shows four member languages from the disjoint classes formed by the above chain. We assume that $N = 2$ and that $\beta \prec \gamma$. Fairness is measured with respect to a fourth event, say τ. When τ is disabled at a state it is colored grey. For each case an open-loop plant is implicit which can be deduced from the pattern of good-bad strings. For instance, since $s = \alpha\beta$ is colored grey in (a), the open-loop plant in (a) can perform a τ transition after $s = \alpha\beta$. On the other hand, $s = \alpha\beta$ is good in (b). Since τ is not a choice in the closed-loop system we conclude that τ is ineligible to occur at $s = \alpha\beta$ in the open-loop system of[6] (b).

In all cases we focus on leading $s = \epsilon$ to safety, as informally speaking, it is the most expensive to save among all strings.

In case (a), all continuations to good strings have length three, and therefore (a) is not 2-fair by any standard. In (b) the string s is led to safety by $t = \alpha\beta$ which

[6]In some cases, we are only interested in the partition of strings into good and bad; as soon as that partition is given, the need for the above analysis disappears. In fact, we need the open-loop system to find which strings are good and which ones are bad. In the above example this information is already given.

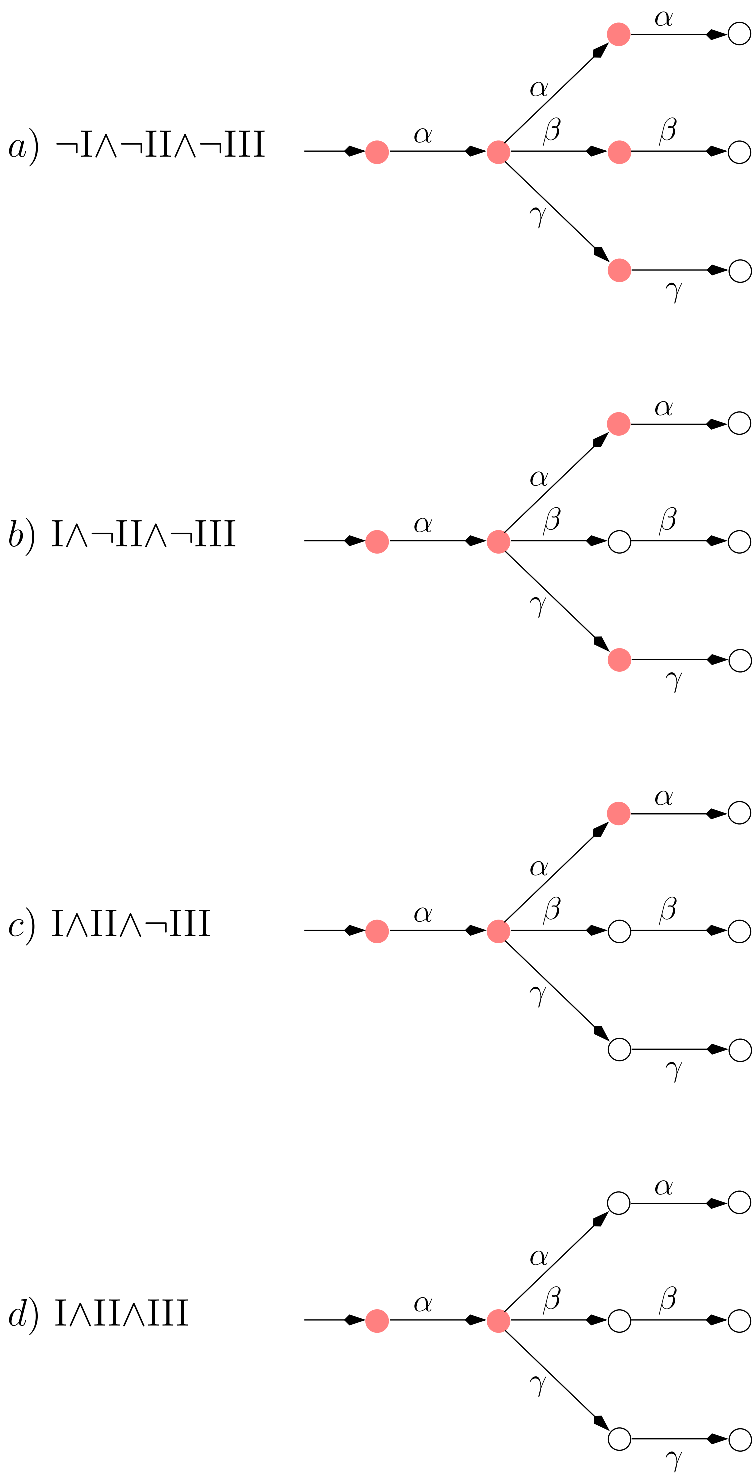


Figure 3.14: Different shades of fairness.

has length two; therefore, (b) is 2-fair by (I). On the other hand, (b) is not 2-fair by (II) as β in the above continuation is overtaken by γ at the prefix $u = \alpha$. Neither is it 2-fair by (III) since there are bad runs starting at s of length more than two, for instance:

$$\{\epsilon, \alpha, \alpha^2\}.$$

(c), on the other hand, is 2-fair by (II) since the continuation $t = \alpha\gamma$ which leads s to safety has maximal priority. (c) remains 2-unfair by (III) exactly for the same reason as in (b). Finally, (d) is 2-fair by (III) since there is no bad run of s—and hence from any other string—of length greater than two.

It is evidently clear that the condition becomes stronger as one goes down the ladder, which is reflected by having to mark more strings as good in order to be able to meet the demands of the new class.

3.8 Optimally fair

In this section we prove that strengthening the fairness condition has paid off and a largest fair sublanguage of a given language exists in the new class.

Define the class of closed sublanguages of E by $\mathcal{P}(E)$:

$$\mathcal{P}(E) = \{K \subseteq E | \ K = \overline{K}\}$$

For the ease of argument the fairness condition is broken into two separate subconditions required of a fair sublanguage K. First, no bad string of K can be terminal, denoted by $x(K)$, where x is a predicate $x : \mathcal{P}(E) \to \{\bot, \top\}$ defined as:

$$x(K) := \Big\{\forall s.\ s \in K_b \to \exists t.\ t \neq \epsilon \wedge st \in K\Big\}$$

And second, no bad run of length greater than N can exist in K, denoted by $y(K)$, where y is a predicate on languages $y : \mathcal{P}(E) \to \{\bot, \top\}$ defined as:

$$y(K) := \big\{\forall s.\ s \in K_b \to (\forall t.\ |t| = N \wedge st \in K \to \exists u \le t.\ su \in K_g)\big\}$$

Let p, q and r be any predicates (on strings). We have:

$$\begin{aligned} & (\forall s.\ p(s) \to q(s)) \wedge (\forall s.\ p(s) \to r(s)) \\ = \quad & \forall s.\ (p(s) \to q(s)) \wedge (p(s) \to r(s)) \\ = \quad & \forall s.\ p(s) \to (q(s) \wedge r(s)) \end{aligned}$$

Letting:

$$\begin{aligned} p(s) &:= s \in K_b \\ q(s) &:= \exists t.\ t \neq \epsilon \wedge st \in K \\ r(s) &:= \forall t.\ |t| = N \wedge st \in K \to \exists u \le t.\ su \in K_g \end{aligned}$$

it follows that a language K is fair if and only if $x(K) \wedge y(K)$.

Let $\mathcal{F}(E)$ denote the subclass of fair and closed sublanguages of E, that is:

$$\mathcal{F}(E) = \big\{K \subseteq E |\ K = \overline{K} \wedge x(K) \wedge y(K)\big\}$$

Note that the subscript III is not used anymore, and unless stated otherwise when talking of fairness this latest version is meant.

We show that the supremum of $\mathcal{F}(E)$ exists and is in $\mathcal{F}(E)$. First we prove that x and y, which are properties of languages, are preserved under arbitrary union.

Lemma 3.7 *x and y are preserved under arbitrary union, i.e. for any $\mathcal{K} \subseteq \mathcal{P}(E)$*

we have:

1. $\big(\forall K \in \mathcal{K}.\ x(K)\big) \rightarrow x(\cup\mathcal{K})$
2. $\big(\forall K \in \mathcal{K}.\ y(K)\big) \rightarrow y(\cup\mathcal{K})$

Proof. We denote $\mathcal{K}^{\uparrow} := \cup\mathcal{K}$.

1. Suppose $\mathcal{K}$ is a subclass whose every member satisfies x. Let $s \in \mathcal{K}^{\uparrow}{}_b$. It follows that $s \in K_b$ for some $K \in \mathcal{K}$. Since $x(K)$ is true, $s \in K_b$ implies $st \in K$ for some $t \neq \epsilon$. But $st \in K$ implies $st \in \mathcal{K}^{\uparrow}$, i.e. $x(\mathcal{K}^{\uparrow})$.

2. Suppose $\mathcal{K}$ is a subclass whose every member satisfies y. Let $s \in \mathcal{K}^{\uparrow}{}_b$ and t be a string such that $|t| = N$ and $st \in \mathcal{K}^{\uparrow}$. It follows that $st \in K$ for some $K \in \mathcal{K}$. Since K is prefix-closed we have $s \in K$, implying that $s \in K_b$. Since $y(K)$ is true it follows that:
$$\exists u \leq t.\ su \in K_g$$
which implies:
$$\exists u \leq t.\ su \in \mathcal{K}^{\uparrow}{}_g$$
since $\mathcal{K}^{\uparrow}{}_g = \cup\{K_g |\ K \in \mathcal{K}\}$. Therefore $y(\mathcal{K})$ is true. ∎

From the above result it follows that the largest fair sublanguage of a given language exists. In other words, the supremum of $\mathcal{F}(E)$ is in $\mathcal{F}(E)$.

Theorem 3.5 *Let E be a language. We have:*
$$\mathcal{F}(E)^{\uparrow} \in \mathcal{F}(E)$$

Proof. Since every member of $\mathcal{F}(E)$ satisfies x and y, it follows that $x\big(\mathcal{F}(E)^{\uparrow}\big)$ and

$y\big(\mathcal{F}(E)^{\uparrow}\big)$ are true. We also have $\mathcal{F}(E)^{\uparrow} \subseteq E$ and:

$$\overline{\mathcal{F}(E)^{\uparrow}} = \overline{\cup\big\{K|\ K \in \mathcal{F}(E)\big\}} = \cup\big\{\overline{K}|\ K \in \mathcal{F}(E)\big\} = \cup\big\{K|\ K \in \mathcal{F}(E)\big\} = \mathcal{F}(E)^{\uparrow}$$

Therefore $\mathcal{F}(E)^{\uparrow} \in \mathcal{F}(E)$, i.e. $\mathcal{F}(E)^{\uparrow}$ is the largest fair sublanguage of E. ■

The existence of $\Psi_x(M)$ and $\Psi_y(M)$ defined below follows from the previous lemma. They denote the largest prefix-closed sublanguages of E that satisfy the x and y properties, respectively.

Definition 3.5 *, Let $M \in \mathcal{P}(E)$ be a language. Define:*

$$\begin{aligned} \Psi_x(M) &:= \big\{K \subseteq M|\ K = \overline{K} \wedge x(K)\big\}^{\uparrow} \\ \Psi_y(M) &:= \big\{K \subseteq M|\ K = \overline{K} \wedge y(K)\big\}^{\uparrow} \end{aligned}$$

3.8.1 Computing a fair sublanguage

In this section we define a function Φ on $\mathcal{P}(E)$ whose fixed-points coincide with the fair sublanguages of E, i.e.:

$$\text{fix}\Phi = \mathcal{F}(E)$$

Therefore, the largest fixed-point of Φ, denoted by $\nu\Phi$, exists and is equal to:

$$\nu\Phi = \mathcal{F}(E)^{\uparrow}$$

We show that Φ is order-preserving and therefore a repeated application of Φ on E, if it ever terminates, will yield the largest fixed-point of Φ. That provides a procedure for computing the largest fair sublanguage of a given language.

Definition 3.6 *Let E be a language. Define $\Phi : \mathcal{P}(E) \to \mathcal{P}(E)$ to be:*

$$\Phi(M) := \Psi_y \circ \Psi_x(M); \quad M \in \mathcal{P}(E)$$

First we show that Φ offers a fixed-point characterization of fair sublanguages of E. For ease of notation we may drop E in our references to $\mathcal{P}$ and $\mathcal{F}$.

Lemma 3.8 *We have fix$\Phi = \mathcal{F}$.*

Proof. **($\subseteq$)** Let M be a fixed-point of Φ, i.e.:

$$\Psi_y\big(\Psi_x(M)\big) = M$$

First we argue that M is also a fixed-point of Ψ_x and Ψ_y. Since Ψ_x and Ψ_y are contractive maps we have:

$$\Psi_x(M) \subseteq M$$
$$M = \Psi_y\big(\Psi_x(M)\big) \subseteq \Psi_x(M)$$

Therefore $\Psi_x(M) = M$, implying that $\Psi_y(M) = M$. Since M is a fixed-point of Ψ_x and Ψ_y, it follows that $x(M)$ and $y(M)$ are true, which together with $M = \overline{M}$ imply that $M \in \mathcal{F}(E)$.

($\supseteq$) Let M be a fair sublanguage of E. Thus M is prefix-closed, and $x(M)$ and $y(M)$ are true. This implies that M is a fixed-point of Ψ_x and Ψ_y. We have:

$$\Phi(M) = \Psi_y\big(\Psi_x(M)\big) = \Psi_y(M) = M$$

i.e. M is a fixed-point of Φ. ■

We next show that Φ is order-preserving.

Lemma 3.9 *Let $M_1, M_2 \in \mathcal{P}$ be such that $M_1 \subseteq M_2$. We have $\Psi_x(M_1) \subseteq \Psi_x(M_2)$ and $\Psi_y(M_1) \subseteq \Psi_y(M_2)$ and therefore:*

$$\Phi(M_1) = \Psi_y\big(\Psi_x(M_1)\big) \subseteq \Psi_y\big(\Psi_x(M_2)\big) = \Phi(M_2).$$

Proof. Let $M_1 \subseteq M_2$. We have:

$$\Psi_x(M_1) \subseteq M_1 \subseteq M_2.$$

Moreover, since $\Psi_x(M_1) = \overline{\Psi_x(M_1)}$ and $x(\Psi_x(M_1))$ is true, it follows that:

$$\Psi_x(M_1) \in \big\{K \subseteq M_2 |\ K = \overline{K} \wedge x(K)\big\}$$

Thus we have:

$$\Psi_x(M_1) \subseteq \big\{K \subseteq M_2 |\ K = \overline{K} \wedge x(K)\big\}^{\uparrow} = \Psi_x(M_2).$$

The proof for Ψ_y is similar. ■

Finally, we prove that repeated application of an order-preserving map Φ on E, if it ever terminates, will produce the largest fixed-point of Φ.

Lemma 3.10 *Let $\Phi : \mathcal{P} \to \mathcal{P}$ be an order-preserving map and the sequence $\{\Phi^i(E)\}_{i\in\mathbb{N}}$ terminate at $i = i^*$, i.e. $\Phi^{i^*+1}(E) = \Phi^{i^*}(E)$. Then Φ has a largest fixed-point and $\nu\Phi = \Phi^{i^*}(E)$.*

Proof. We have:

$$\Phi\Phi^{i^*}(E) = \Phi^{i^*}(E)$$

i.e. $\Phi^{i^*}(E)$ is a fixed-point of Φ. It remains to show that if K is a fixed-point of Φ then we have $K \subseteq \Phi^{i^*}(E)$.

Since $K \subseteq E$, applying Φ to both sides of the inequality i^* times will yield:

$$K = \Phi^{i^*}(K) \subseteq \Phi^{i^*}(E)$$

That completes the proof. ■

The previous results are summed up in the next theorem.

Theorem 3.6 *Let $\Phi = \Psi_y \circ \Psi_x$ and the sequence $\{\Phi^i(E)\}_{i\in\mathbb{N}}$ terminate at $i = i^*$, i.e. $\Phi^{i^*+1}(E) = \Phi^{i^*}(E)$. Then the largest fair sublanguage of E is equal to $\Phi^{i^*}(E)$.*

Proof. We have:

$$\Phi^{i^*}(E) = \nu\Phi = \mathcal{F}^{\uparrow}$$

■

The map Φ attempts to make a language fair first by excluding those bad strings that are terminal, an action carried out by Ψ_x, followed by the removal of all bad runs longer than a prescribed length, or equivalently, an application of Ψ_y.

Disabling an outgoing transition labeled with τ will make a formerly good string bad. Since each application of Ψ_x involves disabling new transitions, we may have to apply Ψ_y once again if a transition labeled with τ is disabled. Thus, removing a bad terminal string may result in creating a new bad run longer than N. Similarly, removing a bad run of length greater than N may make a bad string terminal, which necessitates another application of Ψ_x. The following example illustrates the iterative nature of fixpoint computation for Φ.

Example 8 Consider the language E shown in Figure 3.15. The unspecified labels are known not to be equal to τ. We would like to compute its largest fair sublanguage by repeated application of Φ, where N is chosen to be equal to one.

The first application of Ψ_y removes the only bad run of length greater than one

$$\Phi^0(E) = E$$

$$\Psi_x(E)$$

$$\Psi_y \circ \Psi_x(E) = \Phi(E)$$

$$\rightarrow \Psi_x\big(\Phi(E)\big)$$

$$\Psi_y \circ \Psi_x\big(\Phi(E)\big) = \Phi^2(E)$$

$$\Psi_x\big(\Phi^2(E)\big)$$

$$\Psi_y \circ \Psi_x\big(\Phi^2(E)\big) = \Phi^3(E)$$

$$\Psi_x\big(\Phi^n(E)\big)$$

$$\Psi_y \circ \Psi_x\big(\Phi^n(E)\big) = \Phi^{n+1}(E) \qquad n \geq 3$$

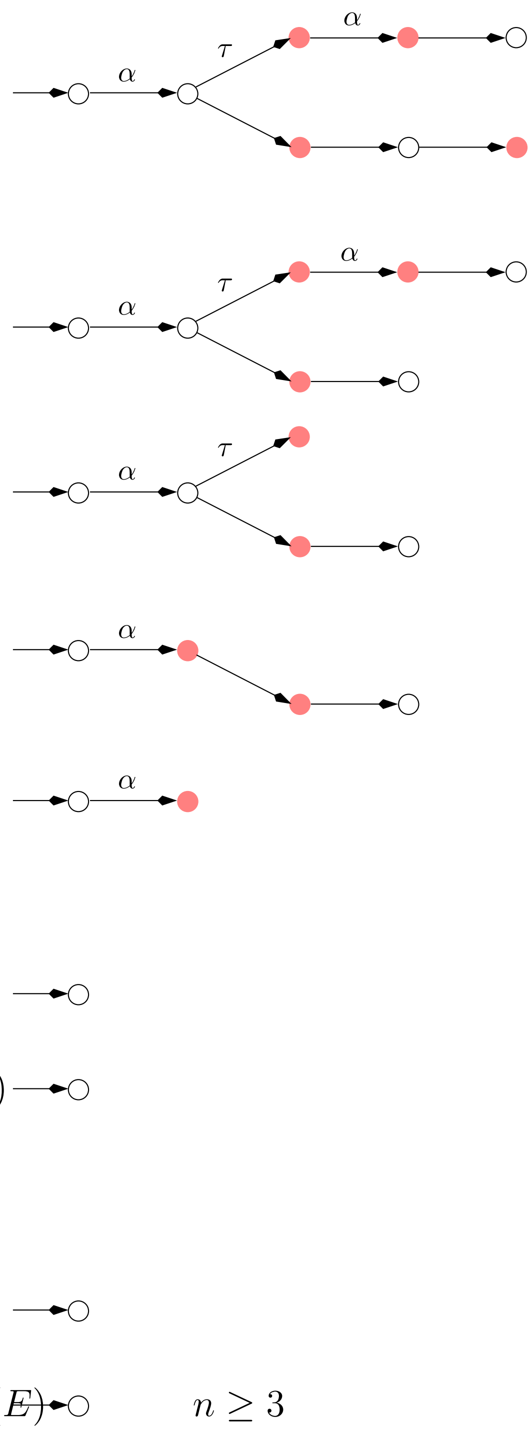


Figure 3.15: Iterative nature of fixpoint computation for Φ.

from $\Psi_x(E)$. The run in question is $\{\alpha\tau, \alpha\tau\alpha\}$, which is made acceptable by removing $\alpha\tau\alpha$ and all its continuations from $\Psi_y \circ \Psi_x(E)$, so that the length requirement is met. But this makes the remaining bad string $\alpha\tau$ terminal and therefore another application of Ψ_x becomes necessary, which simply removes the bad, terminal string $\alpha\tau$ from the closed-loop language.

The result, $\Psi_x\big(\Phi(E)\big)$, is marked by an arrow to highlight a change in the partition of strings into good and bad. The string $s = \alpha$, formerly a good string, has now become bad because the latest application of Ψ_x has disabled τ at $s = \alpha$. Matters could get even worse when, as in the example, the new bad string is preceded and/or followed by a bad run, and the accumulated length exceeds N. A bad run of length *greater* than N is born, which is a fairness violation and necessitates another application of Ψ_y.

The new bad run is eliminated by yet another application of Ψ_y, leaving behind a new bad, terminal string (namely, $s = \alpha$), which has in turn been taken care of by Ψ_x. Subsequent applications of Ψ_x and Ψ_y do not affect the language and the procedure terminates after three iterations. □

In Appendix B we explain how to implement Ψ_x and Ψ_y on a finite automaton.

Chapter 4

Example

4.1 Introduction

We recall from the cat and mouse example that C2M1 was regarded as unfair because there was the possibility of the two cats teaming up against the mouse and preventing his entering the middle room forever. Unlike C1M1, the cats have this liberty in C2M1 even when no individual is allowed to occupy the middle room indefinitely. They simply abuse a loophole in the law governing the system, which requires good behavior of each individual: an *individual* cannot occupy the middle room indefinitely, but a *species* can. Thus, the mouse is denied access indefinitely if each cat stays in the room for an infinite sequence of finite, overlapping time intervals; that is, the union of the two sequences covers the entire time-line.

C2M1 is typical among those systems which show the type of unfair behavior that we would like to study. A system has the potential to behave unfairly if:

1. It is made up of at least two agents, better referred to as teams, competing to access a shared resource, and

2. At least one team has two players or more.

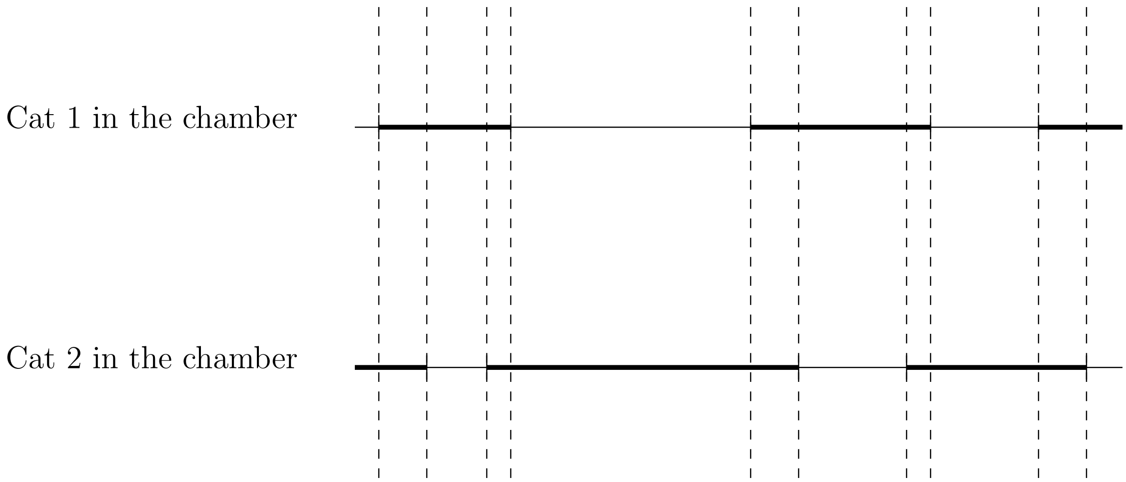


Figure 4.1: Each cat is in the room only for finite intervals, but because of the overlaps the cats are always in the room.

We assume that each player is required to release the resource eventually. Despite this, the possibility of unfairness becomes a reality when one tries to implement mutual exclusion in a minimally restrictive fashion. Then, two players from a team can join forces by alternately accessing the resource in an overlapping fashion, thereby robbing other teams of the chance of ever accessing the resource.

Many familiar unfair cases in practice are unfair in the sense just described. Here we mention two examples, adapted from [MK99].

The first example involves a one-lane bridge or tunnel on a road. Traffic lights are installed at each end to ensure that at any given time the traffic over the bridge flows in one direction only. For the sake of simplicity we assume that the road is running from north to south; then the southbound and the northbound traffic flows each constitute a team, with players being the cars. The assumption that no individual is allowed to use the resource indefinitely translates into not allowing a car to break down—or stop for any other reason—indefinitely when crossing the bridge. Despite this, the situation is ripe for manifesting unfair behavior, since we require that the bridge be used exclusively in one direction, i.e. we require mutual exclusion. In fact, the northbound traffic could be denied access to the bridge forever if the southbound traffic congests the bridge, that is, a car traveling southbound enters the bridge *before* all cars ahead leave the bridge.

Congestion can be modeled efficiently using two cars cruising in a circular fashion, i.e. they are available to re-enter the bridge soon after they leave. In this case, as in Cat and Mouse, a car can be modeled by a two-state automaton: it is at rest when the car is driving to or from the bridge, and it is at its critical state when the car is traversing the bridge.

The second example quoted in [MK99] is the Readers-Writers problem. There are two kinds (*teams*) of processes (*players*), with each process trying to access a shared database (*resource*). Readers can read the database simultaneously, but each writer requires exclusive access when writing the database. Again, a process can be modeled by a two-state automaton: a transition is made to the critical state when the database is accessed by a process. The system specification requires that a writer move to its critical state only when all other processes are at rest.

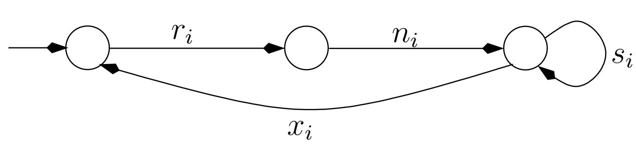
With two readers only, the system described above can run into fairness problems: the readers can read the database intermittently with overlaps (see the pattern of Figure 4.1), never releasing the database to be written to by the writers.

Back to the Cat and Mouse example, we observe that in terms of the number of states it has the minimal requirement to manifest the type of unfairness that we are interested in. According to the rules of Page 76, unfair behavior is plausible when there are at least two teams competing for a shared resource, with at least one team being made up of more than one player. Thus we will need a minimum of two cats and a mouse (or two mice and a cat) in order to generate unfair behavior. But we recall that each process requires at least two states to model whether the resource is in use. In addition, we need another state to register requests if we were to make the system fair by, for example, introducing a First-In, First-Out (FIFO) queue or a "turn" variable. This will give rise to a minimum total number of $3 \times 3 \times 3 = 27$ states required of a system of the type described above to generate unfair behavior. Unfortunately, this large state size makes graphical representations incomprehensible. To overcome this problem, in the next section we propose a simpler system with a smaller number of states which still has all the essentials for generating unfair behavior.

4.2 Example

A system is made up of two components. Each component can be in three different states: rest, request, and critical. Making a transition is possible from one state to its "successor": from rest to request labeled with 'r', from request to critical labeled with 'n', and finally from critical to rest labeled with 'x'. According to the scheme of the previous section we have two teams of one player and therefore we are one player short of generating unfair behavior. To mimic the effect that adding an extra player to each side would have, we add selfloop transitions 's' at critical states.

$\mathbf{A_i}$, $i = 1, 2$



r : request

n : enter

s : stay

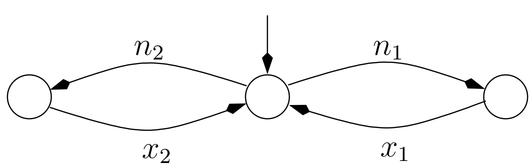
x : exit

Figure 4.2: Plant.

The assumption that each individual is well behaved states that each process must take action eventually, and cannot remain idle at a state indefinitely. But this does not necessitate a state change when a process is at its critical state, as it can instead choose to stay there by taking an 's' transition. In effect, an 's' transition lets a process act without losing hold of the resource, which is exactly what is needed to make the other process starve when the resource is shared and has to be accessed exclusively.

Although our focus is not on controllability here, we observe that a "solution" is a *controllable* sublanguage of the plant and specification, and therefore we need to make sure that in a solution only controllable events are disabled. For that to happen, we need to declare at least 'n' and 's' as controllable. One may question the controllability of 's', by arguing that although the access to a resource may be denied, it may not be up to us to decide how long a process can hold the resource. In response we recall that an 's' transition represents a *second* teammate (process) accessing the resource, thereby giving the first process a chance to take a break and release the resource for a while. Thus we feel justified in regarding 's' as controllable since the action of a second teammate accessing the resource can be controlled.

We consider the plant's operation legal if $\mathbf{A_1}$ and $\mathbf{A_2}$ are in their critical states exclusively, or in other words if they access the shared resource in a mutually exclusive fashion. This requirement is specified by the automaton of Figure 4.3.



Self-loop r_i and s_i, $i = 1, 2$.

Figure 4.3: Specification.

Since the specification is controllable with respect to the plant, it can also serve as a supervisor. The supervised system is shown in Figure 4.4. To help understand the graphical representation, the 'x' transitions are drawn in a third dimension with dashed lines, and may be dropped altogether in future representations. Labels are indicated only for transitions in the first row and column; parallel transitions in other rows/columns will carry the same label.

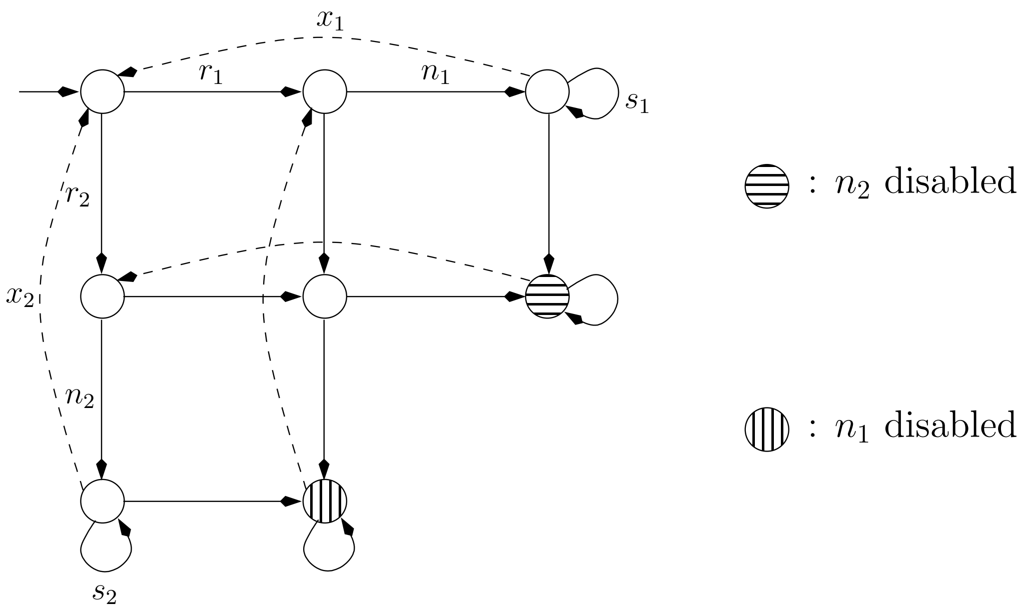


Figure 4.4: Closed-loop under mutual exclusion.

In addition to having mutual exclusion implemented, we would like a closed-loop system that is fair to the 'n' transitions. Recall that a transition which carries the label 'n_i' represents the process number i acquiring the shared resource. A closed-loop system is fair to 'n_1' and 'n_2' iff it is fair to 'n_1' and it is fair to 'n_2'. Clearly the closed-loop system of Figure 4.4 is not N-fair for any finite N since 'n_2' is disabled at the state located at row 2, column 3, and remains disabled while the system stays there taking s_1 transitions. Intuitively, the closed-loop system is unfair to 'n_2' since the supervisor indefinitely neglects a request made by process number 2, while process number 1 is at its critical state taking 's_1' transitions[1]. A similar observation can be made about 'n_1' being disabled at the state located at row 3, column 2.

One traditional fix to the problem is to add a First-In-First-Out queue to the closed-loop system. Using a FIFO queue one registers requests in the order that they

[1]Observe that in this case process number 1 is obliged to take 's_1' transitions every now and then, as otherwise it would violate the laws governing the system's operation.

arrive and service is granted in the order of requests. A FIFO queue for the system is represented by the automaton of Figure 4.5.

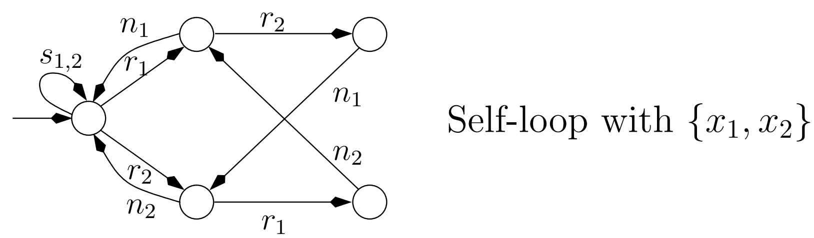


Figure 4.5: A FIFO queue.

Under the queue, a process is required to release the resource (by disabling the 's' transition) as soon as the other process makes a request. The closed-loop system with the queue added is shown in Figure 4.6. According to our definition of fairness, the closed-loop system is 2-fair (to 'n_1' and 'n_2') since, for instance, 'n_1' is disabled in a run of length at most 2. The queue may not provide an ideal solution to, for example, the bridge problem. It may be a more efficient use of time to allow cars to enter the bridge in batches of 10, for instance, and then give a turn to the traffic waiting on the other side, if any. In its simplest form, we send in the cars in batches of one, and use a Boolean "turn variable" to switch turns. The turn variable is represented by the automaton of Figure 4.7. Under the turn variable the system can operate in two different modes: process one is given access to the shared resource (the left-hand side square) and process 2 is given access to the shared resource (the right-hand side square). The turn is initially given to the first process. For convenience we let $\bar{i} := 2$ (1) when $i = 1$ (2, respectively). When the system is in mode i, process i is given preference to access the resource. However, both processes can access and hold the resource as many times as they wish if no request has yet been made by the other. The resource is granted to process i and an 'n_i' transition takes place as soon as it is requested by both processes. This will cause a switch to mode $\bar{i}$ of operation. We use bold arrows to indicate those 'n' transitions that cause a switch. The system under

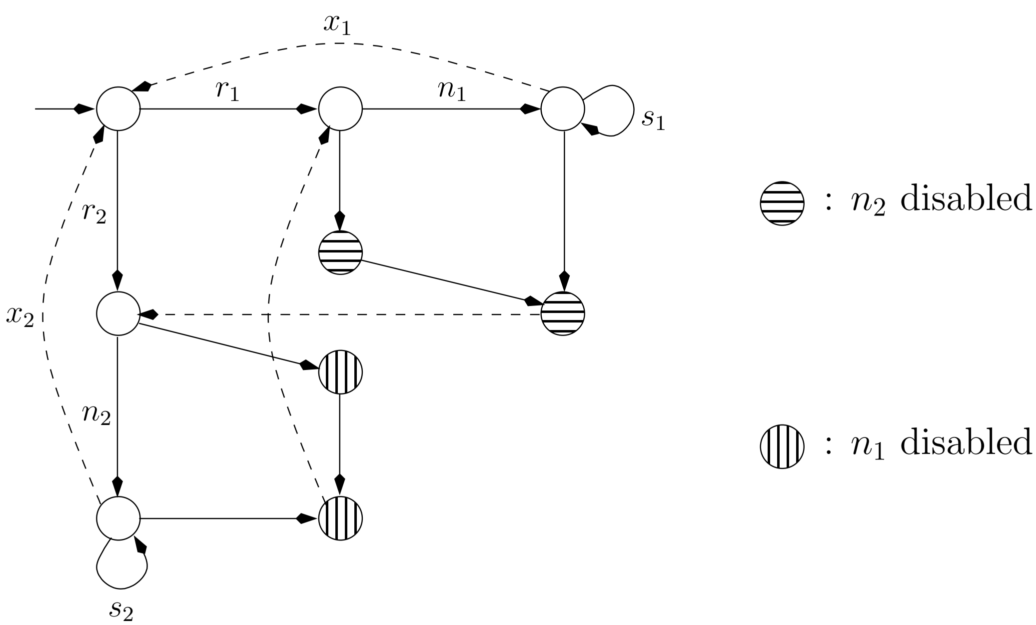


Figure 4.6: Closed-loop system, with the queue added.

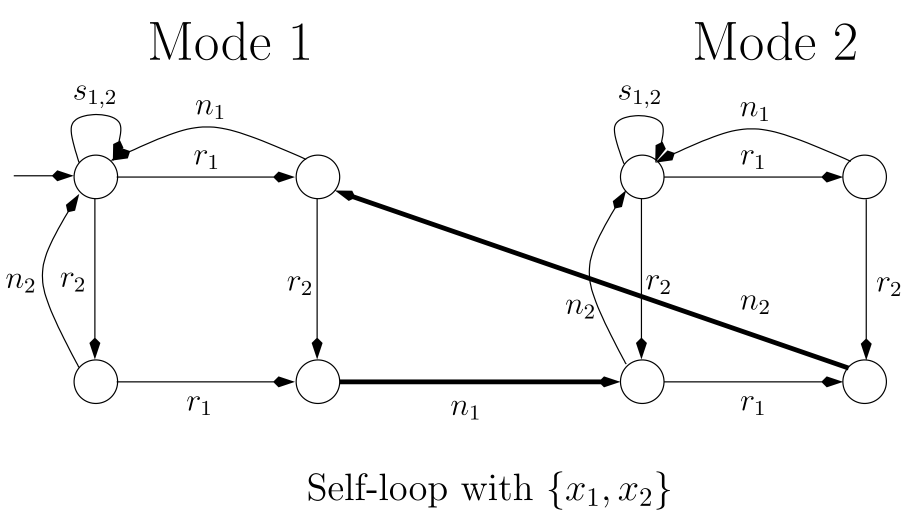


Figure 4.7: Automaton implementation of a turn variable.

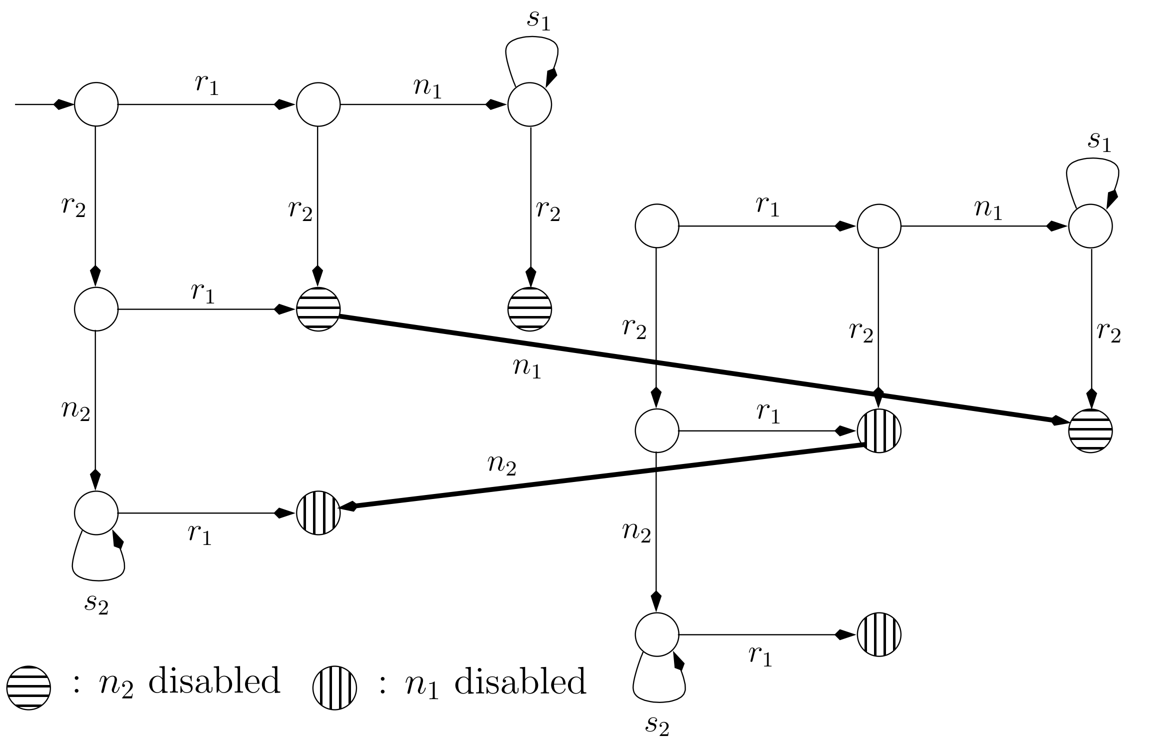


Figure 4.8: Closed-loop system, with the turn variable implemented. Note that to avoid cluttering the picture the 'x' transitions are not shown.

control is shown in Figure 4.8, where the 'x' transitions are not shown for clarity.

With a boolean turn variable implemented in the model, a process is required to release the resource (by disabling the option to stay) as soon as the other process makes a request. The difference from a queue is that the order of requests is irrelevant: the process whose turn is next will be serviced next. We observe that according to our definition of fairness, the closed-loop system is 2-fair (to 'n_1' and 'n_2') since, for instance, 'n_1' is disabled in a run of length at most 2.

Finally, we compute the supremal 2-fair sublanguage of the closed-loop system of Figure 4.4. There are two bad loops that need to be taken care of. The first one is the self-loop at state (row 2, column 3), where 'n_2' is disabled.

To make the system (optimally) 2-fair to 'n_2' the procedure of Appendix B is applied: the state (row 2, column 3) is augmented with a counter variable which

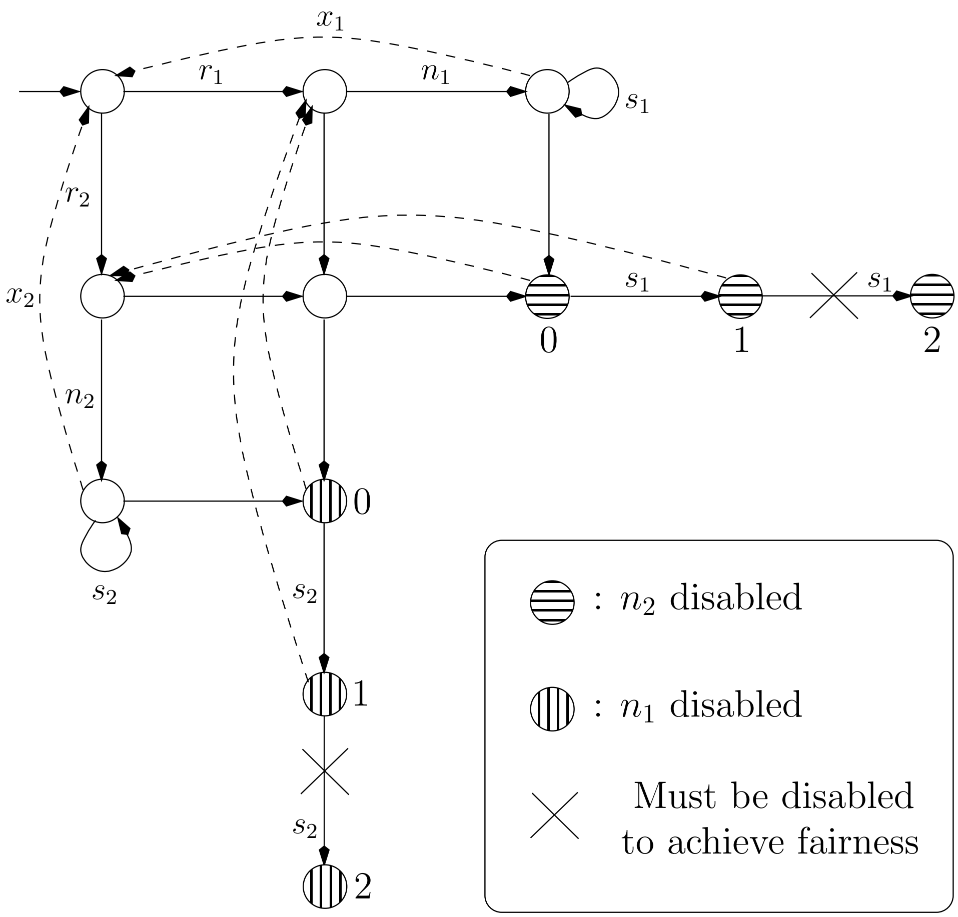


Figure 4.9: Supremal 2-fair sublanguage of Figure 4.4.

counts up to 2. Thus, the state (row 2, column 3) is replaced by three siblings with counter variables equal to 0, 1, and 2. As before, an 'x_1' transition takes all the new siblings to state (row 2, column 1). When the state at (row 2, column 3) is reached by 'r_2' or 'n_1', the counter variable is set to zero. The counter is incremented by one, up to a maximum of 2, each time an s_1 transition is taken. To make the system 2-fair, the sibling with counter variable equal to 2 is cut. The procedure ends since the operation described does not leave a new terminal state behind.

The optimal 2-fair solution is shown in Figure 4.9. One can verify using TCT that its behavior is larger than that generated under the queue (Figure 4.6), or under the turn variable (Figure 4.8), while the reverse cannot be said: The string $s = r_1 r_2 n_2$, generated by the optimal automaton, can be generated under neither the queue nor the turn variable.

Chapter 5

Dining Philosophers

Another textbook example, which we shall discuss in this chapter, is the Dining Philosophers problem (DPP). Introduced by Dijkstra [Dij71], the Dining Philosophers Problem is a benchmark resource allocation problem for comparing the expressive power of different concurrent programming languages [Rin86]. An informal description of the problem is as follows. A finite number of resources (forks) are shared by a finite number of processes (philosophers). The objective is to assign the resources to the processes in such a way that no two processes have simultaneous access to the same resource, and each process has access infinitely often to all resources that it will require for performing its job (dining). Observe that the first requirement is safety while the second one is liveness.

We study the problem from a control-theoretic point of view. First we design an optimal supervisor to achieve the safety requirement. It turns out that such optimal supervision leaves the philosophers with too much freedom, to the extent that it will be possible for the neighbors of a philosopher to collaborate against him and make him starve. We demonstrate how the techniques of this work can be utilized to remedy the situation by systematically constructing a design that meets the liveness condition.

The rest of this chapter is organized as follows. The problem is formally stated in Section 5.1, in which the plant and the requirement specification are modeled by finite automata. Section 5.2 discusses conditions under which a decentralized fair solution is possible to a problem, and we verify that those conditions are met when DP is solved using local queues. In section 5.3 the definition of fairness is slightly generalized to accommodate for parallel composition. In defining fair behavior, we do not care about irrelevant events. Thus, a user is expected to declare not only those events that he wants to have treated fairly, but also those that constitute a relevant behavior.

5.1 Introduction

First we state the Dining Philosophers problem. M philosophers, numbered from zero to $M-1$, $M \geq 3$, are seated around a table. Lying on the table are M forks, again numbered from zero to $M-1$, in such a way that fork number i is located between philosopher i and $i \oplus 1$[1]. (See Figure 5.1.) There is a big bowl of spaghetti at the center of the table. Each philosopher is initially dormant (thinking maybe). It is assumed that when he gets hungry, philosopher number i needs *both* forks adjacent to him in order to eat, and he grabs both forks in one move; otherwise he does not pick up the fork e.g. to his right when the one to his left is in use. Since we may in addition want to register a request to eat made by a philosopher, we identify three events associated with philosopher number i:

1. He may announce his intention to eat by issuing an r_i event;
2. He grabs the two forks adjacent to him and starts eating, represented by an n_i event[2], and finally;

[1]We use circle to denote addition/subtraction modulo M.

[2]Here the bowl of spaghetti is our shared resource, and as before, the event of accessing the

3. He releases the forks and stops eating, modeled by an x_i event.

Thus, the alphabet of events over which the actions of the i^{th} philosopher are defined is:

$$\Sigma_i := \{r_i, n_i, x_i\}\ , \quad i = 0, \ldots, M-1.$$

We denote $\Sigma := \bigcup_{i=0}^{M-1} \Sigma_i$. The seat arrangements and the automaton model for philosopher number i are shown in the Figure 5.1.

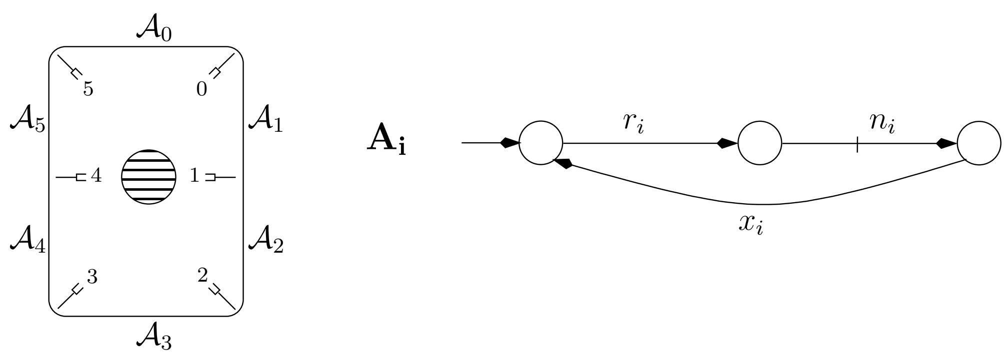


Figure 5.1: The Dining Philosophers problem ($M = 6$).

Based on previous assumptions, the i^{th} philosopher cannot act freely and independently of his neighbors. He is granted permission to eat only when forks number $i \ominus 1$ and i are not in use. Since $\mathcal{A}_i$ shares these forks with his neighbors $\mathcal{A}_{i\ominus 1}$ and $\mathcal{A}_{i\oplus 1}$, respectively, the above restriction states that $\mathcal{A}_i$ is forbidden to eat when either of $\mathcal{A}_{i\ominus 1}$ or $\mathcal{A}_{i\oplus 1}$ is eating. This restriction on the philosophers' behavior is specified by the automaton of Figure 5.2.

Thus, all we care to know about forks is how they restrict the philosophers' interactions. In other words, our plant is simply the shuffle product of the philosophers, while the forks prescribe the legal behavior expected of the system. The plant and

resource (eating) is named 'n_i' after 'enter'.

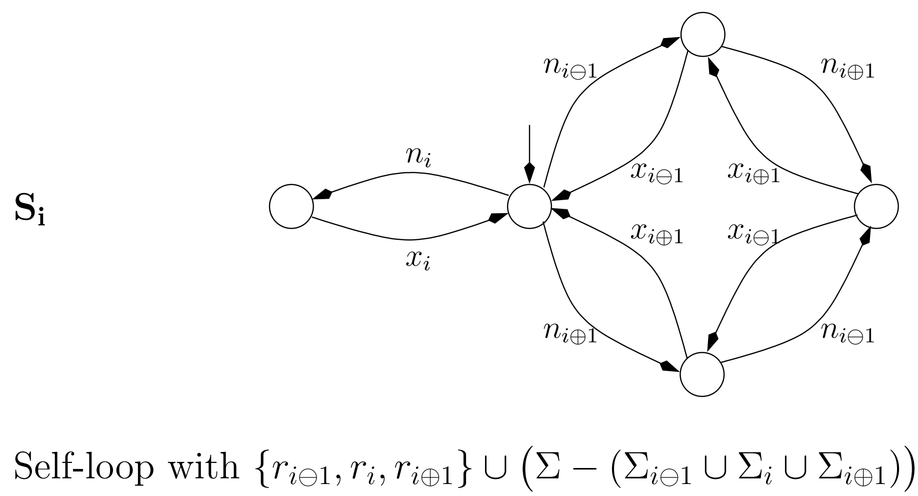


Figure 5.2: The requirement on the behavior.

the specification automata are obtained by composing their respective components:

$$\mathbf{A} := \prod_{i=0}^{M-1} \mathbf{A}_i \qquad\qquad \mathbf{S} := \prod_{i=0}^{M-1} \mathbf{S}_i$$

Taking n_i's as the only controllable events, it can be verified (using TCT for instance) that S (the language generated by $\mathbf{S}$) is controllable with respect to A (the language generated by $\mathbf{A}$) when the 'n' events are controllable. This can be accomplished modularly by verifying that S_i is controllable with respect to A, i.e. the following result[3].

Lemma 5.1 *Let S_i, $i = 0, \cdots, M-1$, be closed and controllable with respect to A. Then $\bigcap_{i=0}^{M-1} S_i$ is closed and controllable with respect to A.*

[3]Obviously $\mathbf{S}_i$ does not demand the removal of any event it is self-looped with, therefore one only needs to verify the controllability of S_i with respect to $A_{i\ominus1}||A_i||A_{i\oplus1}$.

Proof. We have:

$$\begin{aligned}
(\bigcap_{i=0}^{M-1} S_i)\Sigma_u \cap A &= \bigcap_{i=0}^{M-1} (S_i\Sigma_u) \cap A \\
&= \bigcap_{i=0}^{M-1} (S_i\Sigma_u \cap A) \\
&\subseteq \bigcap_{i=0}^{M-1} S_i
\end{aligned}$$

■

As a result, **S** can be thought of as a (modular) supervisor for the system which implements a kind of mutual exclusion among the philosophers: no two neighboring philosophers are allowed to dine (access the resource) at the same time. We know from past experience that such a naive (although in fact 'optimal') implementation of mutual exclusion leaves the door open for starvation. And in fact **A** controlled by **S** can be potentially unfair: The i^{th} philosopher will never be offered a chance to eat if the $(i \ominus 1)^{\text{th}}$ and $(i \oplus 1)^{\text{th}}$ philosophers overlappingly eat while each keeps himself "alive" and never stagnates in a state. This situation can never happen when $M = 3$ since the two neighbors of a philosopher are themselves neighbors and therefore cannot eat at the same time (i.e. overlappingly). That is to say, we note that $\mathcal{A}_j$ and $\mathcal{A}_k$ are neighbors if and only if $j \oplus 1 = k$ or $j \ominus 1 = k$. But when $M = 3$ for each $i = 0, 1, 2$ we have:

$$(i \oplus 1) \oplus 1 = i \ominus 1$$

Thus, $\mathcal{A}_{i\oplus 1}$ and $\mathcal{A}_{i\ominus 1}$ are neighbors and therefore are banned from eating overlappingly by the safety specification.

However, fairness becomes an issue as soon as M is increased to 4, as Figure 5.3 suggests. If $\mathcal{A}_0$ and $\mathcal{A}_2$ adopt the eating pattern shown in the figure, neither $\mathcal{A}_1$ nor $\mathcal{A}_3$ will ever be able to eat.

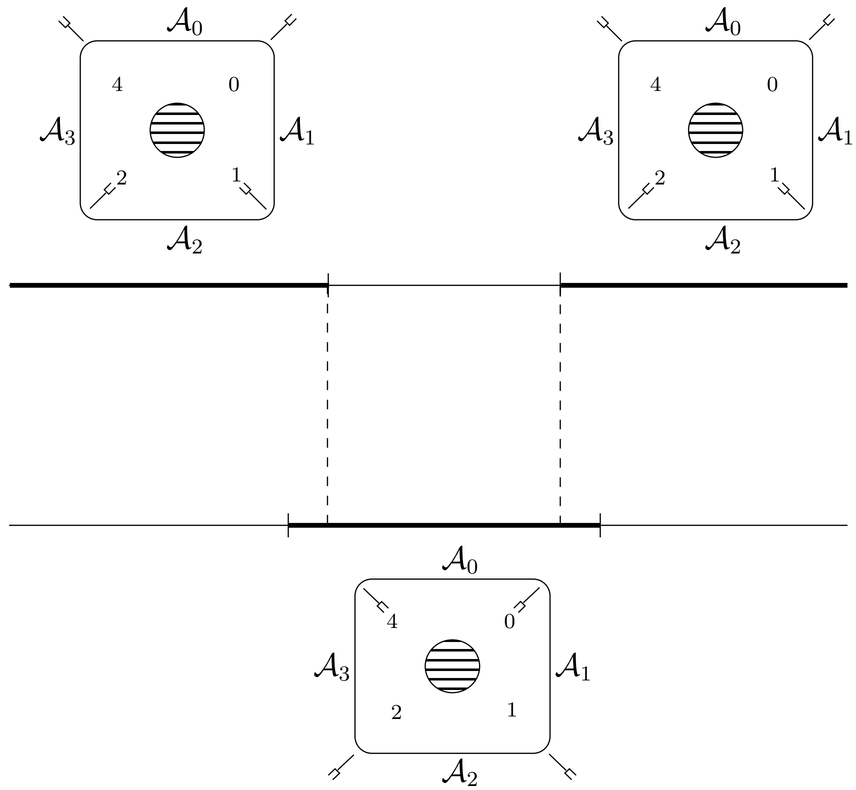


Figure 5.3: The 1st and 3rd philosophers starve.

Figure 5.4 clearly shows how $\mathbf{A}_1$'s moves are restricted by the supervisor component $\mathbf{S}_1$, which governs the concurrent operation of $\mathbf{A}_0$, $\mathbf{A}_1$, and $\mathbf{A}_2$, in other words $\mathbf{A}_1$ and his neighbors. Each coordinate is used to represent a philosopher: for instance, philosopher number 1 is represented in z-dimension. To avoid cluttering the picture 'x' transitions are not shown; an x_i transition follows an n_i and takes the system back to where the corresponding r_i was initiated (therefore, the three form a local loop). It can be seen that while $\mathbf{A}_0$ and $\mathbf{A}_2$ take transitions between the demarcated states, the n_1 transition is constantly disabled, and therefore the first philosopher is forever denied to eat while its two neighbors remain alive.

FIFO queues provide an efficient way to overcome unfairness. Service is granted in the order that requests are made. Back to Figure 5.4, under the queue the system

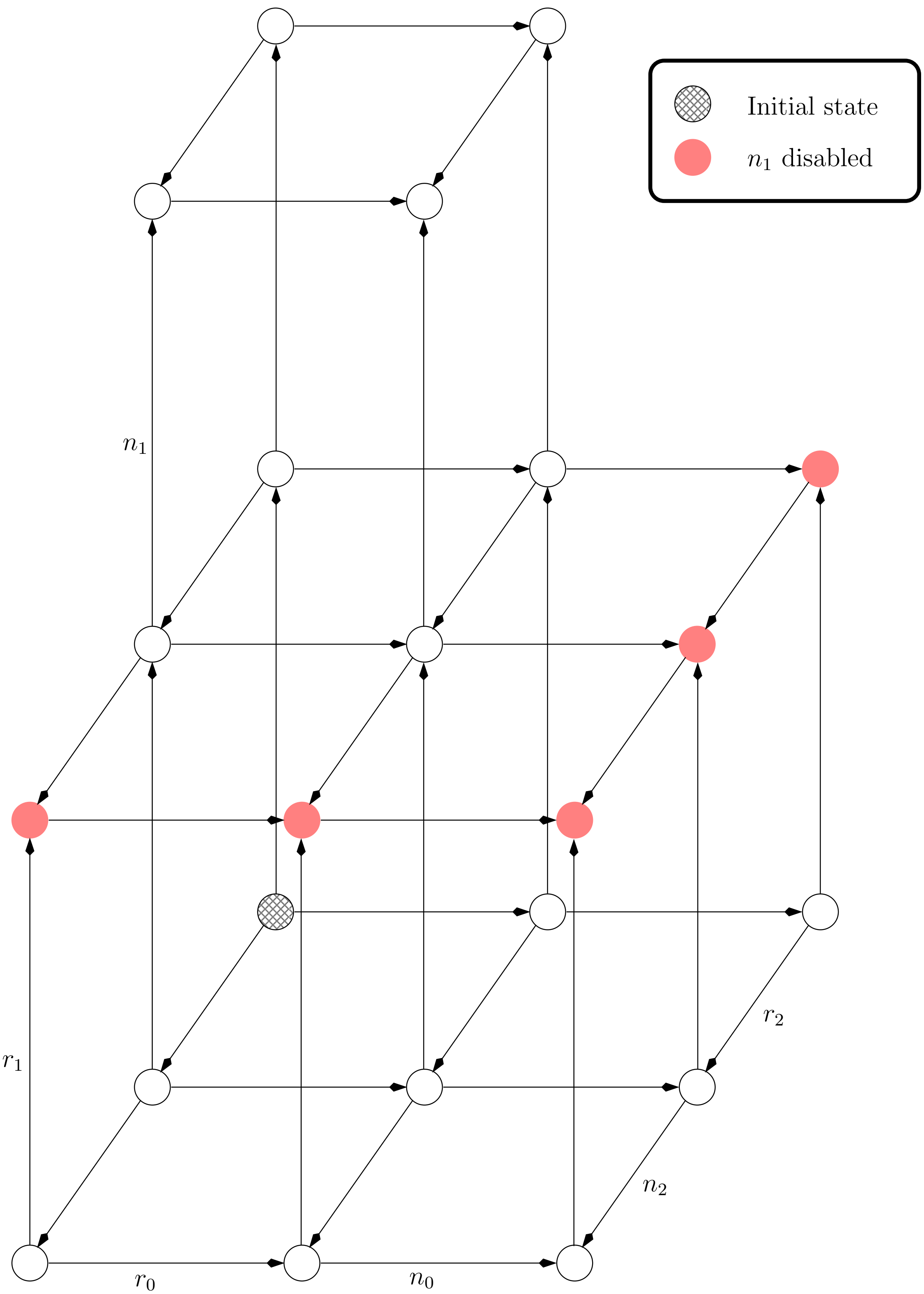
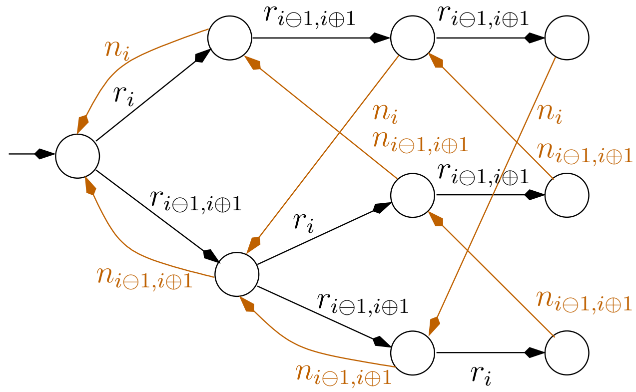


Figure 5.4: Concurrent operation of $\mathbf{A}_0$, $\mathbf{A}_1$, and $\mathbf{A}_2$ under $\mathbf{S}_1$.

can no longer move freely between the grey states on the second floor since a request by $\mathbf{A}_1$ is already placed.

Like the supervisor $\mathbf{S}$, the queue $\mathbf{Q}$ is implemented component-wise: for each $i = 0, \ldots, M-1$, the local queue $\mathbf{Q}_i$ sets the order between philosophers $\mathbf{A}_{i\ominus 1}$, $\mathbf{A}_i$, and $\mathbf{A}_{i\oplus 1}$. The queue component $\mathbf{Q}_i$ is shown in Figure 5.5.

$\mathbf{Q_i}$



Self-loop with $\{x_{i\ominus 1}, x_i, x_{i\oplus 1}\} \cup \big(\Sigma - (\Sigma_{i\ominus 1} \cup \Sigma_i \cup \Sigma_{i\oplus 1})\big)$

Figure 5.5: The queue component $\mathbf{Q}_i$.

By careful examination of all possible combinations of strings in Figure 5.4, it can be verified that when $\mathbf{A}_{i\ominus 1}||\mathbf{A}_i||\mathbf{A}_{i\oplus 1}$ is controlled by $\mathbf{Q}_i||\mathbf{S}_i$, the longest run at all states of which n_1 is disabled has length five (the run

$$\{r_0 r_2 r_1, r_0 r_2 r_1 n_0, r_0 r_2 r_1 n_0 n_2, r_0 r_2 r_1 n_0 n_2 x_0, r_0 r_2 r_1 n_0 n_2 x_0 r_0\}$$

is one example) and therefore the closed-loop system is 5-fair under the queue.

One intriguing facet of the above solution is its modular nature. We recall from the RW theory that decentralized control may cause a system to block, i.e. when all the modular supervisors are nonblocking, the system under all controls may fail to be coreachable. In other words, the intersection of controllable and $\mathcal{F}_{\mathrm{I}}$ sublanguages[4],

[4] Defined earlier.

although controllable (as attested by the Lemma earlier in this section), may fail to be an $\mathcal{F}_{\mathrm{I}}$ sublanguage. Therefore, it is natural to have a similar concern when the $\mathcal{F}_{\mathrm{III}}$ class is considered instead. We show that the intersection of the above $\mathcal{F}_{\mathrm{III}}$ sublanguages is still $\mathcal{F}_{\mathrm{III}}$, but this is not the case in general.

5.2 Decentralized control

In this section we investigate an ad hoc method for design of decentralized supervisory control in the Dining Philosophers Problem, proposed by [Won02], Exercise 4.7.5. Let L as before represent the open-loop behavior of a system.

Definition 5.1 *Given $\tau \in \Sigma$ and $N \in \mathbb{N}$, $K \subseteq L$ is said to be (τ, N)*-realizable *iff:*

1. *K is controllable with respect to L, and*
2. *K is N-fair to τ with respect to L, i.e.:*
   (a) *There is no run in K of length greater than N at all states of which τ is disabled, and*
   (b) *K does not contain a terminal string at which τ is disabled.*

When K is (τ, N)-realizable it can be thought of as the language generated by the closed-loop system under a fair supervisor.

As noted in the previous section, when K_i's are controllable, so is their intersection. Thus, the main issue in a decentralized design is whether the second property in Definition 5.1 is preserved under intersection.

Back to the DP problem, specifically, Figure 5.2 of Page 90, we observe that among all $\mathbf{S}_j$'s, $j = 0, \ldots, M-1$, n_i is disabled only in the specification components $\mathbf{S}_{i\ominus 1}$, $\mathbf{S}_i$, and $\mathbf{S}_{i\oplus 1}$. Moreover, whenever n_i is disabled in $\mathbf{S}_{i\ominus 1}$ or $\mathbf{S}_{i\oplus 1}$, it is also disabled in $\mathbf{S}_i$. In other words, $\mathbf{S}_i$ contains all instances where n_i is disabled in $\mathbf{S}$; if n_i is disabled at $s \in A$ when the system is supervised by $\prod_{j=0}^{M-1} \mathbf{S}_j$, it must have also been disabled if the system were supervised by $\mathbf{S}_i$ alone.

Also, observe in Figure 5.5 that n_i is disabled only in $\mathbf{Q}_{i\ominus 1}$, $\mathbf{Q}_i$, and $\mathbf{Q}_{i\oplus 1}$, and moreover, it is disabled by $\mathbf{Q}_{i\ominus 1}$ or by $\mathbf{Q}_{i\oplus 1}$ only if it is disabled by $\mathbf{Q}_i$.

This observation leads to the following general result.

Lemma 5.2 *Let $K_j \subseteq L$ for $j \in J$, where J is some index set, and $K := \bigcap_{j \in J} K_j$. Suppose there exists an $i \in J$ such that for every $j \in J$ and $s \in K$, when τ is disabled at s in K_j it is also disabled at s in K_i. Then for any $s \in K$, τ is disabled at s only if it is disabled by K_i.*

Proof. The assumption is:

$$\exists i \in J, \forall j \in J, \forall s \in K.\ s\tau \in L - K_j \rightarrow s\tau \in L - K_i$$

Let $s \in K$. If $s\tau \in L - K$, then we must have $s\tau \notin K_j$ for some $j \in J$, i.e. $s\tau \in L - K_j$. By the assumption it follows that $s\tau \in L - K_i$. Thus, we have shown that:

$$\exists i \in J, \forall s \in K.\ s\tau \in L - K \rightarrow s\tau \in L - K_i$$

■

The above result states that under the assumption, when K is obtained by intersecting K_i with other components, no new bad string is created; if τ is disabled at any string of K, it must have been disabled at the same string of K_i. As a matter of fact, since $s\tau \in L - K_i$ implies $s\tau \in L - K$, τ is disabled anywhere in K if and only if it is disabled in K_i.

The following corollary of the lemma is immediate.

Corollary 5.1 *Under the assumption made in the lemma, K satisfies part (a) of*

Definition 5.1 if K_i does.

Proof. Suppose K_i satisfies part (a) of the definition, i.e.:

$$\forall s \in K_{i_b}, t \in \Sigma^*.\ |t| = N \wedge st \in K_i \rightarrow \exists u \leq t.\ su \in K_{i_g} \tag{5.1}$$

Let $s \in K_b$ and $t \in \Sigma^*$ be such that $|t| = N \wedge st \in K$. We have:

$$s \in K \wedge s\tau \in L - K$$

which by the lemma implies:

$$s \in K_i \wedge s\tau \in L - K_i$$

Since $st \in K_i$ it follows from (5.1) that for some $u \leq t$ we have $su \in K_{i_g}$, i.e. $su\tau \in L \rightarrow su\tau \in K_i$. By the lemma this implies $su\tau \in L \rightarrow su\tau \in K$, i.e. $su\tau \in K_g$.

■

Thus, we have shown that if each local supervisor contains all instances of disablement of a certain event, no new bad run is created when the local supervisors are combined and part (a) of Definition 5.1 can be verified *locally*. Unfortunately, as the following example suggests, our assumption does not warrant local verification of part (b); two or more non-terminal strings may have a *longest* prefix (which is therefore terminal in the intersection language), at which τ may happen to be disabled.

Example 9 The languages $K_1, K_2 \subseteq L$ are both 1-fair to α. That is, α is not disabled for two consecutive transitions in their respective generators. In addition, we note that when the system is supervised concurrently, the K_1-part takes complete care of all instances where α has to be disabled.

From the previous lemma it follows that α is not disabled for two consecutive

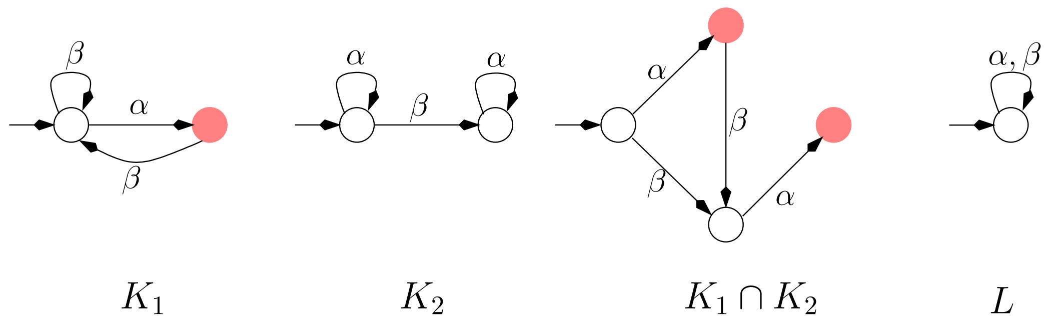


Figure 5.6: Intersection may leave behind bad terminal strings which were non-terminal in the components.

transitions anywhere in the generator of $K_1 \cap K_2$. However, $K_1 \cap K_2$ has a pair of terminal strings (namely, $\beta\alpha$ and $\alpha\beta\alpha$) which are non-terminal in K_1 and K_2, and at which α happens to be disabled. Thus, $K_1 \cap K_2$ is not 1-fair to α. $\square$

Thus, the core difficulty in modular supervisor design seems to be the fact that the local supervisors may not agree on the next move and the entire system may halt as a result. In fact, it will be shown that the general version of this problem is co$\mathcal{NP}$-complete [LP98].

TERMFREE

INSTANCE: A collection of languages K_i, $i = 1, \ldots, l$.

QUESTION: Are all strings of $K := \bigcap_{i=1}^{l} K_i$ non-terminal?

The problem is inherently difficult to solve, and it will be shown to be co$\mathcal{NP}$-complete. First we observe that TERMFREE is in co$\mathcal{NP}$ since a terminal string of K can be regarded as a succinct witness of *non*-membership in TERMFREE. We show completeness by reducing SATISFIABILITY [GJ79] to the complement problem, stated formally below:

TERMFREE^c

INSTANCE: A collection of languages K_i, $i = 1, \ldots, l$.

QUESTION: Does $K := \bigcap_{i=1}^{l} K_i$ have a terminal string?

By means of an example we demonstrate the reduction from SATISFIABILITY to $\mathsf{TERMFREE}^c$.

Example 10 Consider the following boolean formula in Conjunctive Normal Form (CNF):

$$\phi = x_1 \wedge (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2} \vee x_3)$$

In our reduction, each clause in the formula corresponds to a specification component, therefore $l = 3$. The first clause is true for the truth assignment '1xx' where 'x' can be either 0 or 1. This is shown in Figure 5.7.

Each truth assignment is encoded by a string in the usual manner, for instance, '010' is encoded as $\overline{x_1}x_2\overline{x_3}$. Note that all the satisfying truth assignments are terminal strings of K_1, and vice versa.

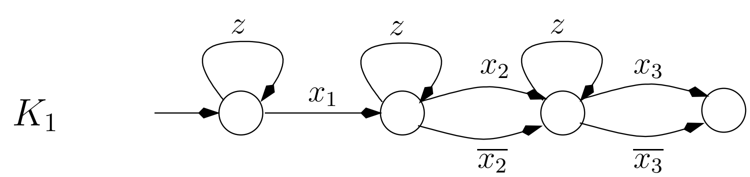


Figure 5.7: K_1: The automaton representation of the 1st clause.

The second clause is true either when the first variable is assigned true, i.e. for any truth assignment of the form '1xx', or when the first variable is assigned false but the second is assigned true, i.e. for all truth assignments of the form '01x'. This is shown in Figure 5.8.

Finally, the last clause is equivalent to $\overline{x_1} \wedge x_2 \rightarrow x_3$. It becomes true when either the antecedent is false (that is, for all truth assignments of the form '1xx' and '00x'), or the antecedent and the consequent are both true, i.e. for the truth assignment '011'. This is graphically depicted by the automaton of Figure 5.9.

We observe that the formula is satisfiable (for instance, T='111' will do) and

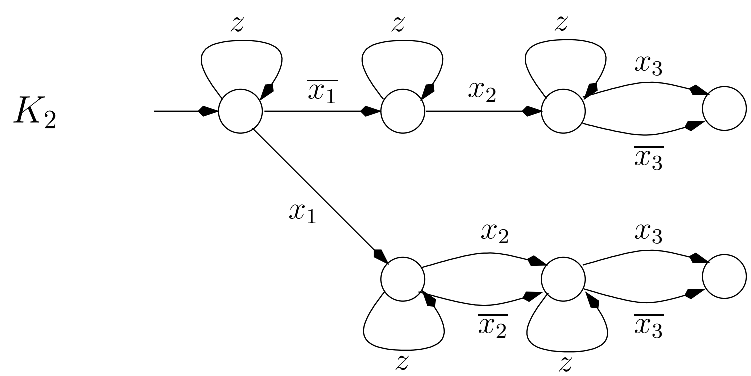


Figure 5.8: K_2: The automaton representation of the 2nd clause..

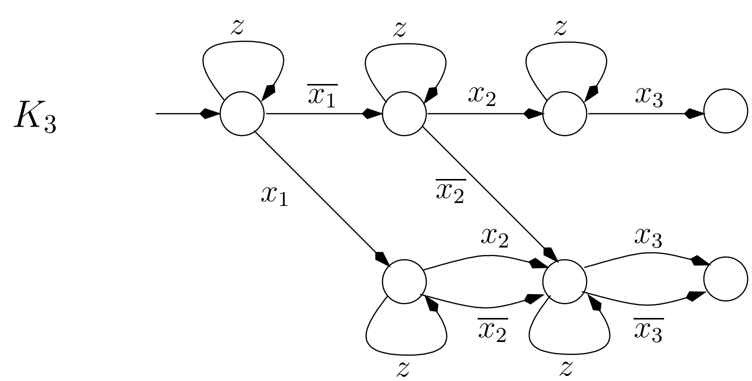


Figure 5.9: K_3: The automaton representation of the 3rd clause.

moreover, $x_1x_2x_3$ is a terminal string of $\bigcap_{i=1}^{3} K_i$. Therefore, $\{K_1, K_2, K_3\}$ is a 'yes' instance of $\mathsf{TERMFREE}^c$. $\square$

Lemma 5.3 *$\mathsf{TERMFREE}^c$ is $\mathcal{NP}$-hard.*

Proof. We prove the result by reducing satisfiability in polynomial time to the problem at hand. Let ϕ be a CNF boolean formula in n variables and m clauses. We assume that literals in a clause appear in numerical order. Corresponding to the j^{th} clause, ϕ_j, $j = 1, \ldots, m$, a finite automaton $\mathbf{K}_j$ is constructed over $\{z\} \cup \bigcup_{i=1}^{n}\{x_i, \overline{x_i}\}$ as follows.

1. $\mathbf{K}_j$ has $2n+1$ states arranged in two rows; the first row is numbered from 0 to

n, and the second from $1'$ to n'.

2. All states, except for n and n', are self-looped with z.

3. Second row transitions: for $2 \leq k \leq n$, there are a pair of transitions labeled with x_k and $\overline{x_k}$ connecting $(k-1)'$ to k'.

4. First row transitions: for $1 \leq k \leq n$:

   (a) If the k^{th} literal is absent from the clause, $(k-1)$ is connected to k by a pair of transitions labeled with x_k and $\overline{x_k}$;

   (b) If x_k appears in the clause:

      i. if it is the last literal, $(k-1)$ is connected to k by an x_k transition;

      ii. otherwise, $(k-1)$ is connected to k by an $\overline{x_k}$ transition, and is connected to k' by an x_k transition[5].

   (c) If $\overline{x_k}$ appears in the clause: switch $\overline{x_k}$ and x_k in (4b).

For example, when $n = 4$, we have the automaton of Figure 5.10 corresponding to $(\overline{x_2} \vee x_4)$.

Observe that the construction of $\mathbf{K}_j$ from ϕ_j is done in polynomial time. We call a string t *complete* if $t = y_1 \cdots y_n$, where $y_i = x_i$ or $\overline{x_i}$ for $i = 1, \ldots, n$. A complete string encodes a truth assignment to the variables $x_1, \ldots, x_n$. A complete string is *j-satisfying* if it satisfies ϕ_j. Denote by P_z the projection map that removes all occurrences of z from a string. By construction we have that:

1. if t is j-satisfying then it is terminal in K_j;

2. if s is terminal in K_j then $P_z s$ is j-satisfying, and finally;

[5]The reason for the negation is that in converting a disjunction to an implication, we take the last literal as the consequent, while the rest are negated and conjoined to form the antecedent, thereby the change in sign.

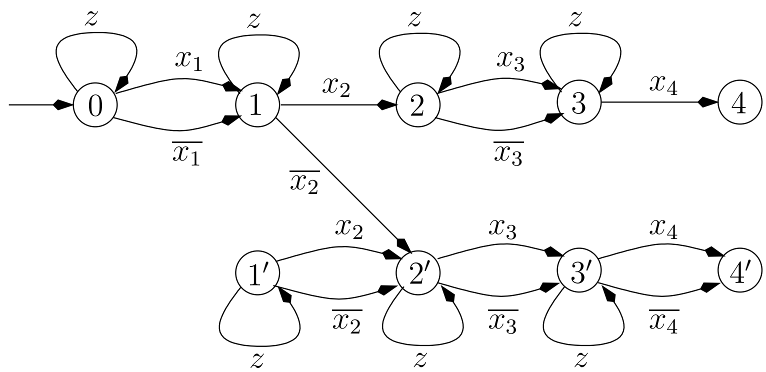


Figure 5.10: Reduction of Lemma 5.3 for the clause $\overline{x_2} \vee x_4$.

3. if s is accepted by K_j and $P_z s$ is complete then $P_z s$ is j-satisfying.

We now show that $\phi = \bigwedge_{j=1}^{m} \phi_j$ is satisfiable if and only if $K = \bigcap_{j=1}^{m} K_j$ has a terminal string. Let ϕ be satisfiable, i.e. there exists a string t that is j-satisfying for all $j = 1, \ldots, m$. Then t must be terminal in all K_j, and therefore terminal in K. Conversely, suppose s is terminal in K. Since z is self-looped at all but the right-hand side states, s must take at least one $\mathbf{K}_j$ to one of its right-hand side states, i.e. s must be terminal in K_j for some j. This implies that $P_z s$ is j-satisfying and hence complete. Since s is accepted by all K_j's and $P_z s$ is complete, we conclude that $P_z s$ satisfies all clauses. That completes the proof. ■

Previously, we have seen that the first step towards achieving decentralized control is to demand a "truly" decoupled control, that is, requiring that each event be disabled by a certain supervisor whenever it is disabled by others. This is as opposed to having the task of disabling an event divided between two or more supervisors. But later we found that this cannot forestall a deadlock: the system under control may be left with no choice if the local supervisors fail to agree on the next move.

When the system is made up of a number of components, deadlock can be averted by requiring that in all cases, *at least one component not be restricted by any of the supervisors*, and be allowed to move freely. Then, we know for sure that any string under control can be extended at least with events participating from one of the components.

Back to the DP problem, we observe that under control, the system may come to a complete halt only when each philosopher has made a request that is not yet satisfied, i.e. every philosopher is ready to eat. This is the only situation where all possible next moves are controllable and therefore can be disabled by local controllers. Suppose the i^{th} philosopher is the first having an active request registered. We claim that n_i is not disabled by any of the controllers, and so there is a choice—namely, n_i—agreed on by all controllers for the next move. To see this, back to Figure 5.5, Page 94, observe that n_i can be disabled by $\mathbf{Q}_{i\ominus 1}$, $\mathbf{Q}_i$, and $\mathbf{Q}_{i\oplus 1}$ only. (Note that $\mathbf{S}_i$'s do not impose any restriction since service is not granted to any philosopher yet.) Since the i^{th} philosopher is the first among all issuing a request (r_i), the three local controllers listed above disable $\{n_{i\ominus 1}\}$, $\{n_{i\ominus 1}, n_{i\oplus 1}\}$, and $\{n_{i\oplus 1}\}$, respectively. Hence, n_i is not disabled by any local controller and therefore the concurrent operation of local controllers is possible (i.e. does not result in any new terminal string).

5.3 Generalizing the definition

So far in our investigation we have been concerned with fair treatment of an event when the context consisted of all events. Sometimes when assessing how fairly an event is treated it may make more sense to be indifferent to a subset of events. For instance, in the Dining Philosophers problem, we would like to give a fair chance to survive to the i^{th} philosopher in a world consisting only of him and his two neighbors. It is then natural to expect that the manner in which the philosopher sitting opposite to him at the table behaves does not affect our verdict on how fairly the i^{th} philosopher is treated: after all, it is only his two immediate neighbors who can restrict his behavior.

As another example, consider a system made up of two copies of the Readers-Writers problem, discussed in the previous chapter. Thus, there are two separate databases which are accessed exclusively by each copy. If we judge that a reader from the first copy is treated fairly in an implementation, it may be reasonable to assume that the way the second copy is implemented should not affect our judgment. In other words, we consider the two copies *asynchronous*, as if two clocks running at independent and perhaps different speeds governed their operations. We simply do not care if one copy is a lot faster than the other, and only one database is being used continually without the other ever being accessed. On the other hand, we may seek fairness in a broader sense; not only do we want fair access to each database, but we also want each database to be accessed frequently. In this case, it is as though the clocks in each copy were *synchronized*, although they still may run at different rates (therefore, we may tolerate, say, up to 10 accesses to the first database before the second has been called). In this case, when we talk of fair treatment of an event, our scope covers the entire set of events.

The following example attempts to illustrate the above differing points of view on fairness.

Example 11 Consider the system below, where the open-loop system is composed of two independent components.

K L

Figure 5.11: Is K 1-fair to α?

We note that α is disabled at the right-hand side state by the controller. Depending on the viewpoint taken, we may or may not consider K to be treating α fairly. In the first reading, we consider α in competition with β only. Thus, while α is disabled, we do not mind if γ is taken non-stop. Therefore, we consider the system 1-fair to α when the "scope" consists of $\{\alpha, \beta\}$ only. On the other hand, if for any reason the user decides that α is competing with γ as well, then the closed-loop system becomes unfair; when α is disabled, the system can stay alive while performing γ self-loops at the right-hand side state indefinitely. $\square$

In the following two subsections we generalize the definition of fairness in two directions: first, we introduce the notion of *scope of behavior*, and then we consider designs that are fair to *all* events (or sets of events) in a set.

5.3.1 Scope of behavior

When talking of fair behavior, we identify two ingredients which were often kept implicit in our discussions: first, a language modeling the open-loop behavior of the system (often denoted by L) and second, an event (or a set of events) which we would like to be treated fairly (usually denoted by τ when a single event is considered). The argument we made earlier in this section favors the introduction of a third ingredient, namely, the subset of events which forms our scope. In the above example, for instance, we consider a controller fair to α in the first scenario if it treats α fairly when the scope of behavior is $\{\alpha, \beta\}$; that is, when the fairness of a string is not affected in any way by transitions labeled with γ.

Let $L \subseteq \Sigma^*$ model the open-loop behavior of a system, $\mathcal{T} \subseteq \Sigma$ be a subset of events, and $\tau \in \mathcal{T}$ be an event. Denote by $P : \Sigma^* \to \mathcal{T}^*$ the natural projection map.

The definition of fairness is generalized accordingly.

Definition 5.2 *For $N \in \mathbb{N}$, we call a sublanguage $K \subseteq L$ N-fair if for all $s \in K_b$,*

$$(\exists t \in \Sigma^*.\ st \in K_g) \wedge (\forall t \in \Sigma^*.\ |Pt| = N \wedge st \in K \rightarrow \exists u \leq t.\ su \in K_g)$$

As before, we require that τ not be disabled at terminal strings (if any) of a fair implementation, but now disabling τ is tolerated as long as the overall *projected* length of the run does not exceed N, i.e. it does not count towards unfair behavior if the system moves "silently" while taking transitions outside $\mathcal{T}$. For instance, when the scope is taken to be $\mathcal{T} = \{\alpha, \beta\}$, the design K_1 of Figure 5.12 is considered 2-fair to α, whereas K_2 and K_3 are not. Although α is disabled by K_1 at all strings of a run of length 3 (namely, $\{\epsilon, \gamma, \gamma\beta\}$), the length of the projected run (that is, $\{\epsilon, \beta\}$) is 2 and therefore K_1 is considered 2-fair.

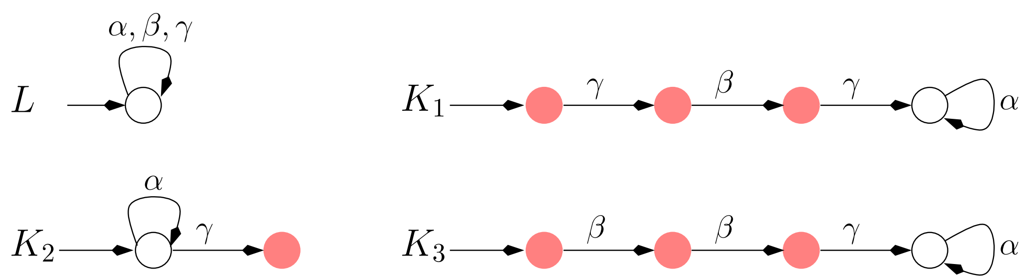


Figure 5.12: With $\mathcal{T} = \{\alpha, \beta\}$, K_1 is 2-fair to α, but K_2 and K_3 are not.

In the previous section we claimed that the local controller $\mathbf{Q}_i || \mathbf{S}_i$ is 5-fair to n_i. Technically speaking, the local controller is 5-fair when the scope consists of philosopher number i and his two neighbors, i.e. $\mathcal{T}_i := \Sigma_{i \ominus 1} \cup \Sigma_i \cup \Sigma_{i \oplus 1}$.

It may make more practical sense to include only controllable events in the scope. After all, the supervisor has no choice over uncontrollable events, and it cannot be held responsible if the plant makes uncontrollable moves while an event is disabled. In other words, the supervisor cannot treat an event any more fairly than the plant itself. Thus, in the Dining Philosophers problem we would like n_i to be treated fairly among $\mathcal{T}_i := \{n_{i\ominus 1}, n_i, n_{i\oplus 1}\}$. A FIFO queue provides an "impartial"[6] solution to the problem, in the sense that it does not discriminate between a philosopher and his two neighbors. It is then no surprise that when the scope $\mathcal{T}_i := \{n_{i\ominus 1}, n_i, n_{i\oplus 1}\}$ is selected, $\mathbf{Q}_i||\mathbf{S}_i$ is "2"-fair to n_i: the i^{th} philosopher may have to wait for his "two" neighbors to finish eating (in case they have made a request before him) before he can start.

The fairness predicate as defined in this section remains preserved under arbitrary union, and a largest N-fair sublanguage of a given language exists as a result. The computation of such a sublanguage remains much the same, with the exception that the numbering of strings is carried out in a slightly different fashion. The counter, initialized at zero, increments to one when τ has just been disabled. It is incremented by one, up to $N+1$, only when τ remains disabled after a transition inside $\mathcal{T}$ is taken. Thus, the function $f : K \to \{0, 1, \ldots, N+1\}$ becomes: $f(s) = 0$ when $s \in K_g$; when $s \in K_b$, f is defined recursively as follows:

- $f(\epsilon) = 1$
- $f(s\sigma) = \begin{cases} f(s) + 1 & ; \quad f(s) = 0 \vee (\sigma \in \mathcal{T} \wedge f(s) \leq N) \\ f(s) & ; \quad \text{otherwise} \end{cases}$

  Note that τ has just been disabled at $s\sigma$ when $s \in K_g$ and $s\sigma \in K_b$, or in other words when $f(s) = 0$ and $s\sigma \in K_b$, in which case we define $f(s\sigma) := 1 = f(s)+1$, regardless of whether $\sigma \in \mathcal{T}$ or not.

Strings that are indexed $N + 1$ belong to a bad run of projected length longer than N and have to be removed from a design in order to make it fair. The resulting sublanguage is N-fair if there is no terminal string at which τ is disabled; if there is one, the procedure outlined in the last chapter must be performed to get rid of all such terminal strings. The above has to be repeated if new bad runs with projected length longer than N are born.

[6] Not related to the notion of "impartiality" formally defined in the literature.

5.3.2 Fair to all

So far we have studied fairness to an event τ, which is a special case of fairness to a single set of events $T \subseteq \Sigma$ where $T = \{\tau\}$. When T is not a singleton, we simply consider a string "good" if at least one event in T is not disabled, or equivalently, a string is considered "bad" if all events in T are disabled:

$$s \in K_b \quad \text{iff} \quad s \in K \wedge \forall \tau \in T.\ s\tau \in L - K$$

Thus, the events in T can be viewed as replacements to one another; it does not matter if K is unfair to a particular event as long as it treats them fairly overall. So the focus is on the fair treatment of the "disjunctive occurrence" of events in T. This could be useful when for example a manufacturing unit can process material in two alternative ways, and fairness to the unit amounts to including at least one of the two methods in our computations. All results of the previous chapter stand unchanged when T is not a singleton.

In this section we seek generalization in a different direction, namely, when fairness to all events in a set is concerned. In other words, our focus is now shifted to the "conjunctive occurrence" of events in a set. This was the case for example in the Dining Philosophers problem, where fairness to *all* philosophers was sought. To accommodate the previous paragraph as well, we define fairness to a set of sets; that is, we specify $T \subseteq Pwr(\Sigma)$, a set of subsets of Σ. An implementation is fair to $T = \{T_1, T_2, \ldots, T_n\}$, where $n \in \mathbb{N}^+$ and $T_i \subseteq \Sigma$ for $1 \leq i \leq n$, if it is fair to *all* T_i's, $1 \leq i \leq n$. We further assume that fairness to each set T_i is determined in a scope $\mathcal{T}_i \subseteq \Sigma$. Thus the definition is generalized as follows.

Definition: Global fairness.

Let $n, N \in \mathbb{N}^+$ and $T_i, \mathcal{T}_i \subseteq \Sigma$ for $1 \leq i \leq n$. A design $K \subseteq L$ is N-fair to $T := \{T_1, T_2, \ldots, T_n\}$ iff for all $1 \leq i \leq n$ it is N-fair to T_i when the scope is $\mathcal{T}_i$. $\square$

The definition is slightly simplified when T_i's are singletons, i.e. $T_i = \{\tau_i\}$ for all $1 \leq i \leq n$. As before, we define:

$$K_{i,b} := \{s \in K |\ s\tau_i \in L - K\} \quad \text{and} \quad K_{i,g} := K - K_{i,b}$$

For $1 \leq i \leq n$ we let $P_i : \Sigma^* \to \mathcal{T}_i^*$ denote the natural projection map.

Definition 5.3 *A design $K \subseteq L$ is N-fair to $T := \big\{\{\tau_1\}, \{\tau_2\}, \ldots, \{\tau_n\}\big\}$ iff:*

$$\forall 1 \leq i \leq n.\ x_i(K) \wedge y_i(K)$$

where:

$$x_i(K) := \forall s \in K_{i,b} \exists t.\ st \in K_{i,g}$$
$$y_i(K) := \forall s \in K_{i,b} \forall t.\ |P_i t| = N \wedge st \in K \to \exists u \leq t.\ su \in K_{i,g}$$

We note that when a finite number of indexed properties are all preserved under arbitrary union, so is their "and"; hence, the property of fairness to T is preserved under arbitrary union and a largest fair design exists. To compute such a design, each state is augmented by n integer variables: the i^{th} variable, $1 \leq i \leq n$, keeps track of the number of past "observable" transitions, up to $N+1$, where τ_i has been consecutively disabled. Naturally, in an admissible design, no counter field exceeds N.

Previously, we have seen that the local queue solution $\mathbf{Q}_i || \mathbf{S}_i$ is 2-fair to n_i when $\mathcal{T}_i := \{n_{i\ominus 1}, n_i, n_{i\oplus 1}\}$. This result is valid for all $i \in \{0, 1, \ldots, M-1\}$, and it turns out that the intersection of local controllers is 2-fair to all "start eating" events.

Theorem 5.1 *The intersection* $\prod_{i=0}^{M-1} \mathbf{Q}_i || \mathbf{S}_i$ *is 2-fair to* $T := \{\{n_0\}, \{n_1\}, \ldots, \{n_{M-1}\}\}$ *when the scope is* $\mathcal{T}_i := \{n_{i\ominus 1}, n_i, n_{i\oplus 1}\}$ *for all* $i \in \{0, 1, \ldots, M-1\}$.

As mentioned earlier, FIFO queues grant service in the order that requests have been made. The DP becomes 2-fair under (local) FIFO queues, as a philosopher may have to wait for his two neighbors to release the forks before he can start eating. Thus, when a philosopher has made a request, it is no longer possible for his left and right neighbors to collaborate to starve him by alternately eating with overlaps.

Assume that a sample run of the system starts off with the sequence of requests $r_0 r_1 r_2$. Then the philosophers are given permission to eat in exactly the same order: $\mathbf{Q}_0$ requires that philosopher number 0 be served before philosopher number 1, while $\mathbf{Q}_1$ demands that philosopher number 1 eats before philosopher number 2. Thus, projecting out all subsequent x_i and r_i events, as well as the events outside $\mathcal{T}_1 := \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$, we conclude that $r_0 r_1 r_2 n_0 n_1 n_2$ is the only possible projected continuation of the sequence.

We notice, however, that keeping the order of requests is not asked by the original problem, where all one cares about is that no philosopher starves; or in the bounded version each philosopher gets to eat at the latest after his two neighbors are finished. Even the aforementioned statement is stronger than what was actually intended: we do not want to force a philosopher to eat, we just want to give him a *chance* to eat following his two neighbors. In other words, we only want the supervisor to stop disabling the "eat" event after his two neighbors are done, but we leave it up to the philosopher to actually decide to eat. If he chooses not to eat the supervisor may have to disable his "eat" event again for up to two more turns in case his neighbors have requested to eat again.

The above two characteristics—that the order of requests does not have to be preserved and that each philosopher must be given a fair *chance* to eat (which he may not take)—are the earmarks of the optimal design presented in this work. They highlight the difference between our design and the more restrictive ones such as queues. For instance, the continuation $r_0r_1r_2n_2n_0$ of the initial request sequence is accepted by our optimal 2-fair design, but not by the queue solution for two reasons: first, the order in which service is granted is different from that of the requests. And second, the string can be continued with $x_2r_2x_0r_0$ in the optimal design, at which point n_1 is enabled and eligible, but does not have to be taken. That means we can instead continue with n_2, after which n_1 is disabled once again. Thus, the collection $r_0r_1r_2(n_2n_0x_2r_2x_0r_0)^*$ can be generated in the optimal design. (See Figure 5.13.)

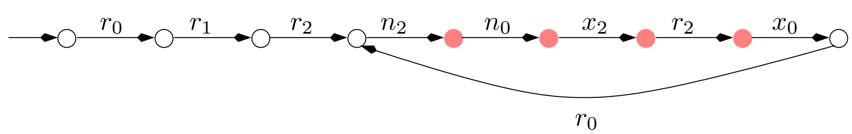


Figure 5.13: A collection of 4-fair (2-fair if only the 'n' events from the neighborhood constitute the scope) sequences that are included in the optimal design, but are excluded from the queue. n_1 is disabled at filled states.

Although n_1 is never taken, we suppose it is because philosopher number 1 has never really been hungry, since otherwise he could have eaten given the fair chance provided by the supervisor.

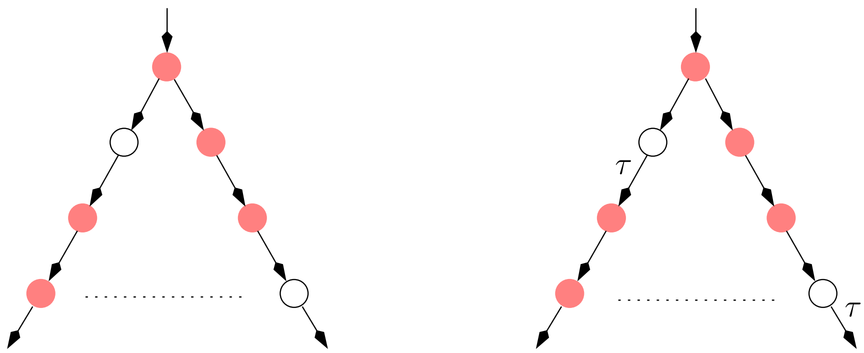
Chapter 6

Strong Fairness

6.1 Introduction

FIFO queues are fair in a stronger sense than what has been described so far. In addition to offering an event a fair chance to be taken, a queue, loosely speaking, forces the occurrence of an event by disabling all its competitors. Therefore, when the moment is right (for instance, when a timer has expired or a certain number of transitions have elapsed), the event of interest becomes the only choice the supervisor offers for the next move. In any sufficiently long run of such a fair system, an event is *taken* after being disabled continuously. This is in contrast with what has been studied previously (known as *weak fairness*, or *justice* [MP92]), where an event only has to become *enabled* or *ineligible* after being disabled long enough. A weakly-fair controller is supposed only to provide an event with a chance to occur; it is indifferent to whether the chance is actually taken. In contrast, we call the new property, which requires that all outstanding requests be granted, *strong fairness*, or *compassion* [MP92]. In the bounded case weak and strong fairness are illustrated schematically in Figure 6.1. As usual a string at which τ is disabled is shaded grey, and a transition with no label corresponds to the occurrence of events other than τ.

We consider a reachability subtree rooted at a bad string. Observe that in (a) it is not the case that τ remains disabled after three consecutive transitions. It means that in any branch of length three there is at least a string at which either τ is enabled or otherwise ineligible. Meanwhile, the much stronger property in (b) requires that τ occurs at least once in any branch of the subtree of length four.



(a) (b)

Figure 6.1: (a) Weak fairness (b) Strong fairness ($N = 3$).

Once again, instead of talking about events occurring at unspecified moments in the future, we place a strict bound N on the number of transitions allowed before something desired by the specification happens. Thus, in effect, we study a safety version of a liveness property. Recall that a safety property states that "bad things never happen". A safety property has a finite counterexample, namely, a bad thing. It means that if the system violates a safety property it does so in a finite number of steps, i.e. it is capable of generating a finite sequence which does not satisfy the property.

A liveness property, on the other hand, states that "something good will (eventually) happen". A liveness property constrains infinite sequences only; unlike safety properties it does not restrict the finite behavior in any way. Thus, no finite counterexample to a liveness property can exist; as a matter of fact, any finite word can be completed to an infinite one accepted by the property.

A Büchi automaton [B6̈0] is often used to express general properties (finite or infinite) of a discrete-event system. An infinite behavior is accepted if and only if it visits a subset of states—called the *accepting states*—infinitely often.

When represented by a Büchi automaton, all states of a safety property are accepting, as it restricts behavior only by constraining finite prefixes of strings. A Büchi automaton representing a liveness property, on the other hand, does not restrict the finite behavior, and therefore all events are valid choices for the next move. In fact, it is shown in [Alpern] that any Büchi automaton can be written as the product of a safety and a liveness automaton. Such a decomposition is also possible in the RW framework when specifying a plant or a requirement on its behavior. A specification on the system behavior can be written in two parts: one that instructs how transitions should be enabled or disabled at each state, and a second part that indicates what marking of states is desired. Figure 6.2 shows how the decomposition is carried out. $\mathbf{A}$ is decomposed into two product components: $\mathbf{A_1}$ where all states are marked, and $\mathbf{A_2}$ where all events are enabled.

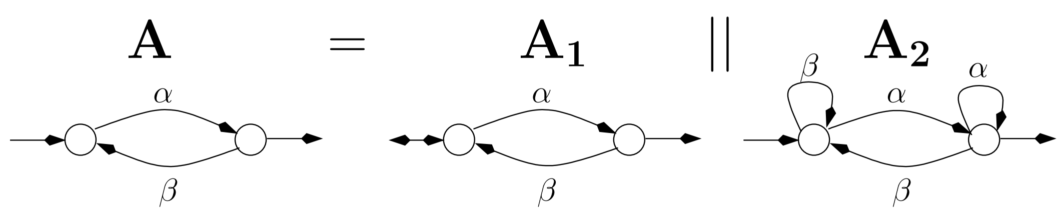


Figure 6.2: Decomposition of an automaton into "all marked" and "all enabled".

As noted earlier, placing a bound in effect converts a liveness property into a safety one. With a bound, a liveness property is violated if good things fail to happen after a specified number of steps. This point of view is illustrated in the example of Figure 6.3. Over the alphabet $\{\alpha, \beta\}$, the Büchi automaton $\mathbf{A}$ expresses the property $\Diamond\Box\beta$, that in any infinite word accepted by $\mathbf{A}$, an infinite sequence of β's follows a

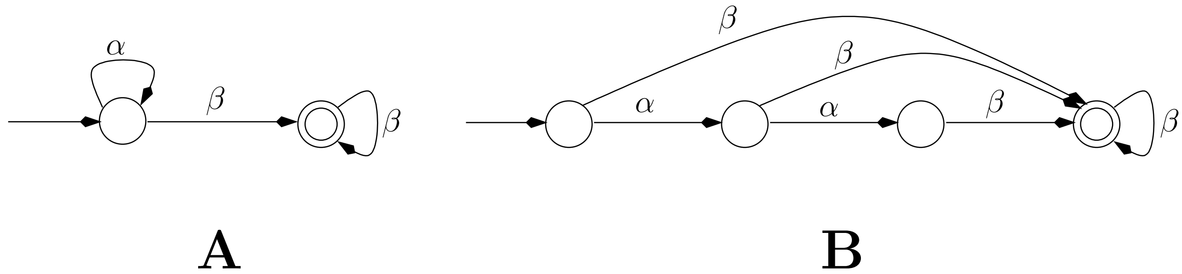


Figure 6.3: A liveness property and its bounded, safety version.

finite sequence of α's.

If one is not patient enough for β's to happen "eventually", he may specify a bound, say $N = 3$, by which the good thing (β) should happen. This is modeled by **B**, where **B** is a restriction of **A**, obtained by disabling α after it occurred twice. Note that **B** is a safety property [1] which prohibits α from being a choice for the next move after it is taken twice.

Finally, we would like to comment that it is no surprise that we have always modeled our specifications by prefix-closed languages: a safety property *is* prefix-closed.

6.2 Strong fairness

In this section we formally define strong fairness, and will show that every strongly fair design is weakly fair. All definitions are with respect to a language L, an event $\tau \in \Sigma$, and a bound $N \in \mathbb{N}$.

Definition 6.1 *A design $K \subseteq L$ is strongly N-fair to τ iff for every string $s \in K$ we have:*

[1] Making the first three states accepting will have no impact on the acceptance condition, as no string visits those states infinitely often.

1. $\exists t.\ st \in K_g$;

2. $\forall t_0, t_1, \ldots, t_N, t.\ t_0 < t_1 < \cdots < t_N < t \wedge st \in K \wedge \forall i.\ st_i\tau \in L \rightarrow \exists u < t.\ u\tau \leq t$.

The first condition concerns only designs with terminal strings: it requires that τ not be disabled at a terminal string.

In the interesting case where τ is a choice at N different prefixes of t but is not yet taken, the second condition states that while executing the rest of t the event τ must be taken the next time it is offered by the plant. Observe that unlike what was assumed in the introduction, the instances of eligibility need not be consecutive. Nonetheless, the above condition is in harmony with strong fairness already defined in the computer science literature.

We first show that strong fairness is in fact a stronger property than weak fairness, i.e. a design is weakly fair if it is strongly fair. To facilitate the proof we repeat the definition of weak fairness.

Definition 6.2 *A design $K \subseteq L$ is weakly N-fair to τ iff for every string $s \in K$ we have:*

1. $\exists t.\ st \in K_g$;

2. $\forall t.\ st \in K \wedge |t| = N \rightarrow \exists u \leq t.\ su \in K_g$.

Note that both conditions are true if s is a good string: the first condition is true simply because $s\epsilon \in K_g$, while the second condition is vacuously true since $u = \epsilon$ satisfies $su \in K_g$.

Now we are ready to prove that strong fairness implies weak fairness.

Theorem 6.1 *Suppose $K \subseteq L$ is strongly N-fair. Then K is weakly N-fair.*

Proof: By contradiction. Suppose t is such that $st \in K$, $|t| = N$, but $\forall u \leq t.\ su \in K_b$. Since K is strongly N-fair, it cannot terminate at the bad string st, therefore we must have $st' \in K$ for some $t < t'$. For $i = 0, 1, \ldots, N$ define t_i to be the i^{th} prefix of t (the 0^{th} prefix is ϵ). Note that $t_0 < t_1 < \cdots < t_N < t'$, $st' \in K$, and that by the contradiction assumption we have $st_i\tau \in L$ for all i. By the strong fairness assumption, therefore, it follows that $\exists u < t'.\ u\tau \leq t'$. This implies $\exists u \leq t.\ su \in K_g$, a contradiction. ■

Remark: The distinction between weak and strong fairness can best be understood by the following example. Consider a Q&A session at the end of a talk. Each audience member raises his hand if he wants to ask a question. Depending on the capacity of the lecture room (and the history of the lecturer, of course!), a person can hold his hand up either briefly just to get noticed or steadily until he is given permission to speak.

The first policy, which may be wise to adopt in small audiences, relies on the fact that the system is strongly fair: no request remains unanswered, and a request is recognized as soon as raised hand is identified. So the request remains registered even after the hand is dropped. On the other hand, in a larger hall, it is more difficult for the lecturer to remember where all the requests have originated. So it will be wiser to renew a request whenever it is time for the next question, or alternatively to continue to hold a hand up until permission is granted. Such a setting is weakly fair, as it guarantees service (if time permits) to continuous requests only. □

Given a scope $\mathcal{T} \subseteq \Sigma$, one can require strong fairness in a world consisting only of members of $\mathcal{T}$. The definition is revised accordingly by introducing the projection map $P : \Sigma^* \to \mathcal{T}^*$.

Definition 6.3 *A design $K \subseteq L$ is strongly N-fair to τ iff for every string $s \in K$ we have:*

1. $\exists t.\ st \in K_g$;
2. $\forall t_0, t_1, \ldots, t_N, t.\ Pt_0 < Pt_1 < \cdots < Pt_N < Pt \wedge st \in K \wedge \forall i.\ st_i\tau \in L \to$

$\exists u < t.\ u\tau \leq t.$

The strong N-fairness property is easily seen to be preserved under arbitrary union, and a largest strongly N-fair design exists as a result. Such a design is obtained by appropriately numbering strings. Initially, or when τ is just taken, a string is numbered 0. When $\sigma \neq \tau$ has just taken at s, the counter is incremented by one, up to $N+1$, if $\sigma \in \mathcal{T}$ and τ was eligible to occur at some prefix of s which is *observationally equivalent* to s. Given a projection map P, two strings are observationally equivalent if they have the same projection under P. A design is strongly N-fair if it does not have any string numbered with $N+1$. Formally, the function $f : K \rightarrow \{0, 1, \ldots, N+1\}$ is defined as follows.

- $f(\epsilon) = f(s\tau) := 0.$
- $f(s\sigma) := \begin{cases} f(s) + 1 & ; \quad \sigma \in \mathcal{T} \wedge f(s) \leq N \wedge \exists s'.\ s' \leq s \wedge Ps' = Ps \wedge s'\tau \in L \\ f(s) & ; \quad \text{otherwise} \end{cases}$

6.3 Example: mutual exclusion by semaphore

The example of this section is adapted from [Manna95]. The control program MUX-SEM shown below ensures mutual exclusion between two programs by semaphores.

$$\begin{array}{c}
\textbf{local}\ r:\ \textbf{integer where}\ r = 1 \\
P_1 :: \left[\begin{array}{l} l_0 :\ \textbf{loop forever do} \\ \left[\begin{array}{l} l_1 :\ \textbf{noncritical} \\ l_2 :\ \textbf{request}(r) \\ l_3 : \textbf{critical} \\ l_4 : \textbf{release}(r) \end{array}\right] \end{array}\right] \quad P_2 :: \left[\begin{array}{l} m_0 :\ \textbf{loop forever do} \\ \left[\begin{array}{l} m_1 :\ \textbf{noncritical} \\ m_2 :\ \textbf{request}(r) \\ m_3 : \textbf{critical} \\ m_4 : \textbf{release}(r) \end{array}\right] \end{array}\right]
\end{array}$$

The schematic statements **critical** and **noncritical** stand for arbitrary sections of code that can be described as critical and noncritical sections, respectively. We assume that these sections of code do not modify the value of r. Moreover, it is assumed that the critical section in each parallel program always terminates, that is to say, no program can stay in its critical section forever. The objective of control is to ensure mutual exclusion, i.e. that the two programs will never be in their critical sections at the same time. In the RW framework, the plant components P_1 and P_2, and the specification on behavior are shown in Figure 6.4. For simplicity, we abstracted away the critical and noncritical sections of the code and focused on the semaphore mechanism.

MUX-SEM uses semaphore statements to implement mutual exclusion. A **request**(r) statement can be executed when r is positive, and it decrements r by one. On the other hand, no condition is placed on the execution of a **release**(r) statement, and it increments r by 1.

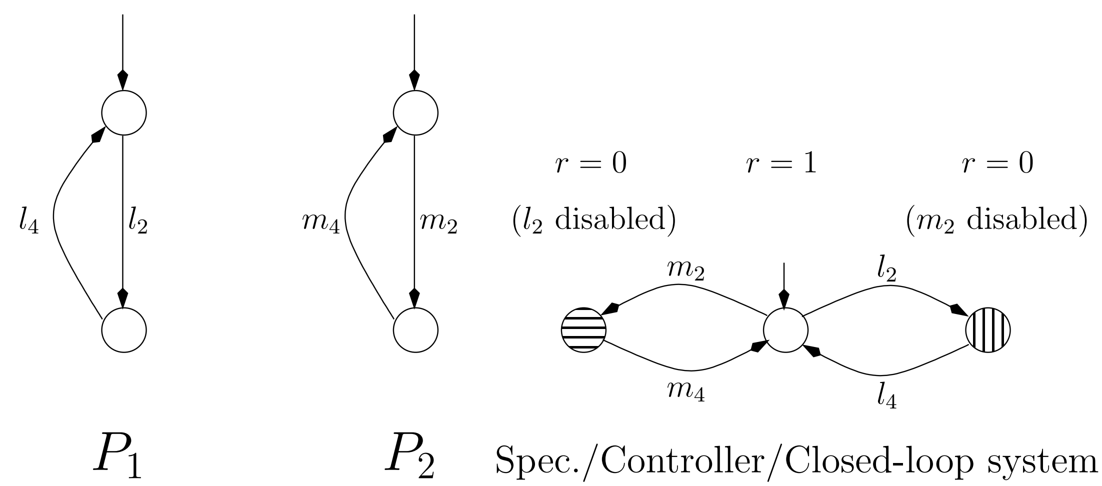


Figure 6.4: Mutex by semaphores. The transitions l_2 and m_2 are guarded by $r > 0$ condition and hence are controllable. Initially, $r = 1$ and both l_2 and m_2 are enabled. As soon as one of the two is taken, r is decremented to zero and the other transition gets disabled (its guard becomes false) as a result. $r = 1$ is restored when the resource is released.

The variable r is initialized at 1, and the shared resource can be seized by either P_1

or P_2. Whichever program executes the request statement first can enter its critical section and block the other by setting the value of r to 0. As soon as the program executes all its critical code it sets r back to one and *gives* the other program a *chance* to advance. The implementation MUX-SEM is in fact optimal in the RW framework.

We observe that m_2, say, will be reenabled only one transition after being disabled by MUX-SEM. Therefore, MUX-SEM is weakly 1-fair to l_2 and m_2 when the scope is $\{l_2, m_2\}$. However, with the same scope MUX-SEM is not strongly N-fair for any bound N. For any N, the left-hand or the right-hand side loops can be taken $N+1$ times consecutively before the request transition from the other side is taken.

The procedure introduced in this section can be employed to compute, say, the optimal strongly 1-fair sublanguage of MUX-SEM. Two number fields are attached to each state in order to keep track of the number of times l_2 and m_2 have been eligible but not taken. The l_2-counter (left) is reset initially or when l_2 has just been taken, and it is incremented by one when l_2 is eligible but m_2 is taken instead. The update rule for the m_2-counter (right) is similar, with the roles of l_2 and m_2 interchanged. The optimal design is reached upon trimming all states whose counter values exceed one.

6.4 Example: C1M1

The Cat and Mouse example has motivated our study of fairness. Initially, cats and mice are locked in their separate cages. The controller implements mutual exclusion by ensuring that at any given moment only one cage is unlocked. In C1M1, there is one of each animal. The closed-loop system has been shown to be weakly 1-fair when the scope consists of all enter events, as the cat and the mouse are given a chance to enter the room as soon as the other is locked back in its cage.

However, we will show in this section that the system fails to be strongly fair.

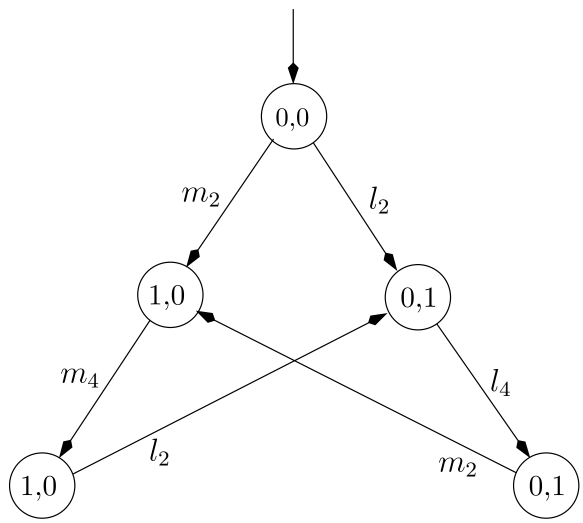


Figure 6.5: Optimal strongly 1-fair sublanguage of MUX-SEM.

This could cause a problem if, for instance, the cat is much faster than the mouse. In this case, although they both are given equal opportunity to enter the room, every competition ends in favor of the cat, and the mouse, being too slow, may never acquire the resource (enter the room in this case). A strongly fair controller enforces order in the system, regardless of how competent the players really are. It takes away from the players some of the individual freedom they had formerly, as the system is now controlled more tightly in order to forestall such malevolence.

We then present two of the traditional remedies, a FIFO queue and a turn variable, under which the system becomes strongly fair. Finally, we compare those with our optimal solution, which restricts the system just enough to implement strong fairness, without introducing the limitations caused by queue or turn variable.

6.4.1 The system model

The cat and the mouse are each modeled by a three-state automaton. We distinguish between when they are at rest, or have made a request, or finally when their cages are open. The closed-loop system under mutual exclusion is shown in Figure 6.6.

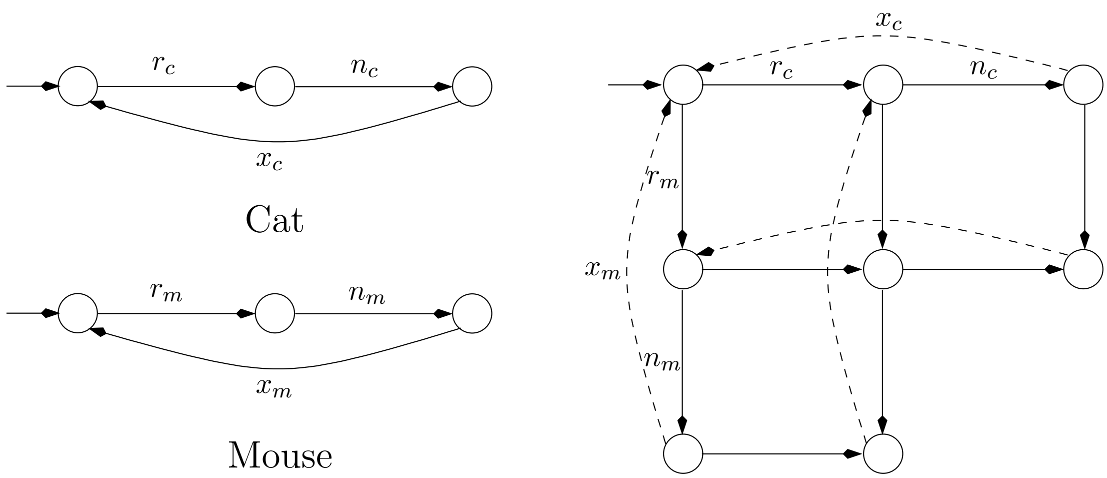


Figure 6.6: The C1M1 model.

To study fairness we take the scope to be $\mathcal{T} := \{n_c, n_m\}$. Then it is reasonable to select $N := 1$ as bound since in a fair implementation we expect a player to be given a chance to move forward (weak fairness), or to be forced to move forward[2] (strong fairness), after the other has just released the resource. It can be verified that the closed-loop system is weakly 1-fair, but not strongly 1-fair. As a matter of fact, the system is not strongly N-fair for any N as, for instance, when a request is made by the mouse, the cat may repeat the moves $r_c n_c x_c$ $(N+1)$-times before the mouse enters.

In the following subsections we present the strongly 1-fair restrictions of the above solution.

6.4.2 Queue

The mutual exclusion design can be made strongly 1-fair by adding a FIFO queue. Since each player is served after the other leaves the room, the system under control

[2]As discussed earlier, this is accomplished by eliminating all competing choices.

is strongly 1-fair. This design is shown in Figure 6.7.

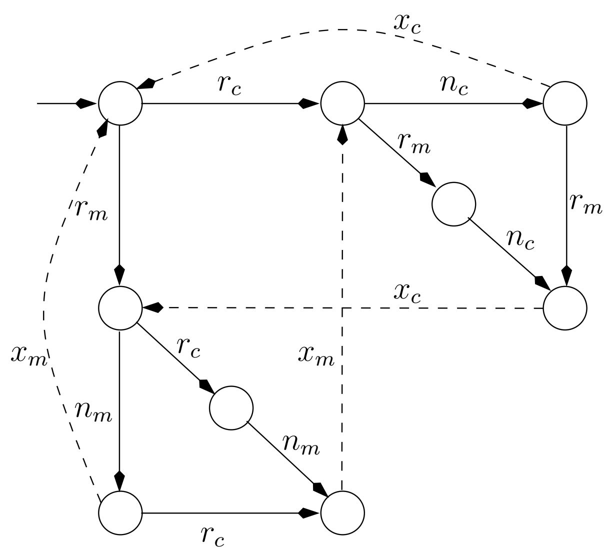


Figure 6.7: C1M1 under queue.

6.4.3 Turn variable

Alternatively, one can achieve strong fairness by implementing a turn variable. Such a design is shown in Figure 6.8. A turn variable acts as a tie-breaker whose value is flipped each time it is invoked. Initially turn is given to the cat. When requests by both are outstanding, permission to enter is granted to whoever is next, and the value of the turn variable is switched to the other. We assume that the turn variable does not change value otherwise, in particular, when there are no simultaneously active requests by both players.

6.4.4 Optimal solution

Finally, the optimal strongly 1-fair solution is computed as outlined earlier and is presented in Figure 6.9. Each state is augmented by two integers which count the

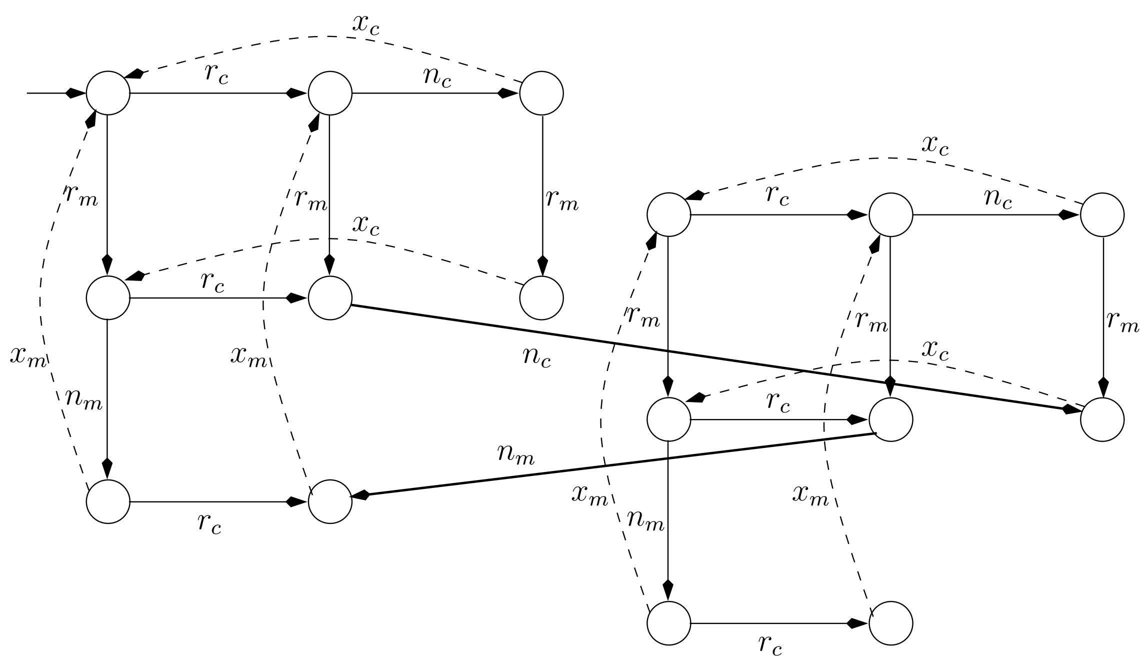


Figure 6.8: C1M1 controlled by a turn variable.

number of times the cat and the mouse, respectively, had a chance to enter the room in the open-loop system. The counter for each player is reset each time he enters the room, and is unaffected by transitions outside the scope.

It can be verified that the optimal solution is less restrictive than both strongly 1-fair designs previously presented. Each player is served as soon as it makes a request, or right after the other has been served. This is precisely what fairness specification requires, namely, that a player is granted service no more than once while the other player waits for access to the resource.

Both solutions seen before implement a property that was not demanded by the fairness specification. A queue grants service in the order that requests are made. The optimal design obviously ignores the order and only ensures that nobody is served twice running when an active request by the other exists. The turn variable design, on the other hand, preselects the cat or the mouse, depending on whose turn is coming,

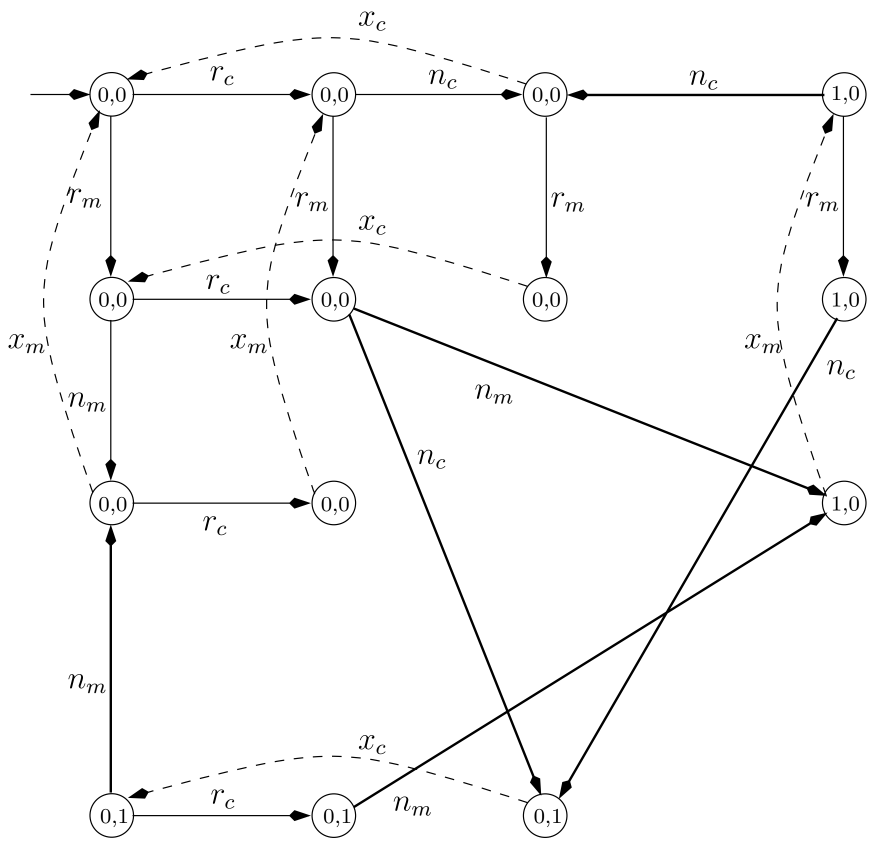


Figure 6.9: The optimal strongly 1-fair solution.

and never offers both players the possibility to advance. This is in contrast with the optimal design, which does not intervene unless one player has already entered the room. It can be verified using TCT that the optimal solution is larger than either of the solutions obtained by queue or turn variable.

Chapter 7

Why Are Queues Efficient?

7.1 Introduction

The strong fairness condition demands that the distance between successive appearances of a symbol never exceeds a given bound. We suggested, through an example, that queues are strongly fair but not optimal, as they insist on maintaining the same order for services and requests. For example, for two agents **A** and **B**, the sequence[1]

$$(r_a r_b) n_b n_a (x_a x_b r_a r_b) n_b n_a (x_a x_b r_a r_b) \ldots$$

is strongly 1-fair to both n_a and n_b, but it is not accepted by a FIFO queue as the order of services does not obey that of requests.

The distinction between queues and optimally fair designs becomes more evident as the number of requests per agent allowed by a queue is increased. In the above example, that number has been implicitly assumed to be equal to one for both agents. Hence, for each agent, requests and accesses must appear alternately. When a number larger than one is selected, regardless of order of service, an optimal design manifests

[1] We grouped events outside the scope between brackets.

a behavior richer than that shown by a queue. In other words, the extra behavior is structurally different from what a queue accepts.

Having said that, we make a case in favor of queues by showing that they can be built *hierarchically.* A queue for a group of players can be formed by grouping players in teams, maintaining order between teams, and finally maintaining order between players in each team. A queue therefore provides an efficient solution to the fairness problem.

7.2 Motivation

Consider three agents who are competing to access a shared resource exclusively. This is modeled by the automaton of Figure 7.1, where the agents access the resource by performing an 'a', 'b' or 'c', and release it by taking an 'x', 'y' or 'z' transition, respectively.

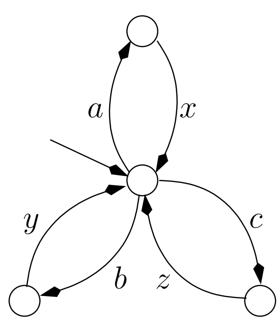


Figure 7.1: Mutual exclusion between three agents.

A single automaton like the above does not provide any information about control. In particular, when a string is absent, the model does not specify whether it was excluded by the supervisor, or it was physically impossible to be generated by the plant. So Figure 7.1 could be used to model a system that can physically do only what is prescribed, or as we intended here it may model a closed-loop system where the agents operate independently, but a controller ensures that when one event in the set $\{a, b, c\}$ is taken the other two are disabled. However, the issue of whether the system is open- or closed-loop is irrelevant when our focus is on strong fairness.

Remark 1. Perhaps this is the right moment to clear up an ambiguity from a claim made earlier. We stated that a design (closed-loop system) cannot be any fairer than the plant (open-loop system). We argued that *weak* fairness concerns controlled systems only, and becomes an issue when an event is continuously disabled longer than allowed. In the absence of control, therefore, a system is always weakly fair. It may, however, fail to be strongly fair if beyond a point it never takes a choice. Thus, a system may be examined for strong fairness even when it is not under control. $\square$

Remark 2. One must be careful with terminology when fairness fails to hold. We call a system *strongly unfair* when it is not weakly fair. A strongly unfair system is unfair to a high degree; C2M1 is an example of one such system where a process (mouse) may never get a chance to access a shared resource. Naturally, in this case the system is not strongly fair either. On the other hand, we call a system *weakly unfair* if it is not strongly fair. Thus we know about the system that it is somewhat unfair, but to what extent we cannot tell. C1M1 and C2M1 are examples of weakly unfair systems. A weakly unfair system may or may not be weakly fair: for example, C1M1 is weakly fair, whereas C2M1 is not. The negation rule is simple: when applying "not", adverb and adjective are both negated. The diagram of Figure 7.2 illustrates the point. As seen from the diagram, the level of fairness drops from fair to unfair as one goes down the ladder. The "strong" adjective describes the intensity of the respective property (fair or unfair) at each end of the ladder. $\square$

We enrich our terminology by the following definition.

Definition 7.1 Let Σ be an alphabet, $\mathcal{T} = \{\tau_i\}_{i=1}^{M}$ be a subset of Σ, and $\{N_i\}_{i=1}^{M}$ be a list of M natural numbers. A string $s \in \Sigma^\omega$ is *strongly fair* to $\{(\tau_i, N_i)\}_{i=1}^{M}$ if it is

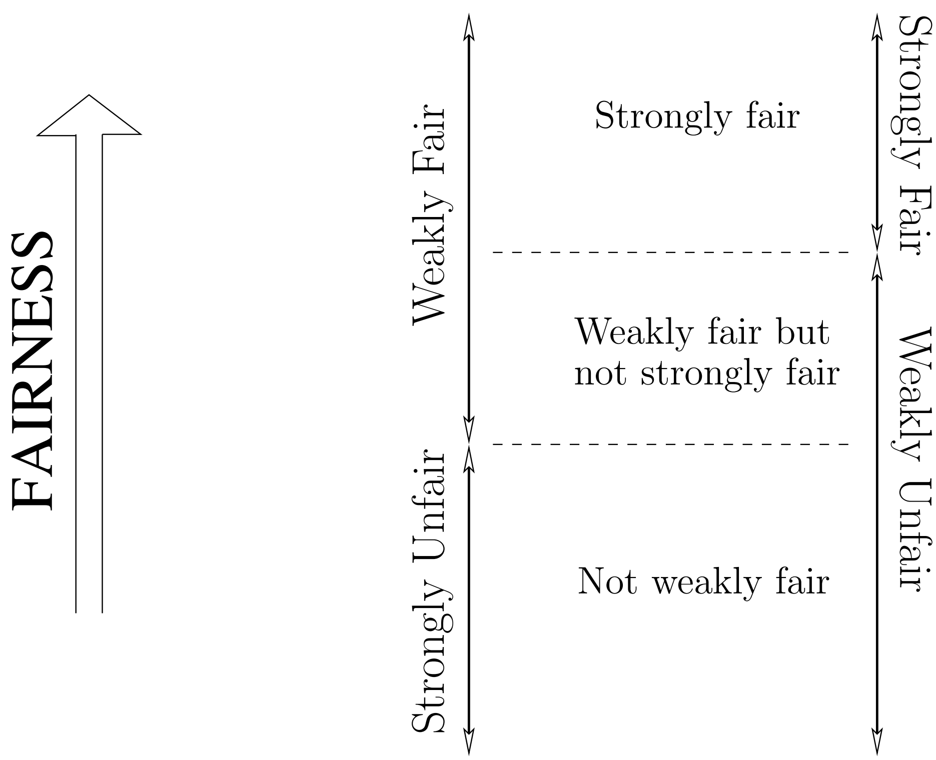


Figure 7.2: Terminology when negating weak and strong fairness.

strongly N_i-fair to τ_i for all $i = 1, \ldots, M$ when the scope is $\mathcal{T}$. A language is strongly fair to $\{(\tau_i, N_i)\}_{i=1}^{M}$ if all its strings are. $\square$

A similar definition may be phrased for weak fairness.

There are two features in the above definition that must be emphasized. First, we only talk about "bounded" fairness. For instance, over the alphabet $\Sigma = \{a, b\}$, the sequence:

$$s = ababbabbbabbbba\ldots$$

is not strongly fair to $\{(a, N_a), (b, N_b)\}$ for any natural number N_a. This is despite the fact that 'a' *does* occur infinitely often in the sequence and in an absolute sense s is strongly fair to 'a'.

And second, we must specify which events are in the race and have to be scheduled fairly. The above sequence is boundedly fair to 'a' if 'b' is out of contention. (In fact, it is strongly fair to $\{(a, 0)\}$ in this case.)

Back to the example in the opening of this section, we observe that the system depicted is capable of generating, for instance, the string $s = (ax)^\omega$, which is unfair to b and c, no matter how large a bound is chosen. We conclude that the system is not strongly fair to $\{(a, N_a), (b, N_b), (c, N_c)\}$ for any numbers N_a, N_b, and N_c.

Thus, using a fair scheduler or the technique learned in the previous chapter, the system has to be restricted properly so that bounded fairness may be achieved. Depending on the bounds selected, we may be left with multiple strings, a single type of string, or no string at all. Here we will present examples of all the categories mentioned. In all examples given, we assume that $\mathcal{T} = \{a, b, c\}$, and we project out '$x$', '$y$', and '$z$' from our strings for clarity.

- Let $N_a = N_b = N_c = 3$. The string $s = (abc)^\omega$ is strongly fair to $\{(a, 2), (b, 2), (c, 2)\}$ and can therefore be accepted by a fair scheduler that guarantees fairness[2] to

[2] From this point on fairness in a strong sense is understood.

$\{(a,3),(b,3),(c,3)\}$. Also acceptable is the sequence:

$$s = (abbc)^{\omega}$$

as it is fair to $\{(a,3),(b,2),(c,3)\}$.

- $N_a = 3$, $N_b = 1$, and $N_c = 3$. Since $N_b = 1$, b's must occur at every other position. Suppose b's are placed at even positions. If, say, two a's occupy two consecutive odd positions, then the distance between the nearest c's will exceed three. Thus, we are forced to place 'a' and 'c' alternately at odd positions in order to meet the bound on the distance between their occurrences. The only acceptable strings are:

$$s = (babc)^{\omega} \quad \text{or} \quad s = c(babc)^{\omega} \quad \text{or} \quad s = bc(babc)^{\omega} \quad \text{or} \quad s = abc(babc)^{\omega}$$

  Since all the above strings differ only in a finite prefix we say that the schedule contains only one type of string. Thus, in this case the schedule is tight, and allows only one genre of strings to be generated. We call such a schedule *slack-free*.

- $N_a = N_b = 1$, $N_c = 3$. Neither a's nor b's can be separated by more than one symbol, so we place them on, say, even and odd positions, respectively. That leaves no room for scheduling 'c', i.e. $N_c = \infty$. Thus the schedule is empty, or *unachievable*.

It seems that a schedule becomes harder to meet as one attempts to decrease its bounds, a fact that is manifested by the smaller class of accepted strings in the above example. Slack-free schedules are the limit of how far a schedule can be improved in one direction (which is another way of saying how much a specific bound can be decreased). For instance, in the direction of N_b, the schedule $(N_a, N_b, N_c) = (3, 3, 3)$ can be improved at most to the slack-free schedule $(N_a, N_b, N_c) = (3, 1, 3)$.

To see a somewhat peculiar property of slack-free schedules, consider two schedules, $(2, 2, 2)$ and $(3, 1, 3)$. They both are slack-free, as neither can be improved any further in any direction without sacrificing achievability. But in a sense $(2, 2, 2)$ treats the components "more fairly" overall: an attempt to decrease the second component of $(2, 2, 2)$ by one is offset by an increase by one of *both* N_a and N_c, resulting in the second schedule $(3, 1, 3)$. In other words, assigning some agents higher priorities comes at the cost of assigning lower priorities to an even larger number of agents.

In what follows we compare slack-free schedules and seek an optimal element.

7.3 Comparing slack-free schedules

To simplify the argument we assume that $\Sigma = \mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_M\}$. Fix $s \in \Sigma^\omega$. For a symbol $\tau_i \in \Sigma$ and $j \in \mathbb{N}^+$ define d_i^j to be the distance between the $(j-1)^{\text{th}}$ and the j^{th} occurrences of τ_i. For $j = 1$ the distance is simply measured from the beginning of the string. When an alphabet is made up of distinct symbols with no subscript, we subscript d with the symbol name instead.

When τ_i appears infinitely often in s define:

$$d_i := \limsup_j d_i^j$$

Otherwise, if τ_i does not appear infinitely often in s, $d_i := \infty$. Finally, let $d := \sum_{i=1}^{M} d_i$ denote the sum of all distances, which we also refer to as the *cumulative distance* of all symbols.

s	a	a	b	b	b	b	a	b	b	a	a	b	a	a	b	a	a	$\dots$
d_a^j	0	0					4			2	0		1	0		1	0	$d_a = 4$
d_b^j			2	0	0	0		1	0			2			2			$d_b = 2$

Table 7.1: An example of how distances are measured.

Example 12 Suppose $\Sigma = \{a, b\}$. If:

$$s = aabbbbabbaabaabaa\dots$$

then the distances are calculated as in Table 7.1. It follows that $d = d_a + d_b = 6$. □

Note that d_i can be infinite even when the i^{th} symbol occurs infinitely often. As an example, consider:

$$s = ababbabbbabbbba\dots$$

Observe that 'a' appears infinitely often in the sequence. Nonetheless, the distances $\{d_a^j\}_{j=1}^{\infty}$ form a diverging sequence $\{0, 1, 2, \dots\}$ and therefore $d_a = \infty$.

For any finite string $t \in \Sigma^*$ denote:

$$n_i := \text{the number of times } \tau_i \text{ appears in } t.$$

Denote the length of t by n and let us assume that t ends with τ_i. Since d_i^j is equal to the number of times symbols other than τ_i occur between the $(j-1)^{\text{th}}$ and the j^{th} occurrences of τ_i, it is immediate that:

$$n = n_i + \sum_{j=1}^{n_i} d_i^j$$

.

Example 13 Back to Table 7.1, let prefix t be the initial segment of s before the dots. Observe that t ends with 'a', $|t| = 17$, $n_a = 9$, and we have:

$$n_a + \sum_{j=1}^{9} d_a^j = 9 + (0 + 0 + 4 + 2 + 0 + 1 + 0 + 1 + 0) = 9 + 8 = 17$$

$\square$

For $i = 1, \ldots, M$ let $N_i \in \mathbb{N}$. We call the tuple $\mathcal{N} := (N_i)_{i=1}^M$ a *schedule*. We say a string s with distances $\{d_i\}_{i=1}^M$ is accepted by a schedule $\mathcal{N} = (N_i)_{i=1}^M$ if $d_i \leq N_i$ for all i. The language of a schedule is defined to be the set of all infinite strings that it accepts, i.e.:

$$L(\mathcal{N}) := \left\{ s \in \Sigma^\omega \mid s \text{ is accepted by } (N_i)_{i=1}^M \right\}.$$

Using the technique of the previous chapter we can algorithmically construct an automaton generating $L(\mathcal{N})$ as follows. We start off with a plant model that is simply a single-state automaton which has M selfloops labeled with $\{\tau_1, \ldots, \tau_M\}$. Then $L(\mathcal{N})$ is the supremal sublanguage of the plant that is strongly fair to $\{(\tau_i, N_i)\}_{i=1}^M$.

A schedule is called *achievable* if its language is nonempty. Henceforth in our discussions we require that all schedules be achievable.

We define the length of a schedule $\mathcal{N} = (N_i)_{i=1}^M$ to be $||\mathcal{N}|| := \sum_{i=1}^M N_i$. We call a schedule $\mathcal{N} = (N_i)_{i=1}^M$ *slack-free* if there does not exist an achievable schedule $\mathcal{N}' = (N_i')_{i=1}^M$ and a position $1 \leq k \leq M$ such that:

$$||\mathcal{N}'|| < ||\mathcal{N}|| \quad \wedge \quad (\forall j.\ 1 \leq j \leq M \wedge j \neq k \rightarrow N_j = N'j)$$

Thus a schedule is slack-free if it cannot be improved in any direction. We call an achievable schedule *optimal* if it has minimal length. From the definition it follows that an optimal schedule has to be slack-free, but the converse is not true in general.

An optimal schedule, in addition, would allow only those strings whose cumulative distance is minimal. Going back to the discussion of Page 132, we observe that the slack-free schedule $(3, 1, 3)$ cannot be optimal since

$$6 = ||(2, 2, 2)|| < ||(3, 1, 3)|| = 7.$$

This is despite the fact that the schedule $(3, 1, 3)$ treats 'b' more fairly than $(2, 2, 2)$ does. Overall, the latter treats all three symbols more fairly. In fact, as we will see shortly, the $(2, 2, 2)$ schedule is optimal.

The next few pages are dedicated to finding the minimum cumulative distance over all infinite strings. Following that we present one type of infinite string for which d is minimal. A schedule that contains only the aforementioned type is then optimal.

The following result claims that for any string the cumulative distance is bounded from below by $M(M - 1)$. We need the lemma stated next before we can proceed.

Lemma 7.1 *Let $\{a_i\}_{i=1}^M$ be a sequence of non-negative numbers. We have:*

$$\sum_{i=1}^{M} a_i \sum_{i=1}^{M} \frac{1}{a_i} \geq M^2$$

Proof: We have:

$$\begin{aligned}
\sum_{i=1}^{M} a_i \sum_{j=1}^{M} \frac{1}{a_j} &= \sum_{i,j=1}^{M} \frac{a_i}{a_j} \\
&= \sum_{i=j=1}^{M} \frac{a_i}{a_j} + \sum_{1=j<i}^{M} \frac{a_i}{a_j} + \sum_{1=i<j}^{M} \frac{a_i}{a_j} \\
&= M + \sum_{1=j<i}^{M} \frac{a_i}{a_j} + \sum_{1=i<j}^{M} \frac{a_i}{a_j} \\
&= M^2 - (M^2 - M) + \sum_{1=i<j}^{M} \frac{a_j}{a_i} + \sum_{1=i<j}^{M} \frac{a_i}{a_j} \\
&= M^2 + \sum_{1=i<j}^{M} (\frac{a_j}{a_i} + \frac{a_i}{a_j} - 2) \\
&= M^2 + \sum_{1=i<j}^{M} a_i a_j (\frac{1}{a_i} - \frac{1}{a_j})^2 \\
&\geq M^2
\end{aligned}$$

which is desired. ■

Now we state the main result of this section.

Theorem 7.1 *For any infinite string s, the cumulative distance is bounded by $M(M-1)$ from below, i.e.:*

$$d \geq M(M-1).$$

Proof:[3] We consider the non-trivial case where d is finite. Let t be any prefix of s that contains at least $M(M-1)$ instances of each symbol. We let n_i denote the number of times τ_i appears in t. Thus, we have that $n_i \geq M(M-1)$ and $n = \sum_{i=1}^{M} n_i$.

[3]Contributed by Professor Alfred B. Lehman, Department of Computer Science, University of Toronto.

By definition we have:

$$\begin{aligned} d_i &= \max_j d_i^j \\ &\geq \frac{\sum_{j=1}^{n_i} d_i^j}{n_i} \qquad \text{(Since maximum is not smaller than the average)} \end{aligned}$$

Let τ_k be the last symbol of t. Then as illustrated in Figure 7.3 we have:

$$\begin{aligned} \sum_{j=1}^{n_i} d_i^j &= n - n_i \quad &;\quad i = k \\ \sum_{j=1}^{n_i} d_i^j &> n - n_i - d_i^{n_i+1} \quad &;\quad i \neq k \end{aligned} \tag{7.1}$$

Since $d_i^{n_i+1} \leq d_i$ it follows that $n - n_i - d_i^{n_i+1} \geq n - n_i - d_i$. Also, we have that $n - n_i > n - n_i - d_i$. Therefore (7.1) implies that:

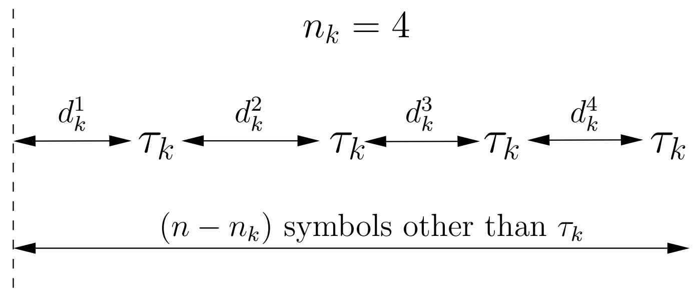
$$\text{for all } i: \quad \sum_{j=1}^{n_i} d_i^j > n - n_i - d_i$$

We have:

$$\begin{aligned} d_i &\geq \frac{\sum_{j=1}^{n_i} d_i^j}{n_i} \\ &> \frac{n - n_i - d_i}{n_i} \end{aligned}$$

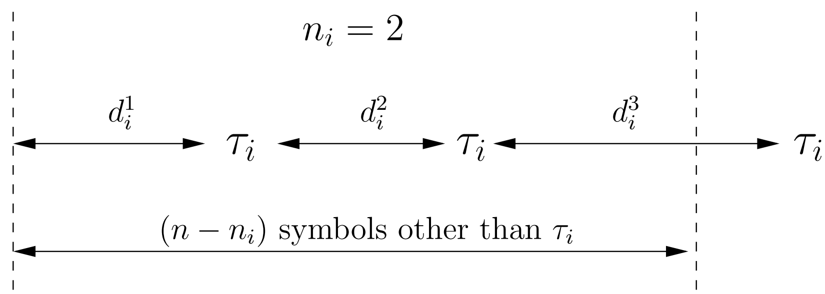
which simplifies to:

$$d_i > \frac{n - n_i}{n_i + 1}$$



If τ_k is the last symbol we have:

$$\sum_{j=1}^{n_k} d_k^j = n - n_k$$



For all other symbols $\tau_i,\ i \neq k,$ we have:

$$\sum_{j=1}^{n_i+1} d_i^j > n - n_i$$

Figure 7.3: Illustration of formulas (7.1).

It follows from the above that:

$$\begin{aligned}
d &= \sum_{i=1}^{M} d_i \\
&> \sum_{i=1}^{M} \frac{n-n_i}{n_i+1} \\
&= -M + (1+n)\sum_{i=1}^{M} \frac{1}{n_i+1} \\
&= -M + (1+\sum_{i=1}^{M} n_i)\sum_{i=1}^{M} \frac{1}{n_i+1} \\
&= -M + \Big(1 - M + \sum_{i=1}^{M}(n_i+1)\Big)\sum_{i=1}^{M} \frac{1}{n_i+1} \\
&\geq -M + M^2 - (M-1)\sum_{i=1}^{M} \frac{1}{n_i+1} && \text{(By Lemma 7.1)}
\end{aligned}$$

We conclude that:

$$d > M(M-1) - (M-1)\sum_{i=1}^{M} \frac{1}{n_i+1} \tag{7.2}$$

But $n_i \geq M(M-1)$ implies $n_i + 1 > M(M-1)$, and therefore:

$$(M-1)\sum_{i=1}^{M} \frac{1}{n_i+1} < (M-1)\sum_{i=1}^{M} \frac{1}{M(M-1)} = 1 \tag{7.3}$$

Since both d and $M(M-1)$ are integers, it follows from (7.2) and (7.3) that $d \geq M(M-1)$, as desired. ■

Thus, the cumulative distance cannot be any shorter than $M(M-1)$. At this limit, an attempt to decrease the distance between successive occurrences of a symbol is countered by a *larger* increase of distance between those of others.

At this point it is natural to ask whether any string with $d = M(M-1)$ exists. Let $p \in \Sigma^M$ be a permutation of Σ, i.e. each symbol appears exactly once in p. Further, we let $s = p^\omega$. So for instance if $\Sigma = \{a, b, c\}$ then we could select $p = bac$

and $s = (bac)^{\omega}$.

It can be readily verified that the above infinite string is accepted by the $(M-1)_{i=1}^{M}$ schedule. Furthermore, no achievable schedule $\mathcal{N}'$ with $||\mathcal{N}'|| < ||(M-1)_{i=1}^{M}|| = M(M-1)$ could exist, for otherwise if s is accepted by $\mathcal{N}' = (N_i')_{i=1}^{M}$ we must have:

$$\begin{aligned} & \forall i.\ d_i \leq N_i' \\ \Rightarrow\ & d \leq ||\mathcal{N}'|| < M(M-1) \end{aligned}$$

which cannot be the case by the above theorem. Therefore we have the following corollary of the theorem.

Corollary 7.1 *The $(M-1)_{i=1}^{M}$ schedule is optimal.*

7.4 More on the optimality of $s = p^{\omega}$

We have seen that the string $s = p^{\omega}$ is optimal in the sense that it has the shortest cumulative distance. In this section we show that it possesses the shortest maximum distance among all symbols as well.

Define $d_{\max}$ to be the maximum distance between any two successive occurrences of any symbol, i.e.:

$$d_{\max} = \max_i d_i = \max_i \limsup_j d_i^j$$

We observe that for $s = p^{\omega}$ we have $d_{\max} = M - 1$. The following simple result shows that for all other infinite strings, the maximum distance is greater than $M-1$. Thus, we always have $d_{\max} \geq M - 1$, with the minimum being achieved if and only if $s = p^{\omega}$ for some permutation p of $\mathcal{T}$. Before stating the result we will present another characterization of $s = p^{\omega}$.

Lemma 7.2 *A string $s \in \mathcal{T}^{\omega}$ is of the form p^{ω} for some permutation p of $\mathcal{T}$ if and* *only if in any sliding window of length M over s every symbol of $\mathcal{T}$ appears exactly once.*

Proof: In the proof we denote by $s[m]$ the m^{th} symbol of s, and by $s[m;n]$ the substring $s[m]. \ldots .s[n]$ of s, $m, n \in \mathbb{N}$ and $m \leq n$. The result in one direction is obvious. Suppose every symbol appears exactly once in any sliding window of length M. Let p denote the prefix of s of length M, i.e. $p := s[0; M-1]$. It follows from the assumption that p must contain all symbols. Without loss of generality we assume that $p = \tau_0 \ldots \tau_{M-1}$. By a simple inductive argument we show that:

$$\forall n \in \mathbb{N}.\ s[n] = \tau_{(n \mod M)}$$

and therefore $s = p^{\omega}$.

- Base. For $n = 0, \ldots, M-1$ the result trivially holds.

- Inductive step. For $n \geq M$ suppose the result holds for all $0 \leq k < n$. By the induction assumption the first symbol of the substring $s[n-M; n-1]$ is given by:

  $$s[n-M] = \tau_{(n-M \mod M)} = \tau_{(n \mod M)}$$

  Since $\tau_{(n \mod M)}$ appears exactly once in $s[n-M; n-1]$ it follows that it is distinct from all other $M-1$ symbols. Slide the window one unit to the right to obtain the substring $s[n-M+1; n]$. Since $\tau_{(n \mod M)}$ has to appear in the substring and is distinct from its first $M-1$ symbols we must have:

  $$s[n] = \tau_{(n \mod M)}$$

■

And now we state the main result.

Theorem 7.2 *The maximum distance is greater than or equal to* $M - 1$*, and the minimum is achieved if and only if* $s = p^\omega$ *for some permutation* p *of* $\mathcal{T}$*.*

Proof: Let s be a string not of the form p^ω. Then there exists some window of length M of s and some symbol $\tau_i \in \mathcal{T}$ which does not appear in the window. But then the distance between the two successive occurrences of τ_i located just before and after the window is at least M. Therefore:

$$d_{max} \geq d_i \geq M$$

which is desired. ■

Hence, the string $s = p^\omega$, or equivalently the schedule $(M-1)_{i=1}^{M}$, has the following important properties:

1. It treats everyone equally,
2. Among all schedules it provides service to all agents in shortest time, and
3. Maximum waiting among all agents is minimal.

7.5 The schedule generated by a queue

In order to crystallize the connection with optimal schedules a few changes are due in our typical representation of a queue. A queue grants service in the order that requests have been made. Thus we need to take into account for each process the request and access of a (shared) resource. Note that there is no need to model the release event. Figure 7.4 shows the complete model for two processes **A** and **B**. We label the queue "$[1, 1]$" as each process is assigned a buffer of capacity one, meaning that the queue remembers only one request per process.

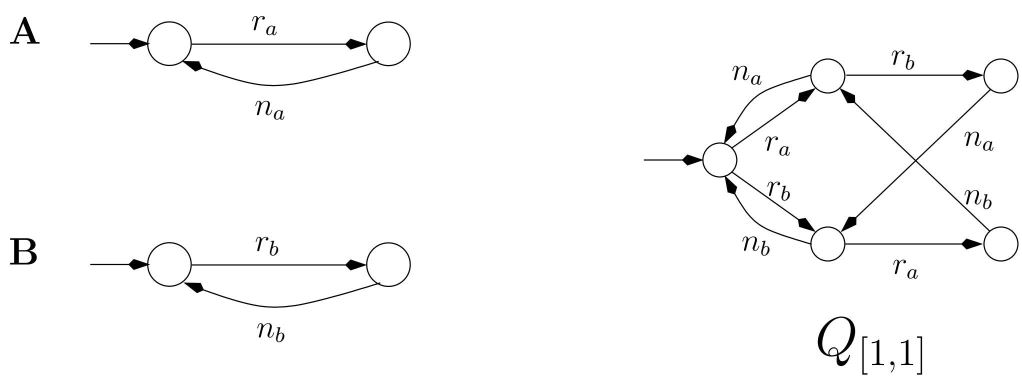


Figure 7.4: Processes **A** and **B**, together with $Q_{[1,1]}$.

As evident from the figure, the queue does not intervene when one process has not requested the resource yet. That means there is no restriction on the number of times **A** can execute its loop while **B** remains idle at its initial state. Thus, the queue comes into play only when there are active requests from both sides, in other words, when it is *busy*. Henceforth, in our analysis of queues we assume that they are busy. In a busy queue a request event follows shortly after an access event, and the two are considered as a single entity. To represent this, we may regard the string "access-request" as one meta-symbol. We obtain the busy queue of Figure 7.5 upon denoting $a := n_a r_a$ and $b = n_b r_b$.

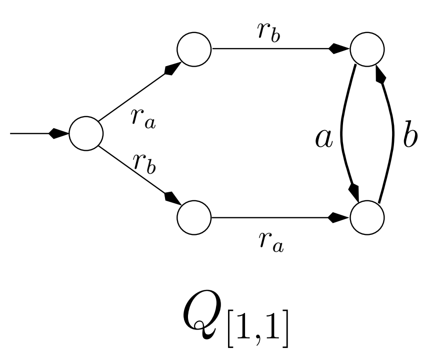


Figure 7.5: Busy $Q_{[1,1]}$, where $a := n_a r_a$ and $b = n_b r_b$.

In obtaining the above representation we have further assumed that access to the resource is granted only when it is requested by all processes (perhaps because access events are much slower than request events, and can be taken only when no additional request is possible). In addition, the requests are represented by thin lines, hinting that they will be projected out whenever sequences over the alphabet $\{a, b\}$ are concerned.

7.5.1 The infinite language generated by a busy queue

With request events projected out, the (busy) queue of Figure 7.5 is capable of generating the infinite strings $(ab)^\omega$ and $(ba)^\omega$, depending on where the infinite loop is entered. If we denote by $\mathcal{L}_{[1,1]}$ the infinite language generated by the queue we will have:

$$\mathcal{L}_{[1,1]} = (\epsilon + b)(ab)^\omega$$

which is incidentally equal to $L(1, 1)$, the language generated by the optimal schedule $(1, 1)$.

Note that the methods of this work can be employed to construct an automaton generating $L(1, 1)$. We start off with a simple plant model, consisting of a single state, self-looped with $\{a, b\}$ transitions. Then we compute the optimal design that is strongly fair to $\{(a, 1), (b, 1)\}$, as carried out in Figure 7.6.

The observation made above is generalized to an alphabet of arbitrary size in the following theorem.

Theorem 7.3 *For $M \in \mathbb{N}^+$ let $\Sigma := \{r_i, n_i\}_{i=1}^M$ be the set of request and access events for M processes. Define the meta-symbols $a_i := n_i r_i$ and let $\mathcal{T} := \{a_i\}_{i=1}^M$. Then the infinite language generated by the queue $Q_{[1]_{i=1}^M}$ is equal to that generated by the optimal schedule $(M-1)_{i=1}^M$ for $\mathcal{T}$, that is:*

$$\mathcal{L}_{[1]_{i=1}^M} = L(M-1)_{i=1}^M$$

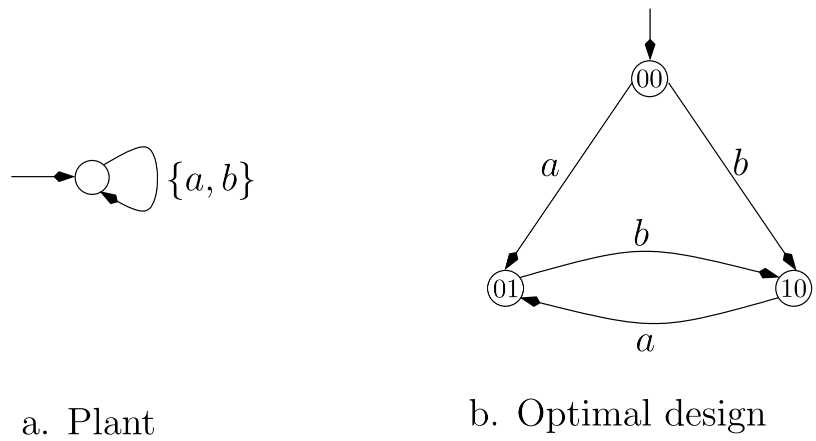


Figure 7.6: (a) Plant (b) Optimal strongly fair design to $\{(a, 1), (b, 1)\}$.

Proof: Consider a sequence of M distinct requests $r_{i_1} r_{i_2} \ldots r_{i_M}$, where $\{i_1, i_2, \ldots, i_M\}$ is a permutation of $\{1, 2, \ldots, M\}$. Then the only choice offered by the queue for executing next is a_{i_1}, which updates the sequence of requests to $r_{i_2} r_{i_3} \ldots r_{i_M} r_{i_1}$. At this point only a_{i_2} becomes available for execution and so forth. Thus, the ensuing infinite sequence is $(a_{i_1} a_{i_2} \cdots a_{i_M})^\omega$. The language generated by the queue therefore is:

$$\mathcal{L}_{[1]_{i=1}^M} = \left\{ p^\omega | \; p \text{ is a permutation of } \mathcal{T} \right\} = L(M-1)_{i=1}^M$$

■

7.5.2 Queues with higher buffer capacities

The queue of the previous section can store only up to one request per process. In this section we study—through an example—queues with larger buffer capacities.

Before we can proceed a small technicality must be dealt with. In our process models, only one active request per agent is permitted, and it is not defined in the model to, say, access the resource after three requests have been made. We get around this problem by allowing a model to do arbitrary numbers of requests and accesses in no particular order. The queue itself will ensure that there is a 1-1 match between requests and accesses, and that they appear in the right order. Henceforth the automaton of Figure 7.7 will be used to model a *team*, a term we use hereafter in place of a process, to convey the idea that a team may consist of several players.

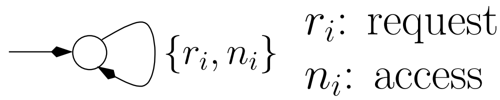


Figure 7.7: Team i.

As before, the meta-symbol 'a_i' abbreviates $n_i r_i$. Observe that we do not distinguish between the players of a team; for example, n_i models the access event of *any* player in team i.

The capacity of a queue of M teams is denoted by $[m_i]_{i=1}^M$, where $m_i \in \mathbb{N}^+$ is the number of requests from team i that can be stored. Equivalently, one may assume as before that the queue stores one request per player, where for each i, team i is now made up of m_i players. Either way, whenever team i gets its turn, service is granted m_i-times (no matter how it is distributed among its players), after which it has to wait $\sum_{i\neq j=1}^M m_j$ times for its turn to come up again. The following result is then immediate.

Theorem 7.4 *The $[m_i]_{i=1}^M$ queue is strongly fair to $\{(a_i, \sum_{i\neq j=1}^M m_j)\}_{i=1}^M$.*

The distinction between teams and players disappears when $m_i = 1$ for all i, in which case we have the following corollary.

Corollary 7.2 *The $[1]_{i=1}^M$ queue is strongly fair to $\{(a_i, M-1)\}_{i=1}^M$.*

Proof: Simply note that:

$$\sum_{i\neq j=1}^{M} m_j = \sum_{i\neq j=1}^{M} 1 = M - 1.$$

■

In the following example, we will investigate a queue of capacity [2, 1].

Example 14 Consider two teams **A** and **B** competing to access a shared resource. Assume that the buffer capacity is 2 for **A** and 1 for **B**. Thus, in addition to maintaining the order, the queue $Q_{[2,1]}$ remembers up to two unprocessed requests for **A** and one for **B**. Figure 7.8 shows the queue in normal operating condition and under stress. It is seen that the infinite loop $(a^2b)^\omega$ allows entry at three different states,

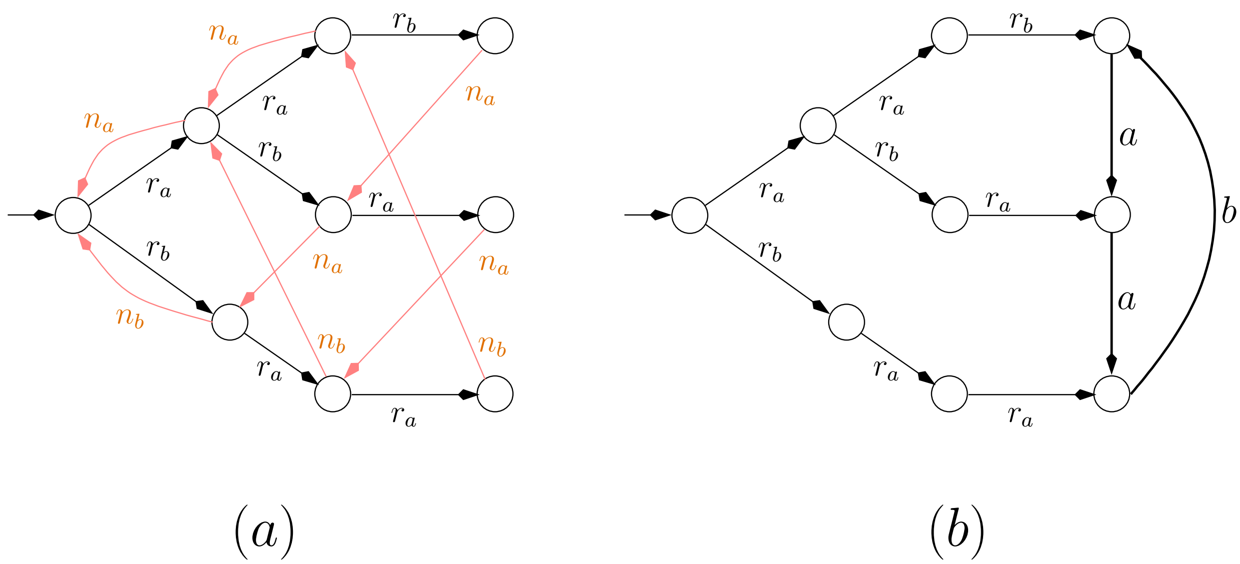


Figure 7.8: (a) $Q_{[2,1]}$ (b) Busy $Q_{[2,1]}$.

and the ω-language generated by the queue therefore is:

$$\mathcal{L}_{[2,1]} = (\epsilon + b + ab)(a^2b)^\omega$$

In general, the number of distinct strings p of length $(m_1 + m_2 + \cdots + m_M)$ that can be formed using m_i instances of the symbol a_i, $i = 1, 2, \ldots, M$, is equal to:

$$\left(\begin{array}{c} m_1 + m_2 + \cdots + m_M \\ m_1, m_2, \ldots, m_M \end{array} \right) = \frac{(m_1 + m_2 + \cdots + m_M)!}{m_1! m_2! \cdots m_M!}$$

which is exactly equal to the number of distinct infinite strings p^ω that $Q_{[m_i]_{i=1}^M}$ can generate at its various *loop* states, i.e. those states out of which no further request is possible.

Theorem 7.5 *The infinite language of $Q_{[m_i]_{i=1}^M}$ is equal to:*

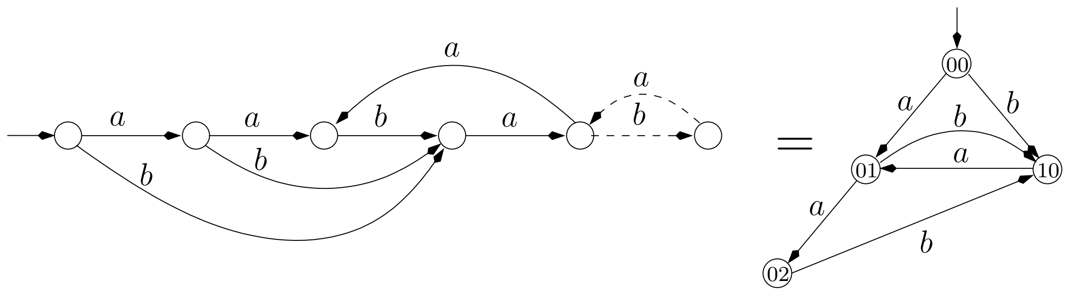
$$\mathcal{L}_{[m_i]_{i=1}^M} = \left\{ p^\omega | p \textit{ contains exactly } m_i \textit{ instances of } a_i,\ i = 1, 2, \ldots, M \right\}$$

Returning to our example, we observe that by Theorem 7.4 the queue is strongly fair to $\{(a, m_b), (b, m_a)\} = \{(a, 1), (b, 2)\}$. But is it supremal? In other words, does its language equal that generated by the (1,2) schedule:

$$\mathcal{L}_{[2,1]} \stackrel{?}{=} L(1, 2)$$

As illustrated in Figure 7.9, it happens that $\mathcal{L}_{[2,1]} \subsetneq L(1, 2)$. For instance, the string $(ab)^\omega$ is in $L(1, 2)$ but not in $\mathcal{L}_{[2,1]}$.

A careful taxonomy of strings in $L(1, 2) - \mathcal{L}_{[2,1]}$ reveals that although the distance between successive instances of a symbol agrees with the $(1, 2)$ schedule, the *second*-distance between instances of 'a' always exceeds 2. Consider, for example, the infinite string $s = (ab)^\omega \in L(1, 2) - \mathcal{L}_{[2,1]}$. The first-distance of (a, b) is equal to $(1, 1)$, and therefore s can be generated by the $(1, 2)$ schedule. But the maximum distance between an instance of 'a' and its second-next instance is equal to 3 as illustrated in Figure 7.10. In contrast, consider any infinite string, say $ab(a^2b)^\omega$, generated by the queue. It can be seen from Figure 7.11 that the 2nd-distance of 'a' (formally defined



$\mathcal{L}_{[2,1]}$ (solid lines) $L(1,2)$

Figure 7.9: Illustrating $\mathcal{L}_{[2,1]} \subsetneq L(1,2)$. The languages become equal when the dashed lines are added to the left automaton.

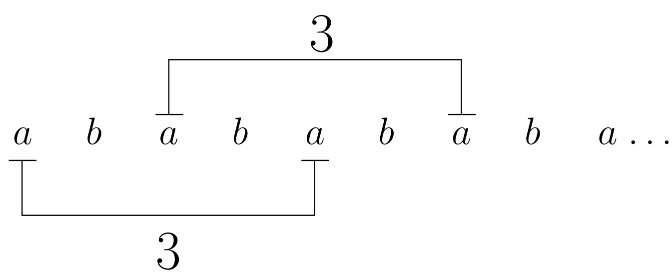


Figure 7.10: The second-distance of 'a' is equal to 3. Note that the distance between two symbols equals the number of symbols separating them.

below) is equal to 2. Before we generalize this observation we need the following definition.

Definition 7.2 *Let s be an infinite string. The m^{th}-distance of 'a' is the maximum distance, over all positions where 'a' appears, to the m^{th} subsequent instance of 'a'.*

Thus, the distance measure we have been using so far is simply a first-distance. A first-distance shows how frequently a player appears in a sequence. Similarly, a higher order m^{th}-distance indicates how frequently a team of m players appears in a sequence.

The following result is of paramount importance in hierarchical design of queues, discussed in the next section.

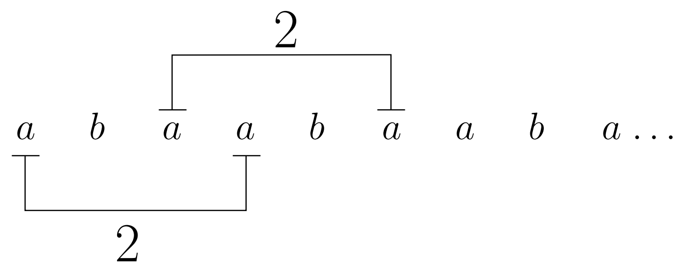


Figure 7.11: The second-distance of ‘a’ is equal to 2.

Theorem 7.6 *For $Q_{[m_i]_{i=1}^{M}}$, the m_i^{th}-distance of a_i is equal to $\sum_{j=1}^{M} m_j - 1$ for all i.*

Proof. We recall that in each turn team i is served exactly m_i-times, after which it has to wait ($\sum_{i\neq j=1}^{M} m_j$)-times for its next turn. On the other hand, the m_i^{th}-distance is measured over a closed interval consisting of (m_i+1) instances of a_i, and therefore the 0^{th} and the m_i^{th} instances are in two consecutive turns. (cf. Figure 7.12.) The

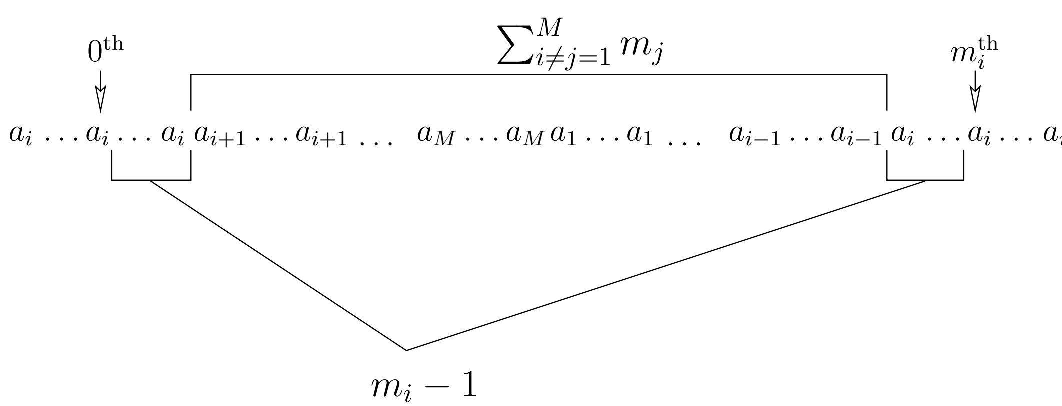


Figure 7.12: Illustration of Theorem 7.6.

distance between them is equal to:

$$\sum_{i\neq j=1}^{M} m_j + m_i - 1 = \sum_{j=1}^{M} m_j - 1.$$

■

In the $[2,1]$ queue of our running example, the second-distance of 'a' is equal to:

$$2+1-1=2.$$

Note that if the m_i^{th}-distance is bounded by k, the 1^{st}-distance will be bounded by $k-m_i+1$, i.e. the following corollary of the theorem, which is an alternative statement of Theorem 7.4.

Corollary 7.3 *For $Q_{[m_i]_{i=1}^M}$, the (first-)distance of a_i is bounded by $\sum_{i\neq j=1}^M m_j$.*

This fact is useful when generating the infinite language of queues algorithmically, as illustrated in the following example.

Example 15 We would like to construct a generator for the infinite language of $Q_{[2,1]}$. For the symbol 'b' we require as before that the first-distance be bounded by $2+1-1=2$. Since a buffer capacity of 2 is selected for 'a', we require that the second-distance of 'a' be bounded by 2. We need not worry about limiting the first-distance of 'a' to one: by the above corollary this follows from the bound placed on the second-distance.

To compute an optimal design satisfying a given bound N on the m_i^{th}-distance we employ m_i integer variables counting up to N. Each variable is reset to zero at the initial state. All counters are incremented by one and up to N with each a_j transition where $j\neq i$. a_i transitions increment exactly one counter in a fixed-order, say from left to right.

For this example, we need 2 counter variables for 'a'—for convenience called *left* and *right* counters—and one for 'b', all counting up to $2+1-1=2$. Each state is labeled with the values of its counters, which are from left to right a's left counter, a's right counter, and b's counter, respectively. The 'b'-counter is incremented with each 'a', and is reset by a 'b'. Both 'a'-counters are incremented by a 'b'. In addition, starting with the left counter, 'a' transitions increment one and reset the other counter alternately. The entire procedure is carried out in Figure 7.13. Note that in the end it is necessary to trim the automaton of terminal states since we are only interested in infinite sequences. □

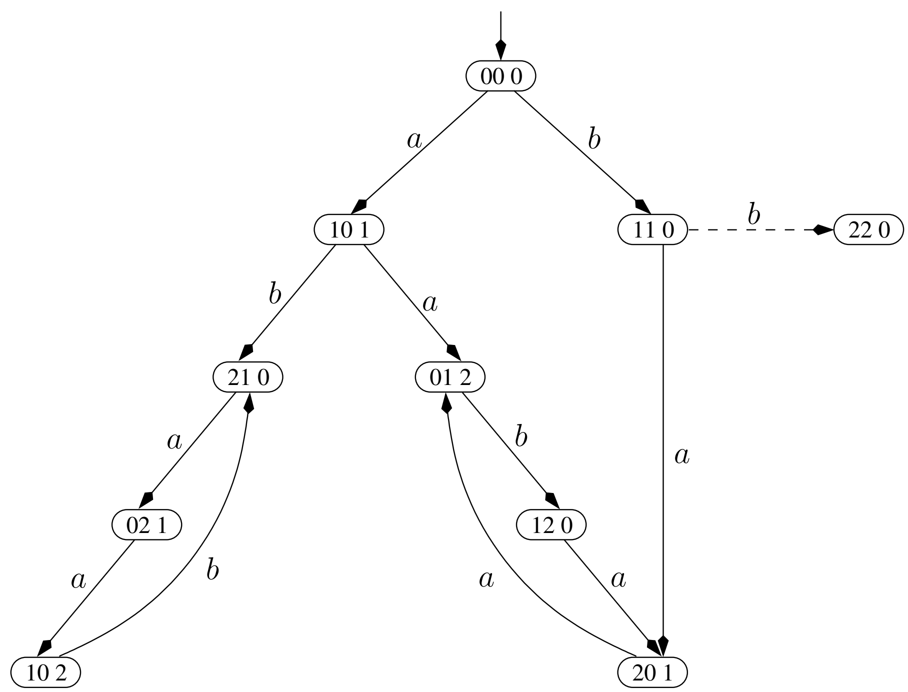


Figure 7.13: Computing the infinite language $\mathcal{L}_{[2,1]}$. The terminal state "22 0" is trimmed in the end.

7.6 Hierarchical design of queues

Suppose we would like to design $Q_{[m_1,m_2,m_3]}$, that is, a queue for three teams '1', '2', and '3', of capacities m_1, m_2, and m_3, respectively. This task can be broken down into two steps. First, we design a queue to account for the fact that the players of team 3 will have to wait $(m_1 + m_2)$-times for the players of the other two teams. Second, we design a queue of capacity $[m_1, m_2]$ to maintain order between teams 1 and 2, respectively. This observation is stated formally below.

Theorem 7.7 *For any* $m_1, m_2, m_3 \in \mathbb{N}^+$ *we have:*

$$Q_{[m_1,m_2,m_3]} \;=\; Q_{[m_1+m_2,m_3]} \;||\; Q_{[m_1,m_2]}$$

Remark 1: Note that transitions pertaining to the first component in $Q_{[m_1+m_2,m_3]}$ (with $m_1 + m_2$ players) are labeled with both their counterparts from teams 1 and 2: thus, two transitions with the same source and destination, labeled with r_1 and r_2, constitute a request for the first component.

Remark 2: The equality remains valid if any number of teams (with given buffer capacities) are added to both sides, e.g.:

$$Q_{[m_1,m_2,m_3,m_4]} \;=\; Q_{[m_1+m_2,m_3,m_4]} \;||\; Q_{[m_1,m_2]}$$

Proof. Note that the second queue to the right never rejects n_3 as the next choice. If r_3 is the first active request, then n_3 is the only choice offered by the first queue to the right for executing next, which is agreeable by what the left-hand side queue offers. If, say, r_1 is the first active request, then either of n_1 or n_2 is offered by the first queue to the right, but only n_1 is offered by the second. The event n_1 will be the sole survivor for the next move, again in agreement with the selection of the left-hand side queue. That completes our argument. ■

Remark 3: The result is equally valid for busy queues.

One can verify that the number of loop states in both sides is the same:

$$\binom{m_1+m_2+m_3}{m_1, m_2, m_3} = \binom{m_1+m_2+m_3}{m_1+m_2, m_3}\binom{m_1+m_2}{m_1, m_2}$$

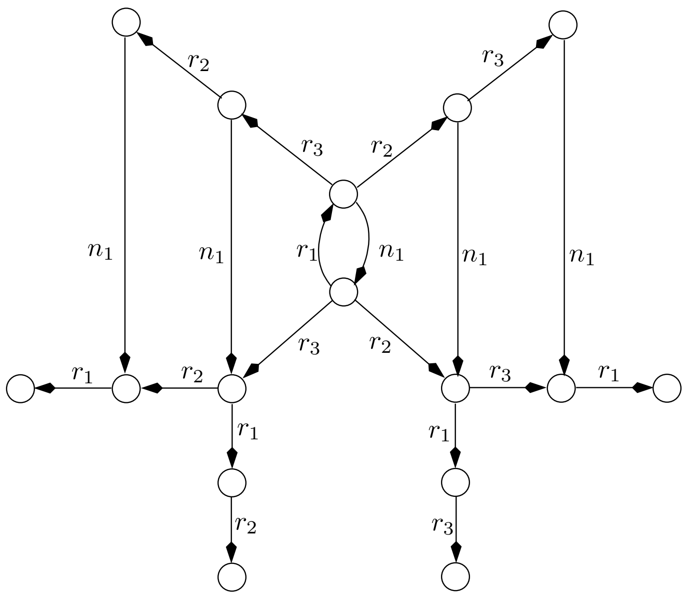


Figure 7.14: $Q_{[1,1,1]}$ has 16 states and 30 transitions. n_2 and n_3 transitions are not shown.

This yields a considerable reduction in the number of states required for hierarchical implementation, as in general:

$$\binom{m_1+m_2+m_3}{m_1+m_2, m_3} + \binom{m_1+m_2}{m_1, m_2} << \binom{m_1+m_2+m_3}{m_1+m_2, m_3}\binom{m_1+m_2}{m_1, m_2}$$

The next two examples illustrate the idea.

Example 16 Suppose we intend to design $Q_{[1,1,1]}$. A flat construction is carried out in Figure 7.14, where to avoid cluttering the figure only the access transitions of the first player are shown.

By applying the decomposition technique of this section we will obtain:

$$Q_{[1,1,1]} = Q_{[2,1]} || Q_{[1,1]}$$

i.e. we group together the first two players as a team, maintain fairness between the

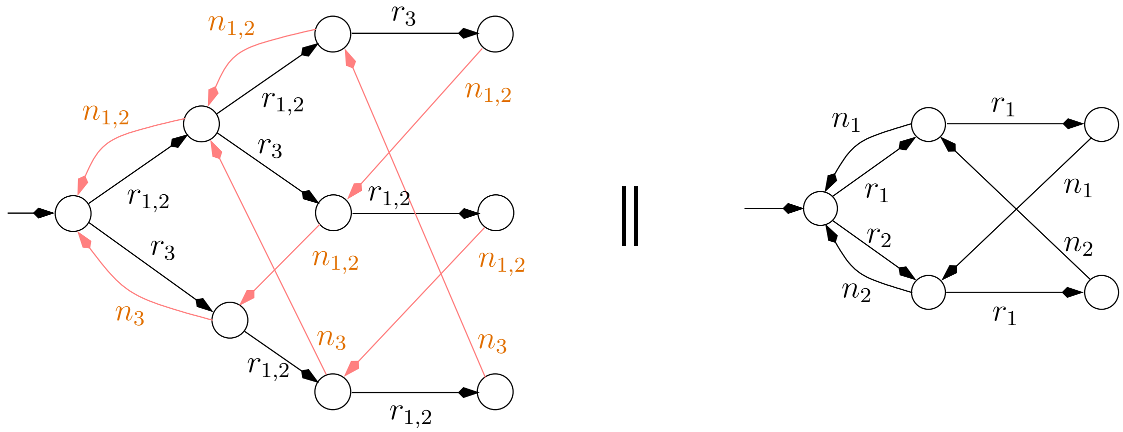


Figure 7.15: Hierarchical decomposition of $Q_{[1,1,1]}$ into $Q_{[2,1]}||Q_{[1,1]}$.

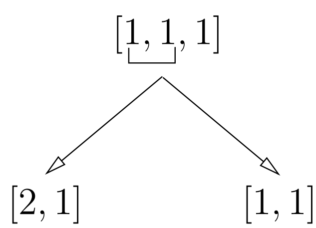


Figure 7.16: The decomposition tree for $Q_{[1,1,1]}$.

team just created and the third player, and finally ensure fairness among the players of the team. This is illustrated in Figure 7.15.

The tree of Figure 7.16 shows the steps of decomposition. At each level we show if a team is formed by grouping players together. Such a tree is called a *decomposition tree* of the queue.

Example 17 Two decomposition trees for $Q_{[1,1,1,1,1]}$ are shown in Figure 7.17.

Note that in the second decomposition a queue with 5! loop states is implemented using four queues with the cumulative number of loop states equal to $2 + 3 + 4 + 5 = 14$. This observation is generalized in the following corollary, which yields a notable reduction in the number of states required in a fair implementation.

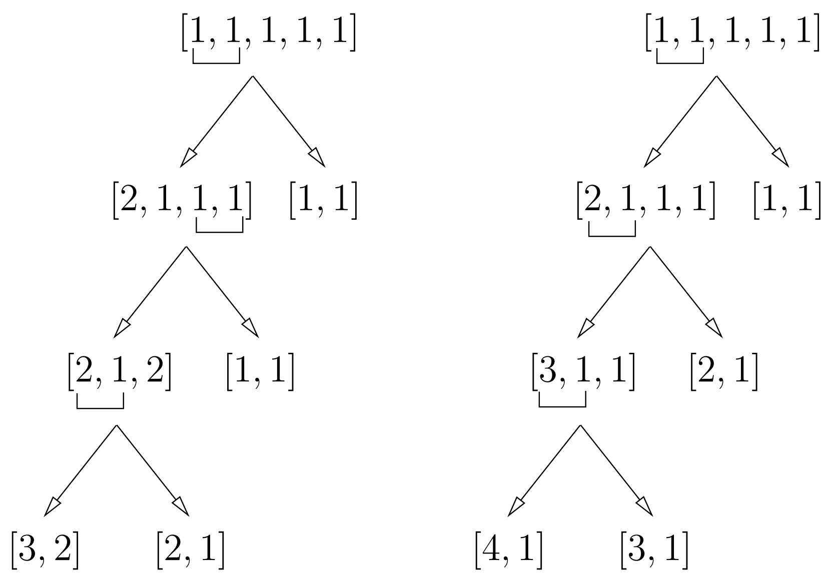


Figure 7.17: Two decomposition trees for $Q_{[1,1,1,1,1]}$.

Corollary 7.4 *For any integer $M \geq 2$ we have:*

$$Q_{[1]_{i=1}^{M}} \;=\; Q_{[1,1]}||Q_{[2,1]}||\cdots||Q_{[M-1,1]}.$$

Observe that if we measure the complexity of a queue by the number of its loop states, then the proposed technique for hierarchical design reduces the complexity from $M!$ for the flat case to $\frac{(M-1)(M+2)}{2}$. Thus by allowing a hierarchical design, queues provide an efficient method for establishing fairness among M processes.

Chapter 8

Conclusions and Future Work

8.1 Fairness in supervisory control

The absence of a universal agreement on the meaning of fairness motivated us to define two measures of fairness that find special relevance in supervisory control. Weak fairness is a property of a closed-loop system. A supervisor is weakly fair if and only if it does not continuously disable an event along any path. Weak fairness is also known as starvation freedom as a process may starve if a supervisor continuously disables the event of accessing a shared resource. One instance where the possibility of starvation exists is the Dining Philosophers problem. To optimally achieve mutual exclusion, the optimal supervisor does not intervene when the two neighbors of a philosopher collaborate to hold the shared resource indefinitely, thereby causing him to starve.

Weak fairness is achieved by exercising an additional level of control on the system. The new supervisor does not allow any behavior that may lead to starvation. Since starvation is a qualitative concept, for the purpose of design we need to be specific on how long the event of accessing a shared resource can be disabled before we proclaim that the process has starved. Thus we introduce a bounded version of the fairness problem, and we show that the class of boundedly fair sublanguages of a given language is closed under arbitrary union, and a supremal element exists as a result.

The weak fairness condition requires eventual enabledness of an event when it has been continuously disabled by a supervisor. However one has to remember that being enabled does not guarantee occurrence if the supervisor offers more than one event for the next choice. Then it is possible that one of the events in the enabled set malevolently steals the chance of occurrence from the others and is selected by the plant in every turn. Against such malevolence we need tighter control to ensure that not only is each event offered a fair chance for execution, but it will also be selected by the plant for the next move infinitely often. We call the new condition strong fairness. To implement strong fairness, a supervisor makes the event of accessing a resource the only enabled event until it is selected by the plant. As in weak fairness, we specify a bound on the number of times service to an event can be postponed while its competitors are served. Besides the practical appeal, placing a bound makes the strong fairness property robust with respect to arbitrary union, and therefore a supremal design exists.

Queues are known as a traditional remedy for the unfairness problem. Our analysis makes it clear that queues implement strong fairness; however, it must be noted that any weaker property, including weak fairness, is satisfied by the closed-loop system as well. Thus, implementing queues to achieve starvation freedom [Won02] is unnecessarily conservative.

Order of requests disregarded, we show that when each process is limited to one request the infinite language generated by a queue is optimal. The infinite language of a queue remains strongly fair but no longer optimal when we allow more than one request per process. But design with queues is proved to be very efficient as they are amenable to hierarchical construction. To design a queue for n processes, one can partition them into two teams of $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ processes. Then while two low-level queues maintain the order of service between the processes within each team, a high-level queue ensures that each team is served exactly $\lfloor \frac{n}{2} \rfloor$ times ($\lceil \frac{n}{2} \rceil$ times when the second team is served) before the other team is served. This results in an exponential reduction in the state-space representation of a queue, making it an efficient solution to the unfairness problem.

8.2 Implementation issues

When designing a fair supervisory control for a discrete-event system, three ingredients must be specified by a user beforehand:

1. The set of events which he would like to be treated fairly. We call this the *fairness set*.

2. A scope of behavior $\mathcal{T}_i$ for each event σ_i in the fairness set.

3. A bound $N_i \in \mathbb{N}$ for each event σ_i in the fairness set.

It is hard for a program to predict any of these parameters without appropriate guidance provided by the user. Some general guidelines are listed below that might help the user with his decision.

1. To determine the fairness set, one should have in mind that since uncontrollable events are generated by the plant they are outside the realm of supervisory control. Thus, we feel justified in saying that *the system cannot be controlled any more fairly than what the plant allows.* For this reason it is reasonable to select only a subset of controllable events as the fairness set. In the DP problem,

the "eat" events of all philosophers constitute the fairness set, which is equal to the set of controllable events.

2. The scope of behavior $\mathcal{T}_i$ for an event σ_i is heuristically determined by including those events whose occurrence is deemed relevant by the user in the fair treatment of σ_i. When σ_i is the access event of the i^{th} process to a resource, we select the scope $\mathcal{T}_i$ initially to consist of σ_i and the progress events of all other processes which share the same resource exclusively. In the DP problem, this set is the "eat" event of a philosopher and his two neighbors. However, the scope may have to be broadened if as a result of some complicated causal relationship between events the fairness procedure returns an empty implementation. When this happens the user should add, by intelligent trial-and-error, enough events to the scope so that a nonempty implementation is achieved.

3. In the absence of any information on the probabilistic behavior of the system, there is not much that can be said about the selection of an appropriate bound. Suppose N processes compete to access a shared resource. If they are to be treated equally, each process will have to wait at least $(N-1)$-times for its turn while the other processes are served. Thus the rule of thumb is to select $N_i = |\mathcal{T}_i| - 1$ as a first estimate. The exact value of N_i , however, could vary depending on the application. A larger bound may be selected if it is acceptable to delay service to a process further. On the other hand, a smaller bound may be desired if σ_i has a higher priority than the other events in the scope, and must be served more frequently at the expense of low priority processes accessing the resource less often.

   When the system is represented by a stochastic model such as a Markov chain, by performing a statistical analysis one can predict how many transitions each process will have to wait on average before it can access a shared resource.

For example, consider a C1M1 system modeled by the discrete-time Markov chain of Figure 8.1. In addition to event labels, each arc is labeled with the probability with which it occurs. Note that at each state the sum of all probabilities equals unity.

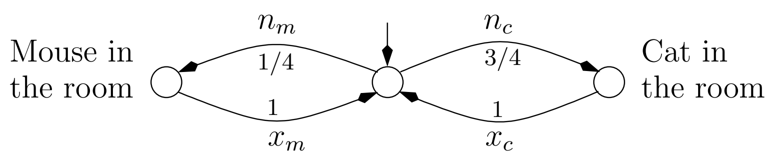


Figure 8.1: A Markov chain model for C1M1.

Suppose the scope is $\mathcal{T} = \{n_c, n_m\}$ and assume that the above four events are independent. Then if m is a random variable denoting the number of transitions (labeled within the scope) the mouse waits before it enters the room for the first time, the expected value of m will be equal to:

$$\begin{aligned} EXP(m) &= 0\frac{1}{4} + 1\frac{3}{4}\frac{1}{4} + 2\frac{1}{4}(\frac{3}{4})^2 + \ldots \\ &= \frac{1}{4}\sum_{k=0}^{\infty} k(\frac{3}{4})^k \\ &= 3 \end{aligned}$$

Thus on average, the mouse waits for three transitions (that is, the cat enters the room three times) before it enters the room. If one is satisfied with the above statistical guarantee, there is no need to implement any kind of fairness. However, if the unlikely long waits cannot be tolerated in an application , then a reasonable first guess for the bound can be obtained analytically by computing the mean access time to the shared resource, as shown in the example.

The above issues aside, the first decision to make in a design problem is to determine which type of fairness is desired to be implemented. When naive supervision results in continuous disablement of an event, weak fairness is violated, and a supervisor is needed to ensure that the event in question does not remain disabled continuously longer than a specified number of times. On the other hand, when despite enabledness the concern is whether an event in the fairness set will in fact occur, one needs the strong fairness guarantee that an event will occur next if it has been eligible previously a specified number of times.

When discrete-event systems are modeled by finite deterministic automata, we implement fairness by augmenting the state space with counter variables. Each counter is initialized to zero at the initial state and counts up to N, where N is a bound specified by the user. Values above N represent unfair situations and are disregarded.

The algorithms of Pages 166 and 167 illustrate the procedures. To implement weak fairness, bad states are replaced with N siblings numbered from 1 to N, while good states are untouched. A transition from a good state to a bad one is redirected to the first sibling of the bad state (line 10). On the other hand, whenever there is a transition from a bad state to a good one, transitions with the same label are created from every sibling of the bad state to the good state (line 12). Finally, in line 15 a transition between two bad states increments the counter by one if the labelling event is within the scope. In this way we keep track of how long an event has been continuously disabled.

In the algorithm of Page 167 each state is replaced with $N + 1$ siblings, corresponding to counter values $0, 1, \ldots, N$. The counter is reset (line 10) whenever the event τ of interest is taken. The counter value is incremented by 1 in line 13 when an event within the scope other than τ happens while τ has *been eligible*. We define in line 17 that τ has been eligible at a state if τ is eligible at that state, or the state is reached via transitions outside of scope from a state at which τ was eligible.

Note that if the above algorithms leave any bad terminal states behind, those states must be removed and the entire procedure must be repeated if necessary.

Although the algorithms presented here return an optimal solution, the state size becomes exponentially large as the number of processes grows. When a monolithic design is not possible, the divide-and-conquer approach described earlier in this section for construction of queues provides a highly efficient alternative to the general fairness problem—weak or strong.

8.3 Future work

We conclude this chapter by listing some relevant issues that deserve further investigation.

- We argued that in practice there is often a bound on how long an event can be disabled, or its occurrence can be postponed. In some applications, however, it may be reasonable to ask whether a smallest such bound exists. For example, in the Readers-Writers problem it may be desired to look for a smallest time a Writer has to wait before it can write the database. We already partially investigated this issue in Chapter 7, where we noted that the smallest bound when equally scheduling M processes is equal to $M - 1$. When the processes do not have to be treated equally, finding a smallest bound is an open question.

- We studied only a special kind of liveness property, namely, fairness. Although we conjecture that the treatment of general liveness properties, such as "β must occur within N steps following the occurrence of α" is similar, such properties can neither be expressed nor implemented in our setting. Thus a more comprehensive framework for design of supervisory control for liveness properties would be a worthwhile goal.

- In Chapter 3 we gave up altogether on allowing priorities into our setting as soon as we found that with a priority relation the fairness property is not closed under arbitrary union. Since priorities are a useful concept in practice, it may be worth looking for sufficient conditions to be satisfied by a priority relation under which robustness with respect to union is regained. For example, the pathological example of Page 50 can be avoided by requiring that P be a total order.

- We found in Chapter 5 that in general it is co$\mathcal{NP}$-hard to determine if the local supervisors always agree on an event for the next move and never bring the system to a halt. Through a detailed analysis of the Dining Philosophers problem we argued somewhat informally that a decentralized solution is possible since the enabled set of the local queues has a nonempty intersection and there is always an action that all supervisors agree on. We feel the above condition could and should be expressed more rigorously. Moreover, there is room to explore other sufficient conditions for a decentralized design.

8.4 Pseudo-code for weak and strong fairness

In the next two pages we present algorithms for implementing weak and strong fairness. The logic behind the algorithms was explained in Section 8.2; in this subsection we identify the input parameters. Some instructions are borrowed from the CTCT software [dp], while a number of high-level instructions are employed without specifying the implementation details. For example, to add a transition t to an automaton $\mathbf{M}$ we invoke the command **addtransition**$(\mathbf{M}, t)$. We assume that all commands return a trim automaton.

Input parameters

- $\mathbf{P} = (X, x_0, \eta, \Sigma)$ Plant
- $\mathbf{S} = (Y, y_0, \xi, \Sigma)$. Specification where:

- X, Y:	State sets;
- x_0, y_0:	Initial states;
- Σ:	Alphabet of events;
- $\eta : X \times \Sigma \to X$	Plant's partial transition function;
- $\xi : Y \times \Sigma \to Y$	Specification's partial transition function.

- $\mathcal{F} = \{\tau_1, \tau_2, \ldots, \tau_M\}$ Fair set
- $\mathcal{T} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_M\}$ Scopes
- $\mathcal{N} = \{N_1, N_2, \ldots, N_M\}$ Bounds

  where for all i, $N_i \in \mathbb{N}^+$, $\mathcal{T}_i \subseteq \Sigma$, and $\tau_i \in \mathcal{T}_i$.

Output

A trim automaton $\mathbf{M}$ that generates the largest $\{(\tau_i, N_i)\}_{i=1}^{M}$-fair sublanguage of $P \cap S$ within the scope $\mathcal{T}$.

Pseudo code for weak fairness

```
procedure wf(P,S,𝓕,𝓣,𝓝)
M :=meet(P, S)
for all τ_i ∈ 𝓕
    for each state z = (x, y) in M
        b(z) := ¬ξ(x, τ_i)! ∧ η(x, τ_i)!
        if b(z) then for k = 1, 2, . . . , N_i
            M := addstate (M,(z, k))
    for each transition t = (u, σ, v) in M
        if ¬b(u) ∧ ¬b(v) then t_1 := t
        elseif ¬b(u) ∧ b(v) then t_1 := (u, σ, (v, 1))
        elseif b(u) ∧ ¬b(v) then for all k = 1, 2, . . . , N_i
            t_k := ((u, k), σ, v)
        elseif b(u) ∧ b(v) then
            for all k = 1, 2, . . . , N_i − 1
                if σ ∈ 𝓣_i then t_k := ((u, k), σ, (v, k + 1))  else t_k := ((u, k), σ, (v, k))
            if σ ∉ 𝓣_i then t_{N_i} := ((u, N_i), σ, (v, N_i))
        for k = 1, 2, . . . , N_i
            M := addtransition (M,t_k)
        M := removetransition (M,t)
    for each state z in M
        if b(z) then M :=removestate(M,z)
return(M)
```

Pseudo code for strong fairness

procedure ***sf***$(\mathbf{P},\mathbf{S},\mathcal{F},\mathcal{T},\mathcal{N})$
$\mathbf{M}$:=**meet**$(\mathbf{P},\mathbf{S})$
for all $\tau_i \in \mathcal{F}$
    for each state $z = (x, y)$ in $\mathbf{M}$
        been_eligible$(z) := \eta(x, \tau_i)!$
        for $k = 0, 1, \ldots, N_i$
            $\mathbf{M}$:= **addstate** $(\mathbf{M},(z, k))$
    for each transition $t = (u, \sigma, v)$ in $\mathbf{M}$
        if $\sigma = \tau_i$ **then for** $k = 0, 1, \ldots, N_i$
            $t_k := ((u, k), \sigma, (z, 0))$
        elseif $\sigma \neq \tau_i \wedge \sigma \in \mathcal{T}_i \wedge$ *been_eligible*(u) **then**
            for $k = 0, 1, \ldots, N_i - 1$
                $t_k := ((u, k), \sigma, (v, k + 1))$
        elseif $\sigma \notin \mathcal{T}_i \vee \neg$*been_eligible*$(u)$ **then**
            for $k = 0, 1, \ldots, N_i$
                $t_k := ((u, k), \sigma, (v, k))$
            if *been_eligible*(u) **then** *been_eligible*$(v) := \top$
        for $k = 0, 1, \ldots, N_i$
            $\mathbf{M}$:= **addtransition** $(\mathbf{M},t_k)$
        $\mathbf{M}$:= **removetransition** $(\mathbf{M},t)$
    for each state z in $\mathbf{M}$
        removestate$(\mathbf{M}, z)$
return(M)

Appendix A

Existence of maxima under priorities

Before we prove the existence of maximal N-fair sublanguages we introduce some elementary definitions from order theory [DP90].

Let $\mathcal{P}$ be any set. A *partial order* $\leq_{\mathcal{P}}$ on $\mathcal{P}$ is a reflexive, anti-symmetric and transitive binary relation on $\mathcal{P}$. The pair $(\mathcal{P}, \leq_{\mathcal{P}})$ is called a *partially ordered set*, or a poset for short. For $x, y \in \mathcal{P}$ we denote:

$$x < y \quad \text{iff} \quad x \leq y \wedge x \neq y$$

A sub-poset is defined in the usual way: $(\mathcal{S}, \leq_{\mathcal{S}})$ is a sub-poset of $(\mathcal{P}, \leq_{\mathcal{P}})$ iff

$$\mathcal{S} \subseteq \mathcal{P} \quad \text{and} \quad \leq_{\mathcal{S}} = \leq_{\mathcal{P}} \ \cap \ \mathcal{S} \times \mathcal{S}$$

When the same relation is used to order the elements of $\mathcal{S}$, we may drop the subscript of $\leq$.

We call a poset $(\mathcal{P}, \leq)$ a CPO (*complete partially ordered set*) iff:

1. $\mathcal{P}$ has a bottom element.

2. The limit of every chain in $\mathcal{P}$ exists.

Observe that $(\mathcal{P}, \leq)$ is a CPO if it is a complete lattice.

A sub-poset $(\mathcal{S}, \leq)$ of a CPO $(\mathcal{P}, \leq)$ is a sub-CPO if $(\mathcal{S}, \leq)$ itself is a CPO.

Let E be a prefix-closed sublanguage of Σ^*. Denote by $\mathcal{P}$ the set of closed sublanguages of E:

$$\mathcal{P} = \left\{K \subseteq E \mid K = \overline{K}\right\}$$

We use set inclusion to order the members of $\mathcal{P}$. It can be easily verified that $(\mathcal{P}, \subseteq)$ is a complete lattice. Define the subclass of N-fair sublanguages of E to be:

$$\mathcal{S} = \left\{K \in \mathcal{P} \mid K \text{ is } N\text{-fair}\right\}$$

We have already seen that every chain in $\mathcal{S}$ has a limit in $\mathcal{S}$ (Lemma 3.3, Page 52). We also note that $\emptyset \in \mathcal{S}$, which in the solution domain is interpreted as "no solution exists"; in other words, $\mathcal{S} = \{\emptyset\}$ means that E does not have an N-fair sublanguage.

Thus, $(\mathcal{S}, \subseteq)$ is a sub-CPO of $(\mathcal{P}, \subseteq)$. By Zorn's lemma it follows that $\mathcal{S}$ has maxima.

Here we present an order-theoretic proof of the above fact, which is adapted from [DP90]. We need to build upon the following lemma, which has roots deep in order theory.

Lemma A.1 *Let $(\mathcal{P}, \leq)$ be a CPO and $\Phi : \mathcal{P} \to \mathcal{P}$ be a map such that $x \leq \Phi(x)$ for all $x \in \mathcal{P}$. Then Φ has a fixed-point.*

Now we have the needed machinery to prove the existence of maxima in $\mathcal{S}$, or equivalently, the Zorn's lemma. In our proof we make explicit use of the Axiom of Choice; that given a nonempty family $\mathcal{A} = \{A_i\}_{i \in I}$ of nonempty sets, there exists a choice function for $\mathcal{A}$; that is, a map

$$f : I \to \bigcup_{i \in I} \{A_i\} \text{ such that } \forall i \in I.\ f(i) \in A_i.$$

Theorem A.1 *$\mathcal{S}$ has maxima.*

Proof. By contradiction. Suppose $\mathcal{S}$ does not have a maximum, i.e.

$$\forall x \in \mathcal{S}.\ \exists y \in \mathcal{S}.\ x < y$$

For every x in $\mathcal{S}$ define:

$$A_x = \{y \in \mathcal{S} |\ x < y\}$$

Since $\mathcal{S}$ does not have a maximum A_x is nonempty for all $x \in \mathcal{S}$. We also note that $\mathcal{S}$ is nonempty. By the Axiom of Choice, therefore, there exists a choice function $\Phi : \mathcal{S} \to \mathcal{S}$ such that:

$$\forall x \in \mathcal{S}.\ \Phi(x) \in A_x$$

$(\mathcal{S}, \subseteq)$ is a (sub)-CPO, and since $x \leq \Phi(x)$ for every $x \in \mathcal{S}$ it follows from Lemma A.1 that Φ has a fixed-point, i.e. $x^* = \Phi(x^*)$ for some $x^* \in \mathcal{S}$. But as a consequence of the definition $x^* < \Phi(x^*)$, a contradiction. ∎

Appendix B

Computation of Ψ_x and Ψ_y

The functions Ψ_x and Ψ_y, defined in Page 70, are computed as follows.

To compute $\Psi_x(\cdot)$ we remove all terminal strings (or states) that are bad, that is, at which τ is disabled, from the language (or the automaton, respectively) and back-track.

The function Ψ_y is computed first by attaching a natural number between 0 and $N+1$ to every string. Good strings are indexed with zero. The index of a bad string indicates its position in a bad run. All bad strings positioned $N+1$ or higher are numbered $N+1$, as they all violate the N-fairness assumption and are considered undesirable. An N-fair sublanguage is obtained by removing all strings indexed with $N+1$. The $N = 0$ case is treated separately.

B.1 Computing Ψ_x

Recall that for $M \in \mathcal{P}(E)$ we define:

$$\Psi_x(M) = \mathcal{K}_x(M)^{\uparrow}$$

where

$$\mathcal{K}_x(M) := \left\{K \subseteq M |\ K = \overline{K} \wedge \forall s \in K.\ s\tau \in L - K \rightarrow \exists t.\ t \neq \epsilon \wedge st \in K\right\}.$$

This defines a map $\Psi_x : \mathcal{P}(E) \rightarrow \mathcal{P}(E)$ since for all $M \in \mathcal{P}(E)$ we have $\Psi_x(M) \subseteq M$.

The map Ψ_x removes as many bad strings from a subset M of E as is necessary to ensure that no bad string is terminal in what is left, that is, $\Psi_x(M)$. It is natural to expect that in a state-machine this operation is performed one step at a time, or iteratively. Each iteration takes us one step closer to the desired goal, and $\Psi_x(M)$ is obtained when the iteration terminates.

To this end we define a map $\Omega_x : \mathcal{P}(E) \rightarrow \mathcal{P}(E)$ on the sublanguages of E as follows. Let $M \in \mathcal{P}(E)$. Define:

$$\Omega_x(M) = \mathcal{R}_x(M)^{\uparrow}$$

where

$$\mathcal{R}_x(M) := \left\{K \subseteq M |\ K = \overline{K} \wedge \forall s \in K.\ s\tau \in L - M \rightarrow \exists t.\ t \neq \epsilon \wedge st \in M\right\}$$

It can be quickly checked that Ω_x is order-preserving.

Lemma B.1 Ω_x *is order-preserving, i.e.:*

$$M_1 \subseteq M_2 \rightarrow \Omega_x(M_1) \subseteq \Omega_x(M_2)$$

■

From the definition it immediately follows that $\mathcal{R}_x(M)$ is a bigger class than

$\mathcal{K}_x(M)$. The implication

$$s\tau \in L - K \rightarrow \exists t.\ t \neq \epsilon \wedge st \in K$$

in the definition of $\mathcal{K}_x(M)$ has been weakened twofold: First, by strengthening the antecedent, which is accomplished by replacing $s\tau \in L - K$ with $s\tau \in L - M$ and noting that:

$$s\tau \in L - M \rightarrow s\tau \in L - K$$

Thus, concern about a string is raised when it is bad in M.

Second, the consequent is weakened, as $st \in K$ is replaced by $st \in M$ and we have:

$$st \in K \rightarrow st \in M$$

That is, a bad string is considered desirable if it has a non-empty extension in M, which is a larger language than K.

Figure B.1 contrasts the actions of Ψ_x and Ω_x on languages. As before, the unspecified labels are not equal to τ. It also suggests the lemma that follows.

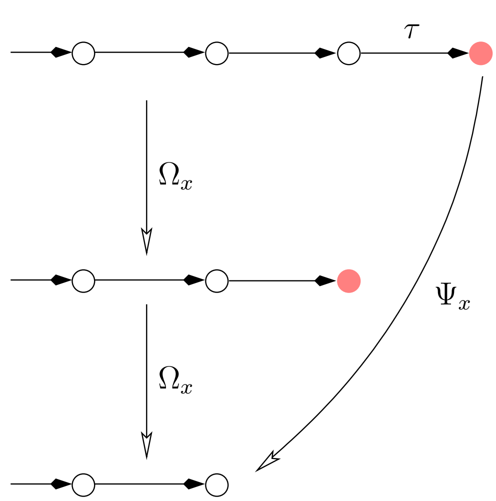


Figure B.1: Step-by-step implementation of Ψ_x.

Lemma B.2 *Let $M \in \mathcal{P}(E)$ and assume the sequence $\{\Omega_x^i(M)\}_{i\in\mathbb{N}}$ terminates at $i = i^*$, i.e. $\Omega_x^{i^*+1}(M) = \Omega_x^{i^*}(M)$. Then we have $\Psi_x(M) = \Omega_x^{i^*}(M)$.*

Proof. ($\supseteq$) We note that $\Omega_x^{i^*}(M) \subseteq M$. Since $\Omega_x^{i^*}(M)$ is a fixed-point of Ω_x we have:

$$\Omega_x^{i^*}(M) = \overline{\Omega_x^{i^*}(M)} \wedge \forall s \in \Omega_x^{i^*}(M).\ s\tau \in L - \Omega_x^{i^*}(M) \rightarrow \exists t.\ t \neq \epsilon \wedge st \in \Omega_x^{i^*}(M)$$

Thus $\Omega_x^{i^*}(M) \in \mathcal{K}_x(M)$ and therefore $\Omega_x^{i^*}(M) \subseteq \Psi_x(M)$.
($\subseteq$) We have previously shown that iterating down from E yields the largest fixed-point of an order-preserving map $\Theta : \mathcal{P}(E) \rightarrow \mathcal{P}(E)$. A mild generalization of this result states that iterating down from $M \subseteq E$ yields the largest fixed-point of Θ that is smaller than M.

But it is readily verifiable that $\Psi_x(M)$ is a fixed-point of Ω_x that is smaller than M, therefore we must have:

$$\Psi_x(M) \subseteq \Omega_x^{i^*}(M)$$

and that completes the proof. ■

All that is left to show is that the iteration indeed terminates when it is implemented on a finite automaton. We prove that repeated applications of Ω_x decrease the state size of an automaton and therefore the iteration must terminate after at most n repetitions, where n is the state size of the original automaton.

Before stating the result we introduce some notation. In harmony with including only prefix-closed languages in our setup, we only consider those automata whose states are all accepting. We denote by M the closed language generated by an automaton $\mathbf{M}$. Also we let $||\mathbf{M}||$ denote the number of states of an automaton $\mathbf{M}$. Finally, we denote by $\boldsymbol{\Theta}$ the implementation on automata of a map Θ on languages.

Let M be a language, and s_1 and s_2 be two strings in M. s_1 is *Nerode equivalent* to s_2, denoted by $s_1 \equiv_M s_2$, if they share the same future with respect to membership in M, i.e.:

$$s_1 \equiv_M s_2 \quad \text{iff} \quad \forall u \in \Sigma^*.\ s_1 u \in M \leftrightarrow s_2 u \in M$$

Equivalent strings in a language can all be lumped into a single equivalence class to form a canonical recognizer (or generator) for the language. For any automaton all strings generated in a state are equivalent[1].

The following result states that no additional memory (i.e. state) is required to implement Ω_x on an automaton.

Lemma B.3 *Let $M \in \mathcal{P}(E)$, and s_1 and s_2 be in $\Omega_x(M)$. As usual $E \subseteq L$ represents a closed-loop behavior of the system L. We have:*

$$s_1 \equiv_M s_2 \quad \wedge \quad s_1 \equiv_L s_2 \quad \longrightarrow \quad s_1 \equiv_{\Omega_x(M)} s_2$$

Proof. Let $s_1, s_2 \in \Omega_x(M)$ be such that $s_1 \equiv_M s_2$ and $s_1 \equiv_L s_2$, and $u \in \Sigma^*$ be such that $s_1 u \in \Omega_x(M)$. It follows that $s_1 u \in M$ and therefore $s_2 u \in M$ since s_1 and s_2 are equivalent mode M. If $s_2 u\tau \in L - M$ it follows that:

$$\begin{array}{lll} & s_1 u\tau \in L - M & \text{; since } s_1 \equiv_M s_2 \quad \wedge \quad s_1 \equiv_L s_2 \\ \Rightarrow & \exists t.\ t \neq \epsilon \wedge s_1 ut \in M & \text{; since } s_1 u \in \Omega_x(M) \\ \Rightarrow & \exists t.\ t \neq \epsilon \wedge s_2 ut \in M & \text{; since } s_1 \equiv_M s_2 \end{array}$$

Thus $\Omega_x(M) \cup \overline{s_2 u} \in \mathcal{R}_x(M)$. Note that only $s_2 u$ had to be scrutinized for membership as all its proper prefixes have non-empty extensions to $s_2 u$ in M. It follows that $\Omega_x(M) \cup \overline{s_2 u} \subseteq \Omega_x(M)$, i.e. $s_2 u \in \Omega_x(M)$. If $s_2 u\tau \notin L - M$, it follows from $s_2 u \in M$ that $s_2 u \in \Omega_x(M)$. Thus, in either case we have found that $s_2 u \in \Omega_x(M)$ and therefore $s_1 \equiv_{\Omega_x(M)} s_2$. ■

[1]Let $\mathbf{M} = (X, x_0, \xi, \Sigma)$ be an automaton. $s \in \Sigma^*$ is generated in $x \in X$, or leads to $x \in X$, iff $x = \xi(x_0, s)$.

Let s_1 and s_2 be two strings leading to a state of $\mathbf{M} \times \mathbf{L}$. We have:

$$s_1 \equiv_M s_2 \quad \text{and} \quad s_1 \equiv_L s_2$$

If s_1 and s_2 survive the application of Ω_x, from the above lemma it follows that:

$$s_1 \equiv_{\Omega_x(M)} s_2 \quad \text{and} \quad s_1 \equiv_L s_2$$

Applying the result $i \in \mathbb{N}$ times we will get:

$$s_1 \equiv_{\Omega_x^i(M)} s_2 \quad \text{and} \quad s_1 \equiv_L s_2$$

Thus we have:

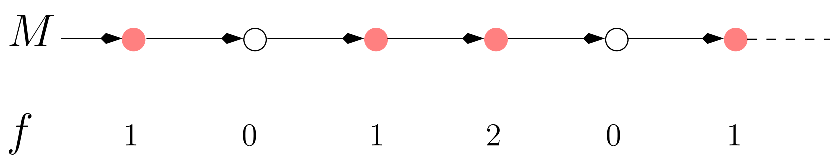
$$||(\boldsymbol{\Omega_x^i}(\mathbf{M}) \times \mathbf{L}|| \leq ||\mathbf{M} \times \mathbf{L}|| = ||\mathbf{M}|| \times ||\mathbf{L}||$$

Thus, repeated applications of $\boldsymbol{\Omega_x}$ on the **meet** of $\mathbf{M}$ and $\mathbf{L}$ at least do not increase the size. In fact, if $\Omega_x(M) \subsetneq M$, then let $s \in M - \Omega_x(M)$. It follows from the lemma that for any t such that $s \equiv_M t$ and $s \equiv_L t$ we have $t \in M - \Omega_x(M)$, that is, when applied on M, Ω_x shaves off *all* strings that are equivalent to s. This will amount to removing at least one state from $\mathbf{M} \times \mathbf{L}$. Therefore, the iterative application of $\boldsymbol{\Omega_x}$ on $\mathbf{M} \times \mathbf{L}$ terminates in at most $||\mathbf{M}|| \times ||\mathbf{L}||$ steps.

B.2 Computing Ψ_y

Defining an index function

Let $M \in \mathcal{P}(E)$ be a closed sublanguage of E. We start by defining an index function $f : M \to \{0, 1, \ldots, N+1\}$. f is initialized to zero on good strings. If s is bad then $f(s)$ indicates its position in a bad run ending at s. It is unimportant to know how far past N a position lies so all positions exceeding N are numbered $N+1$.



The index function f is formally defined below.

Definition B.1 *Let M be a closed sublanguage of E. Define a map $f : M \to \{0, 1, \ldots, N+1\}$ as follows:*

1. *If $s \in M_g$ then $f(s) = 0$.*

2. *If $s \in M_b$ then f is defined recursively as follows:*

$$\begin{aligned} f(\epsilon) &= 1 \\ f(s\sigma) &= \begin{cases} f(s)+1 & ; \ \textit{if } f(s) \leq N \\ N+1 & ; \ \textit{otherwise.} \end{cases} \end{aligned}$$

So starting from 1 f counts the number of bad strings in a run up to $N+1$. It is reset to zero when a good string is encountered. When there is the possibility of confusion we subscript f with M.

B.2.1 Computing Ψ_y when $N > 0$

A string that is indexed with $N+1$ is positioned $N+1$ or higher in a bad run and is clearly undesirable as far as N-fairness is concerned. Formally, we define:

$$M_{>N} := \{s \in M | \ f(s) = N+1\}$$

$M_{>N}$ is the collection of all strings whose positions exceed the bound placed on the length of a bad run and therefore must be excluded from a fair sublanguage of M.

Specifically we have the following lemma. It claims that the event of interest remains disabled as one moves up to N steps back from an excluded string.

Lemma B.4 *Let $s' \in M_{>N}$. Then there exist s and t such that $s' = st$, $|t| = N$, and*

$$\forall u \leq t.\ su \in M_b$$

Proof. Immediate from the definition. ■

Let:

$$M_{\leq N} := \{K \subseteq M - M_{>N} |\ K = \overline{K}\}^{\uparrow}$$

Thus, $M_{\leq N}$ is the largest closed sublanguage of M that does not contain any of the excluded strings, nor their continuations. This will give us $M_{\leq N} = \Psi_y(M)$ as long as no *new* bad strings are created after trimming M. Suppose $s\tau$ has to be trimmed from M, that is, $s\tau \in M_{>N}$, or $f(s\tau) = N + 1$. Since $s\tau \in M$ the string s must be good in M and therefore:

$$f(s\tau) \leq f(s) + 1 = 0 + 1 = 1$$

Hence $N = 0$. In other words, when reducing M to $M_{\leq N}$, the event τ will not be disabled unless $N = 0$. Note that $N = 0$ corresponds to the case where the event of interest cannot be disabled by a fair supervisor *at all*.

Before getting involved in formally establishing that $M_{\leq N} = \Psi_y(M)$, we spell out two rather obvious results that are utilized in the proof. The first result states that if τ is disabled at a string s in a language E, it remains disabled at all smaller languages $K \subseteq E$, that is, disablement is inherited downwards. The second result states in simple words that enablement is inherited upwards.

Lemma B.5 *Let* $K \subseteq E$ *be two languages. Then:*

1. $K \cap E_b \subseteq K_b$

2. $K_g \subseteq E_g$

Proof. 1. Let $s \in K$ and $s \in E_b$. We have $s\tau \in L - E$, implying that $s\tau \in L - K$. Therefore $s \in K_b$.

2. Let $s \in K_g$, i.e. $s \in K$ and $s\tau \in L \rightarrow s\tau \in K$. It follows that $s \in E$ and

$$s\tau \in L \rightarrow s\tau \in E$$

Therefore $s \in E_g$. ■

We now prove the main result of this section, that $M_{\leq N} = \Psi_y(M)$ when $N > 0$.

Theorem B.1 *Let the language* $M_{\leq N}$ *be defined as above and* $N > 0$*. Then* $M_{\leq N} = \Psi_y(M)$*.*

Proof. ($\supseteq$) By definition $\Psi_y(M) = \overline{\Psi_y(M)}$ and $\Psi_y(M) \subseteq M$. It remains to show $\Psi_y(M) \subseteq M^c_{>N}$, where "$c$" stands for complement. Let $s' \in M_{>N}$. From a previous result it follows that s' can be written as $s' := st$, where $|t| = N$ and:

$$\forall u \leq t.\ su \in M_b$$

In particular, note that we get $s \in M_b$ upon choosing $u = \epsilon$.

If $s' \in \Psi_y(M)$ it follows that $s \in \Psi_y(M)$. This, together with $s \in M_b$ imply $s \in \Psi_{y_b}(M)$. Since by assumption

$$|t| = N \wedge st = s' \in \Psi_y(M)$$

it follows from the definition of $\Psi_y(M)$ that

$$\exists u \le t.\ su \in \Psi_{y_g}(M)$$

But since $\Psi_{y_g}(M) \subseteq M_g$ we would have:

$$\exists u \le t.\ su \in M_g$$

which is a contradiction. Thus $s' \notin \Psi_y(M)$ and therefore $\Psi_y(M) \subseteq M^c_{>N}$.
($\subseteq$) Clearly $M_{\le N} \subseteq M$ and $M_{\le N} = \overline{M_{\le N}}$. Define:

$$\mathcal{K}_y(M) := \left\{K \subseteq M \middle| K = \overline{K} \wedge \forall s \in K_b, t \in \Sigma^*.\ |t| = N \wedge st \in K \to \exists u \le t.\ su \in K_g\right\}$$

If $M_{\le N} \notin \mathcal{K}_y(M)$ there must exist $s \in (M_{\le N})_b$ and $t \in \Sigma^*$ such that

$$|t| = N \wedge st \in M_{\le N} \wedge \forall u \le t.\ su \in (M_{\le N})_b$$

Let u be any prefix of t. Since $su \in (M_{\le N})_b$ we must have $su \in M_{\le N}$ and $su\tau \in L$ but $su\tau \notin M_{\le N}$. If $su\tau \in M$ we would have $su \in M_g$ and therefore:

$$f(su\tau) \le f(su) + 1 = 0 + 1 = 1 < N + 1$$

implying that $su\tau \in M_{\le N}$, a contradiction. It follows that $su\tau \notin M$, which in turn implies $su \in M_b$. Thus we have found a $t \in \Sigma^*$ such that:

$$|t| = N \wedge st \in M \wedge \forall u \le t.\ su \in M_b$$

It follows from the definition that $f(st) = N+1$, or $st \notin M_{\leq N}$, a contradiction. Thus we must have $M_{\leq N} \in \mathcal{K}_y(M)$ as desired, and therefore $M_{\leq N} \subseteq \Psi_y(M)$. ■

Implementation on automaton

In this section we implement the partial function f on an automaton with at most $(N+1) \times ||\mathbf{M}|| \times ||\mathbf{L}||$ states. Thus f has an implementation of size polynomial in the sizes of $\mathbf{M}$ and $\mathbf{L}$. After removing all states indexed with $N+1$, the language $M_{\leq N}$ is generated by the automaton that remains.

The idea is to remember, at any state where τ is disabled, how many bad states have been visited *in succession*—including the current one—prior to reaching the current state. When we are at the expanded state (q, n), the machine is at state q while the event of interest has been continuously disabled in the last n states. We only need to count up to $N+1$; a number k attached to a state corresponds to a bad run of length k, and is considered undesirable if $k \geq N+1$.

Formally, let $\mathbf{M} = (X, x_0, \xi, \Sigma)$ and $\mathbf{L} = (Y, y_0, \eta, \Sigma)$. To implement f, we form an automaton $\mathbf{F} = (Z, z_0, \theta, \Sigma)$, where:

$$Z = X \times Y \times \{0, 1, \ldots, N+1\}$$

and

$$z_0 = \begin{cases} (x_0, y_0, 1) & ; \quad \neg\xi(x_0, \tau)! \wedge \eta(y_0, \tau)! \\ (x_0, y_0, 0) & ; \quad \text{otherwise.} \end{cases}$$

On $(z, \sigma) := \big((x, y, n), \sigma\big) \in Z \times \Sigma$, the partial transition function $\theta : Z \times \Sigma \to Z$ is defined iff $\xi(x,\sigma)$ and $\eta(y,\sigma)$ are defined, in which case:

$$\theta\big((x,y,n),\sigma\big) = \begin{cases} \big(\xi(x,\sigma),\eta(y,\sigma),n+1\big) & ; \ \neg\xi(x,\tau)! \wedge \eta(y,\tau)! \wedge n < N+1 \\ \big(\xi(x,\sigma),\eta(y,\sigma),N+1\big) & ; \ \neg\xi(x,\tau)! \wedge \eta(y,\tau)! \wedge n = N+1 \\ \big(\xi(x,\sigma),\eta(y,\sigma),0\big) & ; \ \text{otherwise} \end{cases}$$

It is straightforward to verify that the language generated by the automaton **F** is equal to $M \cap L$, and that for $s \in F$, if $\theta(z_0,s) = (x,y,n)$ then $f(s) = n$.

Lemma B.6 *The following can be said about the automaton* **F***:*

1. *$F = M \cap L$.*
2. *For $s \in F$, if $\theta(z_0,s) = (x,y,n)$ then $f(s) = n$.*

Proof. (1) A direct consequence of the definition: for every (x,y,n), $\theta\big((x,y,n),\sigma\big)$ is defined if and only if both $\xi(x,\sigma)$ and $\eta(y,\sigma)$ are defined.
(2) By induction on the length of s. The details are omitted. $\square$

After implementing f on the product automaton, it remains to remove all states of the form $(x,y,N+1)$ from the state space of **F** in order to generate $M_{\leq N}$, which is equal to $\Psi_y(M)$ when $N > 0$.

For $N > 0$ we define $\mathbf{M_{\leq N}}$ to be the reachable restriction of **F** to $X \times Y \times \{0,1,\ldots,N\}$. Then it is easy to check that $\mathbf{M_{\leq N}}$ indeed generates $M_{\leq N}$.

Example 18 Let $\mathbf{M} \wedge \mathbf{L}$ be as given in Figure B.2. We assume that τ is disabled in grey states, and that N is given to be equal to 2. The automata $\mathbf{F}_{\text{reach}}$ and $\mathbf{M_{\leq N}}$ are shown in the same figure, where we have denoted by $\mathbf{F}_{\text{reach}}$ the reachable sub-automaton of **F**. $\square$

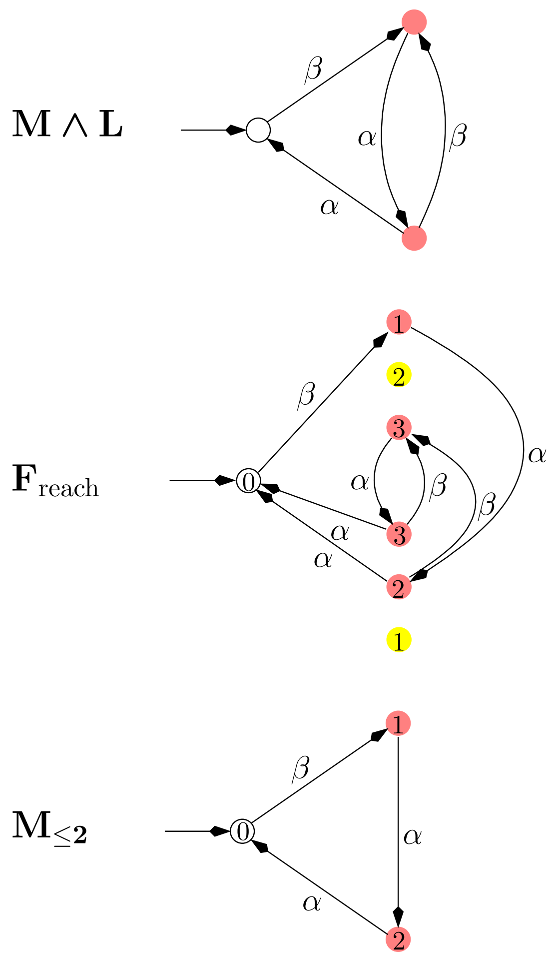


Figure B.2: Computing $\Psi_y(M)$ when $N = 2 > 0$.

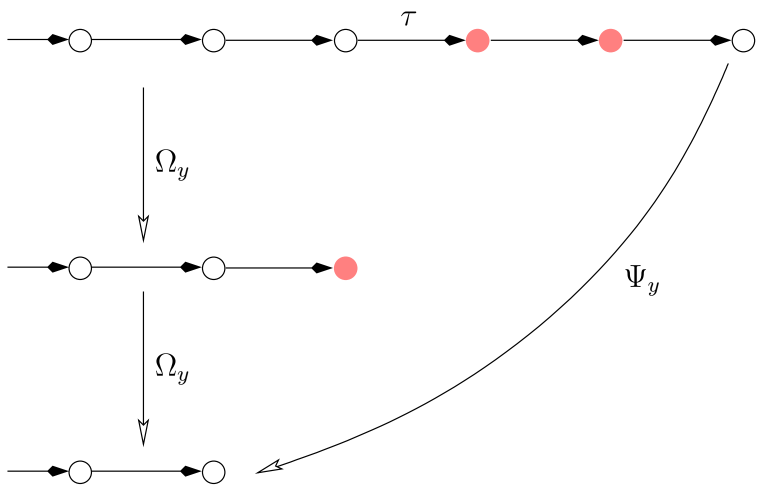


Figure B.3: Computing $\Psi_y(M)$ when $N = 0$.

B.2.2 Computing Ψ_y when $N = 0$

When $N = 0$, no bad string is allowed in $\Psi_y(M)$. If a bad string is of the form $s\tau$, this is achieved by disabling τ at s, thereby creating a *new* bad string, which must be taken care of accordingly. Thus, the procedure of obtaining $\Psi_y(M)$ from M is iterative, as Figure B.3 suggests. Once again, the unspecified labels are not equal to τ.

The map $\Omega_y : \mathcal{P}(E) \to \mathcal{P}(E)$ is defined on the sublanguages of E as follows. Let $M \in \mathcal{P}(E)$. Define:

$$\Omega_y(M) = \mathcal{R}_y(M)^{\uparrow}$$

where

$$\mathcal{R}_y(M) := \{K \subseteq M |\ K = \overline{K} \wedge \forall s \in K.\ s\tau \notin L - M\}$$

Note that when $N = 0$ the class $\mathcal{K}_y(M)$ can be written as:

$$\mathcal{K}_y(M) := \{K \subseteq M |\ K = \overline{K} \wedge \forall s \in K.\ s\tau \notin L - K\}$$

It can be quickly checked that Ω_y is order-preserving. The following result confirms that iterating Ω_y on M, if it ever terminates, will yield $\Psi_y(M)$.

Lemma B.7 *Let $M \in \mathcal{P}(E)$ and the sequence $\{\Omega_y^i(M)\}_{i\in\mathbb{N}}$ terminate at $i = i^*$, i.e. $\Omega_y^{i^*+1}(M) = \Omega_y^{i^*}(M)$. Then we have $\Psi_y(M) = \Omega_y^{i^*}(M)$.*

Proof. ($\supseteq$) Since $\Omega_y^{i^*}(M)$ is a fixed-point of Ω_y we have:

$$\Omega_y^{i^*}(M) \subseteq M \wedge \Omega_y^{i^*}(M) = \overline{\Omega_y^{i^*}(M)} \wedge \forall s \in \Omega_y^{i^*}(M).\ s\tau \notin L - \Omega_y^{i^*}(M)$$

Thus $\Omega_y^{i^*}(M) \in \mathcal{K}_y(M)$ and therefore $\Omega_y^{i^*}(M) \subseteq \Psi_y(M)$.
($\subseteq$) We note that $\Omega_y^{i^*}(M)$ is the largest fixed-point of Ω_y that is smaller than M, and that $\Psi_y(M)$ is a fixed-point of Ω_y that is smaller than M. It follows that

$$\Psi_y(M) \subseteq \Omega_y^{i^*}(M).$$

■

As we did in Section B.1 for Ω_x, it remains to show that the iteration indeed terminates when it is implemented on a finite automaton. We prove this result by showing that each application of Ω_y decreases the state size of the product automaton and therefore the iteration must terminate after at most n applications of Ω_y, where n is the state size of the product automaton.

Lemma B.8 *Let $M \in \mathcal{P}(E)$, and s_1 and s_2 be in $\Omega_y(M)$. As usual $E \subseteq L$ represents the closed-loop behavior of the system L. We have:*

$$s_1 \equiv_M s_2 \quad \wedge \quad s_1 \equiv_L s_2 \quad \rightarrow \quad s_1 \equiv_{\Omega_y(M)} s_2$$

Proof. Let $s_1, s_2 \in \Omega_y(M)$ be such that $s_1 \equiv_M s_2$ and $s_1 \equiv_L s_2$, and $u \in \Sigma^*$ be such that $s_1u \in \Omega_y(M)$. For $v \leq u$ we have:

$$\begin{aligned} s_2v\tau \in L &\Rightarrow s_1v\tau \in L \ ; \text{since } s_1 \equiv_L s_2 \\ &\Rightarrow s_1v\tau \in M \ ; \text{since } s_1v \in \Omega_y(M) \\ &\Rightarrow s_2v\tau \in M \ ; \text{since } s_1 \equiv_M s_2 \end{aligned}$$

Hence $s_2v\tau \in L \rightarrow s_2v\tau \in M$, or $s_2v\tau \notin L - M$, for all $v \leq u$. Since $s_2 \in \Omega_y(M)$ it follows from the supremality of Ω_y that $s_2u \in \Omega_y(M)$. ■

Let s_1 and s_2 be two strings leading to a state of $\mathbf{M} \times \mathbf{L}$. We have:

$$s_1 \equiv_M s_2 \ \text{ and } \ s_1 \equiv_L s_2.$$

If s_1 and s_2 survive the application of Ω_y, from the above result it follows that:

$$s_1 \equiv_{\Omega_y(M)} s_2 \ \text{ and } \ s_1 \equiv_L s_2$$

Applying the Lemma $i \in \mathbb{N}$ times we will get:

$$s_1 \equiv_{\Omega_y^i(M)} s_2 \ \text{ and } \ s_1 \equiv_L s_2$$

Thus we have:

$$||\boldsymbol{\Omega_y^i}(\mathbf{M}) \times \mathbf{L}|| \leq ||\mathbf{M} \times \mathbf{L}|| = ||\mathbf{M}|| \times ||\mathbf{L}||$$

Thus, the repeated applications of $\boldsymbol{\Omega_y}$ on the **meet** of $\mathbf{M}$ and $\mathbf{L}$ at least do not increase the size. In fact, if $M \subsetneq \Omega_y(M)$, then we must have a string $s \in M - \Omega_y(M)$. It follows from the lemma that for any t such that $s \equiv_M t$ and $s \equiv_L t$ we have $t \in M - \Omega_y(M)$, that is, when applied on M, Ω_y shaves off all strings generated at the same state as s. This will amount to removing at least one state from $\mathbf{M} \times \mathbf{L}$.

Therefore, the iterative application of $\boldsymbol{\Omega_y}$ on $\mathbf{M} \times \mathbf{L}$ terminates in at most $||\mathbf{M}|| \times ||\mathbf{L}||$ steps.

Bibliography

[AH98] R. Alur and T.A. Henzinger. Finitary fairness. *ACM Transactions on Programming Languages and Systems*, 20(6):1171–94, 1998.

[AS87] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[B̈60] J.R. Büchi. Weak second order arithmetic and finite automata. *Zeitschrift fur̈ Math. Log. und Gruňdl. der Math*, 6:66–92, 1960.

[BH89] Y. Brave and M. Heymann. On stabilization of discrete event processes. In *Proceedings of the 28th Conference on Decision and Control*, pages 2737–42, 1989.

[Cas93] Christos G. Cassandras. *Discrete Event Systems: Modeling and Performance Analysis*. Aksen Associates, 1993.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, vol 8(2):244–263, April 1986.

[Dij71] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta ¿ Informatica*, 1:115–138, 1971.

[dp] CTCT download page. http://www.control.toronto.edu/people/profs/wonham/wonham.html.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[Fra86] Nissim Francez. *Fairness*. Springer-Verlag New York Inc., 1986.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[JD86] D.N. Jayasimha and N. Dershowitz. Bounded fairness. Technical Report 615, University of Illinois, Urbana, IL, 1986.

[LP98] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, second edition, 1998.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.

[MK99] Jeff Magee and Jeff Kramer. *Concurrency: State Models & JAVA Programs*, chapter 7. John Wiley & Sons, 1999.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag New York, Inc, 1992.

[NB98] R. Negulescu and J.A. Brzozowski. Relative liveness: From intuition to automated verification. *Formal Methods in System Design*, 12:73–115, 1998.

[Rin86] G.A. Ringwood. The dining logicians (concurrent programming problem). In *Proceedings of the SEAS Spring Meeting*, volume 2, pages 1103–29, 1986.

[Rut00] J.J.M.M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.

[RW87] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM J. Control and Optimization*, 25(1):206–230, 1987.

[TW94] J.G. Thistle and W.M. Wonham. Supervision of infinite behavior of discrete-event systems. *SIAM Journal on Control and Optimization*, 32(4):pp. 1098–1113, 1994.

[VN92] N. Viswanadham and Y. Narahari. *Performance Modeling of Automated Manufacturing Systems*. Prentice-Hall, 1992.

[Won02] W.M. Wonham. *Notes on Control of Discrete-Event Systems.* revised 2002.07.01. http://www.control.toronto.edu/people/profs/wonham/wonham.html, 2002.

Index