The Committee Problem and Delay-Insensitive Circuits

by

Igor Benko

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1994

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

The committee problem (n-party rendezvous) is a general synchronization problem that combines mutual exclusion and condition synchronization. We show that the following problems are special cases of the committee problem: phase synchronization, mutual exclusion, dining philosophers, the arbitrating test-and-set, and the n-input SEQUENCER. Specifications of committee schedulers for these special cases and for the general case are given in terms of a CSP-like language called **commands** and in terms of Petri nets. Efficient implementations of the special-case schedulers as delay-insensitive circuits are derived. A number of implementations of a general committee scheduler are proposed: a naive implementation which suffers from deadlock, and several implementations based on sequential finite state machines. Different ways of implementing the finite state machines for the general committee scheduler are discussed.

Acknowledgments

I wish to express my gratitude to my mentor Jo Ebergen. His door was always open and he always found time for a long discussion. Thanks are extended to my readers John Brzozowski and Michael Coffin, who both carefully read the thesis and contributed many valuable comments. John Segers, my fellow student, was always around when a raw idea called for polishing.

The funding provided by the Institut Jožef Stefan, the Institute for Computer Research, the Information Technology Research Centre, and the Natural Sciences and Engineering Research Council made my study at the University of Waterloo possible. Their support is gratefully acknowledged.

I would like to thank my parents for raising me as a curious child and for supporting my wish to go overseas. Finally, many thanks go to my soul-mate Jasna for her love and support.

My grandparents and uncle Ivo were always present in my thoughts. To them, I dedicate this thesis.

Contents

List of Figures

Chapter 1

Introduction

Synchronization issues inevitably arise when a computation involves cooperation of concurrent processes. They have been attracting researchers for more than 25 years, beginning with resource-sharing problems that arose during the development of early multi-user operating systems.

There are two basic forms of synchronization: *mutual exclusion* and *condition synchronization* (sometimes referred to as just synchronization). When dealing with condition synchronization, one usually breaks a computation into phases and requires that before the beginning of a phase a process is delayed until a number of conditions, which often depend on other processes, are true. On the other hand, the classic exclusion problem considers processes sharing a common resource. A solution of the exclusion problem must ensure that only one process is accessing the resource at any time.

There is a large class of concurrency control problems that combine synchronization and exclusion. These problems have been studied extensively, because of their importance in operating systems, real time systems, distributed systems, database systems, and other areas. Many of them are now folklore and their solutions have become a part of standard concurrent programming techniques. Among such problems are producers/consumers, readers/writers, bounded buffer, dining philosophers, and the sleeping barber problem.

In order to obtain solutions for concurrency control problems, several types of synchronization have been introduced, *rendezvous* being one of them. Rendezvous is a means of synchronization and communication between *two* processes: The processes first synchronize, then some computation and communication takes place, and, finally, the two processes separate. Rendezvous has proven to be a powerful and convenient method of combining synchronization and communication and is present in many programming languages. Examples are CSP, Ada, SR, and RADDLE.

A shortcoming of the rendezvous is that the synchronization involves two processes only. In order to overcome this limitation there have been proposals for the rendezvous of n-processes, also known as the *committee problem.*

1.1 The Committee problem

The committee problem captures the essence of synchronization in many concurrent systems. It was introduced by Chandy and Misra [CM88] with the following story:

> Professors in a certain university have organized themselves into committees. Each committee has an unchanging membership roster of one or more professors. From time to time, a professor may decide to attend a committee meeting (presumably, when it has no more publishable ideas); it starts waiting and remains waiting until a meeting of a committee of which it is a member is started. The restrictions on

> meetings are as follows: (1) a committee meeting may be started only if *all* members of that committee are waiting, and (2) no two committees may meet simultaneously if they have a common member. Given that all meetings terminate in a finite time, a problem is to devise a protocol, satisfying these restrictions, that also guarantees that if all members of a committee are waiting, then at least one of them will attend some meeting.

In the context of multiprocess systems, a professor represents a process, a committee represents a set of processes that may synchronize, and a committee meeting represents a synchronization event. The problem is then to guarantee that if all processes required for a synchronization event are ready to synchronize, then at least one of these processes participates in some synchronization event. Note that we cannot require that, if all members of a committee are waiting, then the committee meets: if two committees have a common member and all members of both committees are waiting, then only one of them can meet.

Let us emphasize that the problem includes aspects of both the condition synchronization and the mutual exclusion. Synchronization is required when a committee meets, because *all* members of the committee have to be waiting. Exclusion, on the other hand, is captured by the request that two committees with a common member cannot meet at the same time.

1.2 Related work

The idea of n-party rendezvous (also called multiway rendezvous, or multirendezvous) has been discussed in different areas: concurrent programming, programming languages, and formal description techniques. Here, we give a brief overview of this work and then introduce asynchronous circuits, the target of our implementation of a solution for the committee problem.

1.2.1 Programming languages

The work in programming languages can be divided into two parts:

1. searching for an appropriate synchronization abstraction, and
2. exploring whether existent programming constructs are appropriate for an implementation of the multiway rendezvous.

Two examples of proposed abstractions are a *compact* [Cha87] and a *script* [FHT86]. In the compact, the rendezvous synchronization is achieved by blocking processes that are to take part in the rendezvous until a set of communication actions ("ACQUIRE ENTRY BLOCK") has been performed. Then, the rendezvous computation ("RENDEZVOUS STATEMENT BLOCK") can take place. Unfortunately, the compact as proposed by Charlesworth [Cha87] does not support the option that a process waits for one among several specified synchronizations, which would be equivalent to the condition that a professor can attend a meeting of any committee he or she is a member of. Furthermore, no "low level" implementation is proposed.

The script is a parametrized program section to which processes *enroll* in order to execute their *roles*. The execution of roles may be delayed until processes enroll in all the roles of a given instance of a script; this is the way to achieve rendezvous synchronization. Since each process executes its own role, no common computation is present (as it is in the compact). Bagrodia [Bag87] argues that, because a script enrollment is allowed within a guard, a process may be ready to enroll in anyone of several different script instances, which corresponds to a professor waiting for a meeting of any of his or her committees. However, there may be objections to such a reasoning: (1) the decision which script to enroll in is made before a request for the enrollment is issued; this corresponds to the situation where a professor decides himself or herself which meeting he or she will be waiting for, and (2) the evaluation of the guards is based on local information only; in Section 6.1 we show how such a decision can lead to deadlock. Hence, we claim that waiting for one among several possible synchronizations is not directly provided using scripts. No directions for implementing the script are given.

Besides introducing the compact, Charlesworth [Cha87] also stated a number of conditions that should be fulfilled for a programming language to support a method of programming using the multiway rendezvous. Four well-known languages, CSP, Ada, SR, and Distributed Processes, were examined and none of them were found to provide the adequate support according to the conditions stated by [Cha87]. Thus, a class of simple and natural solutions cannot be programmed easily in these languages. As a response, Coffin and Olsson [CO89] proposed an elegant approach to programming multiway rendezvous in SR using only existing language mechanisms. However, their method suffers from the same lack of generality as the compact and the script: it does not cover waiting for one of several possible synchronizations.

1.2.2 Concurrent programming

The efforts in the area of concurrent programming have been focused on the development of algorithms for implementing multiway rendezvous in a shared-memory and in a message-based system. The majority of the work has considered special cases of the multiway rendezvous; examples are *mutual exclusion* (with only the exclusion aspect present, [Ray86]), *barrier synchronization* (also called *phase synchronization*; only the synchronization aspect is present, [Mis91, And91, MCS91]), and *dining philosophers* [Dij68, CM88, And91]. In Chapters 3 and 5 we discuss the specification and implementation of algorithms for these special cases. Our implementation consists of asynchronous circuits.

Chandy and Misra [CM88] stated the general multirendezvous problem in terms of committees and proposed a centralized and a distributed solution. In the distributed solution, the exclusion problem is solved by mapping the multiway rendezvous onto the dining philosophers problem. They utilize a solution, colloquially called *hygienic philosophers*. The synchronization is achieved using tokens.

Bagrodia [Bag87] proposed a simple distributed solution that uses a number of committee managers connected into a token ring. Each manager is assigned a set of committees and each committee is assigned to at least one manager. Committee managers accept ready messages from members of their committees. When a manager has a token and some of its committees may meet, the manager allows meetings by sending messages to all the members of the committees that will meet. Then it passes the token to the next manager. Because one committee may be assigned to several committee managers, the token carries information (similar to Lamport's logical clocks [Lam78]) about the current state of the professors. In [Bag89], Bagrodia combined algorithms from [CM88] and [Bag87] into a new solution.

All three algorithms we have mentioned so far have a common failure: starvation can occur. In other words, it is possible that all members of a committee are waiting for a meeting for an unbounded number of times and the committee never meets, because some members attend a meeting of another committee. We discuss this problem in Section 6.2.3.

Kumar [Kum90] observed this shortcoming and proposed a fair algorithm (without a proof), based on resource ordering strategy [SPG91]: a professor has the role of a resource, and a token (one per committee) has the role of a task, "grabbing" the resources. Kumar's statement of the problem is more general than the original committee problem: every time when a professor is willing to attend a meeting, he or she chooses (from the set of committees of which he is a member) a set of committees he or she is waiting for to meet (a willing set). Recall, that in the original problem, the professor's willing set is equal to the set of all committees he or she is a member of. We can modify a solution of the original problem to satisfy Kumar's generalization: We create an instance of a professor for every willing set he or she might wish to wait for. However, we have to admit that in the worst case the number of instances of a professor can be exponential in the number of the committees in which he or she participates.

1.2.3 Formal description techniques

The multiway rendezvous is inherently present in the language LOTOS [ISO89, BB87]. The language was standardized by International Standards Organization and have been used mostly for a formal description of Open-System-Interconnection protocols. Implementation of the multiway rendezvous is therefore a key issue in developing tools for the execution of LOTOS "programs". Moreover, a solution could also be used to emulate the synchronization mechanisms of LOTOS in Estelle [DB89], another formal description language, standardized by ISO. This would enable a LOTOS user to take advantage of a tool, developed for Estelle, but not available for LOTOS.

The algorithm by Sisto, Ciminiera, and Valenzano [SCV91] implements a more general version of the multiway rendezvous than captured in the committee problem: a process which is ready to participate in one or more multiway rendezvous does not necessarily wait until one of them takes place. Instead of waiting, the process can execute local actions which may lead to a withdrawal from a rendezvous before it takes place. In the solution, processes are organized in a binary tree structure. The processes that can participate in a rendezvous have the role of leaves, and additional synchronizing processes are added to fill the tree structure (this doubles the total number of processes). The idea behind the solution is that every parental process acts as a synchronization manager for its child processes. This solution does not prevent starvation.

1.3 Asynchronous circuits

The vast majority of circuits designed today operate in lockstep with successive pulses of a clock; such circuits are called *synchronous* circuits. The correct operating of a synchronous (i.e., clock driven) system relies on the bounds of delays in elements and wires. Since the satisfaction of these requirements cannot be guaranteed under all circumstances, interest arose in the design of circuits without a clock, generally called *asynchronous circuits*. The target of our implementation is a special type of asynchronous circuits, called *delay-insensitive* circuits. Delay-insensitive circuits can be characterized as circuits with functional operation insensitive to delays in basic elements and connecting wires.

There are a number of reasons why one may want to design delay-insensitive circuits. Let us mention some of them:

Performance: Asynchronous systems tend to reflect average-case behavior, while synchronous systems tend to reflect worst-case behavior.

Metastability: Any computer circuit has a number of stable states and metastable states. When a circuit gets into a metastable state (i.e., a *glitch* [CM73] occurs) it can remain there for an indefinite period of time. If a synchronous circuit remains in a metastable state longer than the clock period, erroneous data may be sampled at the time of the clock pulses. For delay-insensitive circuits it does not matter whether a glitch occurs, since the computation is delayed until a stable state is reached.

Scaling: Because of the improvements in technology, integrated circuits are diminishing in size. However, if all characteristic dimensions of a circuit (including the clock period) are scaled down by a certain factor, the delays in long wires do not scale down proportionally to the clock period. The consequence is that a synchronous design may no longer work properly when scaled down. Delay-insensitive circuits may not suffer from this phenomenon, if the basic elements are chosen small enough so that the effects of scaling are negligible with respect to the functional behavior of these elements[vdS85].

Power consumption: Clock distribution can contribute up to 50% of the power consumption of an IC. With transition signaling [Sut89], where both rising and falling transition represent an event, we also avoid redundant events, where a clock signal has to return to the initial state, and save on energy. For both reasons, asynchronous circuits are expected to be more energy-efficient than synchronous circuits.

Correctness concerns: The physical and functional correctness concerns that can be applied to delay-insensitive circuits are clearly separated [Ebe89]. The correctness of the behavior of basic elements is proved by physical principles only. On the other hand, the correctness of the behavior of networks of basic elements is proved by mathematical principles only. Thus, in the delay-insensitive design of a network of basic elements, one does not consider delays in wires or elements, which is not true for synchronous design.

The design and analysis of asynchronous systems has been and still is a difficult subject. Several formalizations, design techniques, and analysis techniques have been proposed [Udd86, Mar86, BS89, vBNRS88, Ebe89, Dil89]. However, many problems still remain to be solved. For example, formulation of complexity and performance criteria, including liveness and progress properties of the design, and testing of the asynchronous circuits are topics of current research.

1.4 Thesis overview

The goal of this thesis is to give a formal specification of the committee scheduler and its decomposition into a network of basic (delay-insensitive) components. We propose a number of implementations for special cases and for the general case of the committee problem and discuss their complexity, liveness, and progress properties.

The remainder of the thesis is organized as follows. In Chapter 2 we introduce trace structures and the **commands** language that we use to represent regular trace structures. In Chapter 3 we give specifications of schedulers for a number of special cases of the committee problem and a specification of the scheduler for the general case. The specification in terms of a **command** is compared to a Petri net specification. In Chapter 4 we define correctness criteria for a decomposition and for a DI-decomposition, and present two theorems that help us in the verification and in the derivation of a decomposition: the *Substitution Theorem* and the *Separation*

Theorem [Ebe91a]. In Chapter 5 we give decompositions for several special cases, namely mutual exclusion, phase synchronization, dining philosophers, arbitrating test-and-set, and sequencers. In Chapter 6 we discuss a number of decompositions of a general-case committee scheduler. Chapter 7 includes concluding remarks and some proposals for further research.

Chapter 2

Trace theory

Trace theory is the underlying mathematical framework that we use for specifying a computation and for designing asynchronous circuits that implement the specification. In this chapter we introduce directed trace structures and a notation for representing them [Ebe91a]. A more detailed discussion of trace theory can be found in [vdS85, Ebe89].

2.1 Trace structures

A parallel computation is performed by several components, all operating concurrently. A trace structure can be interpreted as a prescription for the behavior of a component *and* its environment; it defines all sequences of communication events that may occur between the component and its environment.

Formally, a (directed) trace structure R is a triple $\langle I, O, T \rangle$, where the set I is called the *input alphabet*, O is called the *output alphabet*, and T is called the *trace set*. With $\mathbf{i}R$, $\mathbf{o}R$, and $\mathbf{t}R$ we denote the input alphabet, the output alphabet and the trace set of R. An element of a trace set, a *trace*, is a finite sequence of symbols from $I \cup O$. A trace structure R is *non-empty* if $\mathbf{t}R \neq \emptyset$, and is *prefix-closed* if any prefix of every trace in $\mathbf{t}R$ is also an element of $\mathbf{t}R$. The empty trace is denoted by ε and is a prefix of any trace.

Notational convention: In the following, trace structures are denoted by the capitals R and S, traces by the lower-case letters t and s, alphabets by the capitals A and B. Alphabet symbols are usually denoted by the lower-case letters. With $\#i : R(i) : F(i)$ we denote applying operation $\#$ to $F(i)$ over the range $R(i)$. For example, the sum $1 + 2 + \ldots + n$ is denoted by $+i : 1 \leq i \leq n : i$. $\square$

In order to see how a component can be specified using trace structures, let us introduce the simplest component we use: a WIRE. It has two communication ports or *terminals*; let us call them a and b. An event at terminal a is always triggered by a component that is connected to the WIRE; such a terminal is called an *input terminal*. On the other hand, transitions at terminal b are always caused by the WIRE, therefore b is an *output terminal*. Each terminal is either an input or an output terminal. In the specification of the WIRE we require that events at input terminal a and output terminal b alternate, starting with an event on a (if any).

A behavior of a component can be described by a non-empty, prefix-closed trace structure with disjoint input and output alphabets. The input and the output alphabets represent the input and the output terminals of a component, and the trace set contains all possible communication behaviors between the component and its environment; a sequence of symbols in a trace can be interpreted as a sequence of events occurring at the corresponding terminals. The input and output alphabets are required to be disjoint, since every terminal is either an input or an output terminal. A trace structure must be prefix-closed, because any prefix of a behavior of the component is also a behavior. The condition that a trace structure is non-empty follows from the fact that the initial state of a component corresponds to the empty trace. Therefore ε is an element of the trace set of any trace structure that describes a component.

As an example, we give the trace structure E that prescribes the behavior of the WIRE. Obviously, $\mathbf{i}E = \{a\}$ and $\mathbf{o}E = \{b\}$. For the trace set we have a restriction that events at the input and the output terminals alternate, starting with an event on a (if any). The trace set satisfying this restriction can be described by a regular expression, thus $\mathbf{t}E = \{\varepsilon, a, ab, aba, \ldots\} = \mathcal{L}(\ (ab)^* \ + \ a(ba)^*\ ).\square$

2.2 Operations

The trace set of any component is a regular set; the corresponding trace structure is therefore called a *regular trace structure*. However, we do not specify the trace sets by regular expressions or by corresponding finite automata. Rather, we construct a trace structure by applying a number of operations to *atomic* trace structures. To denote regular trace structures, we use a CSP-like notation, called **commands** [Ebe91a]. The name was inspired by the "guarded command" language introduced by Dijkstra. First, we define the atomic trace structures and the corresponding commands:

command		trace structure
ε	$=$	$\langle \emptyset, \emptyset, \{\varepsilon\} \rangle$
$a?$	$=$	$\langle \{a\}, \emptyset, \{a\} \rangle$
$b!$	$=$	$\langle \emptyset, \{b\}, \{b\} \rangle$

We can apply the following operations on trace structures (and hence on commands):

$$
\begin{array}{llcllll}
\textit{concatenation:} & R\,;S & = & \langle\, \mathbf{i}R \cup \mathbf{i}S, & \mathbf{o}R \cup \mathbf{o}S, & (\mathbf{t}S)(\mathbf{t}R)\,\rangle \\
\textit{union:} & R \mid S & = & \langle\, \mathbf{i}R \cup \mathbf{i}S, & \mathbf{o}R \cup \mathbf{o}S, & \mathbf{t}S \cup \mathbf{t}R\,\rangle \\
\textit{repetition:} & *[R] & = & \langle\, \mathbf{i}R, & \mathbf{o}R, & (\mathbf{t}R)^*\,\rangle \\
\textit{prefix closure:} & \mathbf{pref}\,R & = & \langle\, \mathbf{i}R, & \mathbf{o}R, & \{\,t \mid \exists s\, : ts \in \mathbf{t}R\,\}\,\rangle \\
\textit{projection:} & R \downarrow A & = & \langle\, \mathbf{i}R \cap A, & \mathbf{o}R \cap A, & \mathbf{t}R \downarrow A\,\rangle \\
\textit{weaving:} & R \,\|\, S & = & \langle \mathbf{i}R \cup \mathbf{i}S, & \mathbf{o}R \cup \mathbf{o}S, & \\
 & & & \multicolumn{4}{r}{\{\,t \in (\mathbf{a}R \cup \mathbf{a}S)^* \mid t{\downarrow}\mathbf{a}R \in \mathbf{t}R \;\wedge\; t{\downarrow}\mathbf{a}S \in \mathbf{t}S\,\}\,\rangle}
\end{array}
$$

Concatenation of sets is denoted by juxtaposition; A^* denotes the set of all finite-length concatenations of elements of A (Kleene closure); $t \downarrow A$ denotes the projection of trace t on alphabet A, i.e. trace t from which all symbols not in A are deleted; with $\mathbf{t}R \downarrow A$ we denote the projection extended to sets: $\mathbf{t}R \downarrow A = \{t \downarrow A \mid t \in \mathbf{t}R\}$.

Concatenation, union, and repetition are known operations from formal language theory. Taking the prefix closure of trace structure R returns a trace structure with the same alphabets as R and with a trace set that includes all prefixes of traces in $\mathbf{t}R$. For example, the restriction that input and output transitions on a WIRE's terminals alternate can be expressed as

$$\textsc{wire}' = *[a?; b!]$$

However, the behavior of the WIRE contains traces ε, ab, $abab$, ..., but does not contain traces a, aba, ... By means of the **pref** operation we obtain a prefix-closed trace structure that specifies the WIRE:

$$\textsc{wire} = \mathbf{pref} *[a?; b!]$$

Weaving is the operation (or programming primitive) that expresses parallel composition *and* synchronization on common symbols. All traces in $\mathbf{t}(R\|S)$ are in accordance with $\mathbf{t}R$ and $\mathbf{t}S$; for this reason, weaving can be considered the *conjunction of subbehaviors* expressed with R and S.

To illustrate weaving we introduce another basic component with a communication behavior expressible as a conjunction of two subbehaviors: a JOIN. The JOIN has two input terminals, a and b, and one output terminal, c. The mechanistic interpretation of its behavior is that the JOIN may produce an output at c after inputs at both a and b have been received. Note that there is no restriction on the order in which the environment may produce a and b. This behavior can be interpreted as the conjunction of two subbehaviors: one behavior where a and c alternate, given by $\mathbf{pref} *[a?; c!]$, and one where b and c alternate, given by $\mathbf{pref} *[b?; c!]$. The conjunction of both subbehaviors gives the specification of the JOIN:

$$\text{JOIN} \;=\; \mathbf{pref} *[a?; c!] \;\|\; \mathbf{pref} *[b?; c!]$$

This specification of JOIN takes advantage of an important property of weaving, namely synchronization on common symbols. Output c is a part of both subbehaviors and thus expresses their synchronization. In order to clarify what we mean by synchronization on common symbols, we specify the JOIN in terms of a Petri net. We use the transitions in a Petri net to represent receipt of an input symbol or production of an output symbol. The specification is shown in Figure 2.1.

$$\text{JOIN} = \begin{array}{r} \mathbf{pref} *[a?; c!] \\ \| \; \mathbf{pref} *[b?; c!] \end{array} =$$

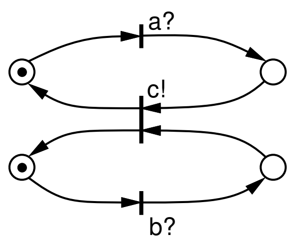


Figure 2.1: JOIN specified by commands and by a Petri net.

There are two special cases of weaving: If $\mathbf{a}R \cap \mathbf{a}S = \emptyset$, then $R \parallel S$ reduces to the interleaving of the traces from $\mathbf{t}R$ and $\mathbf{t}S$. If $\mathbf{a}R = \mathbf{a}S$, then $R \parallel S$ results in the intersection $\mathbf{t}R \cap \mathbf{t}S$.

We emphasize that weaving should be used as a means for specifying the parallel behavior of *one* component. Specifying the behavior of a component as a conjunction of subbehaviors is called *constraint-oriented* specification. It is often compared to the "divide-and-conquer" approach in algorithm design. We remark that "weavands" in a command do not necessarily represent subcomponents into which a component can be decomposed. A JOIN, for example, cannot be implemented by two WIREs.

Projection is the operation that is used for hiding symbols: We may introduce internal symbols in each protocol, use them to express synchronization, and finally hide them. Projection can be seen as a means of *delimiting the scope* of internal symbols. We use it when we discuss implementations of a component as a network of basic components. Usually, we emphasize the locality of the internal symbols by using an alternative notation: If $\mathbf{a}R = A \cup B$, where A is the set of (internal) symbols and $A \cap B = \emptyset$, then

$$|[A :: R]| \;=\; R \downarrow B$$

With symbol $|[$ we denote opening of the scope of A, and with $]|$ we close the scope. Because internal symbols are not used for a communication between a component and its environment, we denote a "production" of an internal symbol without the question or the exclamation mark.

We often introduce internal symbols in order to rewrite a specification to a form that is more convenient for further manipulation. Note that the introduction of internal symbols does not change the external alphabets of a component. However, because of a possible synchronization on an internal symbol we have to verify that the trace set is not affected.

In order to avoid many parentheses, we stipulate priority rules for the operations on trace structures. The unary operators — repetition and taking the prefix closure — have higher priority then the binary operators. Among the binary operators, weaving has the highest priority, next come concatenation, union, and finally projection.

2.3 Specification

We already mentioned that we specify components by means of commands that represent non-empty, prefix-closed trace structures with disjoint input and output alphabets. Let us describe more precisely how we interpret a specification.

Suppose the communication behavior is specified by command R, where $\mathbf{i}R \cap \mathbf{o}R = \emptyset$, and $\mathbf{t}R$ is prefix-closed and non-empty. Furthermore, let the communication actions that have already taken place correspond to trace $t \in \mathbf{t}R$ (initially $t = \varepsilon$). If $ta \in \mathbf{t}R$, we say that the action a may be occur next. Note that there may be more than one possible way to extend the trace t. If $a \in \mathbf{i}R$, then the environment may produce the next communication action a. On the other hand, if $a \in \mathbf{o}R$, then the component may trigger the communication action a. Because $\mathbf{i}R \cap \mathbf{o}R = \emptyset$, a communication action may be produced either by the component or by the environment, but not by both.

According to our interpretation, a specification prescribes not only the behavior of a component, but also what the environment may do (or, how the component should be used). In other words, the specification stipulates the conditions under which correct behavior of the component must be guaranteed. Consequently, the specification R can be interpreted as "if the environment produces the inputs as prescribed by R, then the component may (after an arbitrary delay) produce the outputs as prescribed by R". If the environment violates the prescription, nothing is guaranteed about the behavior of the component. For example, a violation of the environment prescription occurs when the environment of a WIRE produces two inputs in a row without waiting for the WIRE to produce an output. A violation of the environment prescription is also called *computation interference*. Later, when we consider networks of components as implementations of specifications, one of our proof obligations will be to show that computation interference cannot occur.

Our specification of the component also demands that every trace specified may occur. This becomes important when we try to implement a specification as a network of basic components. Implementations where some traces cannot occur are excluded. However, we do not require that every trace is guaranteed to occur. The actual occurrence of a trace may depend on nondeterministic decisions made during the computation. The requirement that all traces are possible has been criticized in [Pee90] as being too strong and too weak at the same time. In Sections 6.1 and 6.2.5 we present examples which demonstrate this deficiency.

We do not prescribe when a choice must be made. For example, consider the commands $R = \mathbf{pref}\,(a?; b!|a?; c!)$ and $S = \mathbf{pref}\,(a?; (b!|c!))$. A possible interpretation of the command R is that a component makes a choice which output to produce *before* (or at the time when) input a is received. The interpretation of the command S would then be that the component makes a choice *after* the input a has been received. Although different interpretations of commands R and S are possible, we cannot make a distinction between R and S, because their trace structures are the same. In some other formalisms, like CCS and LOTOS, R and S would not be considered (observationally) equivalent [BB87].

Non-empty, prefix-closed trace structures, which are the domain of our specifications, form one of the simplest semantic domains. Unfortunately, there is a price we may have to pay for this simplicity: We cannot specify all liveness properties such as the absence of deadlock or livelock. We encounter this problem in Chapters 5 and 6.

Although several physical implementations are allowed by our interpretation of a specification, we mention an electrical implementation only. We associate a terminal of a circuit with each symbol of the alphabet. Each occurrence of a symbol in a trace corresponds to a voltage transition at that terminal. We do not distinguish between rising and falling transitions; both transitions are denoted by the same symbol. This type of signaling is called *transition signaling* [Sut89]. Outputs of a component are transitions caused by a circuit and inputs are transitions caused by the environment. We do not discuss electrical implementations any further, but rather focus on the abstract interpretation.

2.4 Basic components

Let us describe some basic delay-insensitive components. Their specifications and schematics are given in Figure 2.2.

The WIRE and the IWIRE serve for the transmission of a signal between terminals. The only difference in their behaviors is in the initial states: For the WIRE, given by **pref** $*[a?; b!]$, the environment may initially produce the input a, and for the IWIRE, given by **pref** $*[b!; a?]$, the component may initially produce the output b. Operationally speaking, the WIRE corresponds to a physical wire. Note that as

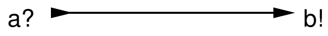
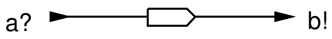
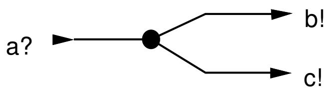
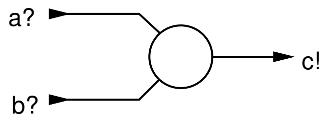
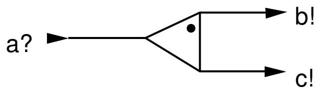
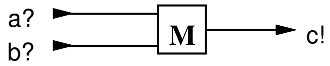
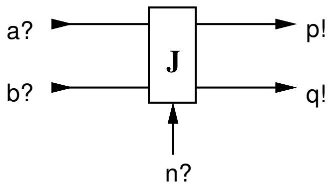
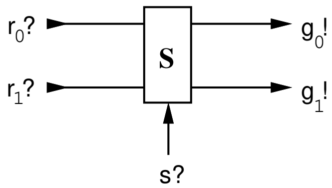
Name	Specification	Schematic
WIRE	**pref** $*[a?;b!]$	
IWIRE	**pref** $*[b!;a?]$	
FORK	**pref** $*[a?;\ (b!\|\|c!)]$	
JOIN	**pref** $*[a?\|\|b?;c!]$	
TOGGLE	**pref** $*[a?;b!;a?;c!]$	
MERGE	**pref** $*[(a?\mid b?);c!]$	
2-by-1 JOIN	**pref** $*[(a?;p!)\mid(b?;q!)]$ $\|\|$ **pref** $*[n?;(p!\mid q!)]$	
SEQUENCER	**pref** $*[r_0?;g_0!]$ $\|\|$ **pref** $*[r_1?;g_1!]$ $\|\|$ **pref** $*[s?;(g_0!\mid g_1!)]$	

Figure 2.2: Specifications of some basic components.

long as the environment behaves as required by the specification, there is always at most one transition propagating along a WIRE or IWIRE.

The FORK is used to duplicate a signal: After an input has been received, it may produce two outputs in parallel (in our context this means in an arbitrary order) and become ready to receive the next input from the environment.

The JOIN performs a primitive synchronization: After *both* inputs have been received, the JOIN may produce an output, which may be followed by the next input from the environment.

The TOGGLE determines the parity of the number of occurrences of the input signal. After an odd occurrence of input a it may produce output b and after an even occurrence of a the TOGGLE may produce output c. The environment is allowed to produce the next input only after the component has responded to the previous input.

The MERGE merges two input signals into one output signal. Note that the environment is allowed to produce only one of the inputs a and b in each "cycle". In both cases the MERGE may produce output c, after which the behavior may repeat.

The 2-by-1 JOIN is a generalization of the JOIN; it performs a synchronization between a or b and n. Either a and n are received in an arbitrary order and then p may be produced, or b and n are received and then q may be produced. Note that the environment is restricted to produce only one of the inputs a and b in each "cycle". In Chapter 5 we generalize the 2-by-1 JOIN to the n-by-m JOIN.

The SEQUENCER is an arbitrating component: Output g_0 is the *grant of request* r_0 and, similarly, output g_1 is the grant of request r_1. We say that request r is *pending* after trace t if $\#r(t) - \#g(t) = 1$, where $\#x(t)$ denotes the number of occurrences of symbol x in trace t. The SEQUENCER grants one request for each input s it receives. If two requests are pending upon occurrence of s, the component has to arbitrate which request to grant. The arbitration is resolved nondeterministically. If only one request is pending, no arbitration is needed. The environment is allowed to produce the next request after the component has produced the corresponding grant. Note that there is no requirement that the requests be granted fairly. However, since the specification requires that all traces are possible, we do not accept an implementation of the SEQUENCER that, in the case of arbitration, always favors the same request. In Chapter 5 we generalize the two-input SEQUENCER to the n-input SEQUENCER.

Chapter 3

Specification

In this chapter we discuss a specification of a committee scheduler. We see the scheduler as a component that accepts *ready messages* sent by professors when they become available for a meeting. When a committee is ready to meet, the scheduler sends messages to all members of the committee to inform them they can attend the meeting. After the meeting, a professor can, after some rest, send another ready message to the scheduler and thus signal his or her availability for another meeting. Note that professors send ready messages independently from each other.

We begin the discussion by giving specifications of schedulers for a number of special cases of the committee problem. At the end of the chapter we give a specification of a general committee scheduler using commands, and an equivalent specification using Petri nets.

3.1 Phase synchronization

The phase synchronization problem (sometimes called barrier synchronization, or just synchronization) can be stated as follows [Mis91]: Consider a set of n concurrent processes where each process executes a sequence of phases; a process begins a phase only upon completion of the previous phase. What exactly constitutes a phase is irrelevant for our purposes. We are required to design a synchronization mechanism that guarantees for each $k = 1, 2, \ldots$, that

- no process begins phase $k + 1$ until all processes have completed phase k;
- no process is permanently blocked from executing phase $k + 1$ if all processes have completed their phase k.

Phase synchronization is a basic means of achieving synchronization in a system that consists of asynchronous components. For example, a PRAM consists of processes that, in one step, read from a common memory, perform a local computation, and write into a common memory. No process begins the next step until all processes have completed their current steps. Another application of the phase synchronization can be found in iterative algorithms that successively compute better approximations to an answer until a termination criterion has been satisfied. Typically, such an algorithm manipulates an array of values, and each iteration performs the same computation on *all* array elements. Hence, we can often use a number of parallel processes to compute disjoint parts of an approximation; good examples are numerical algorithms. Because every process uses all array elements, processes have to synchronize upon completion of their current phases.

How is phase synchronization related to the committee problem? Every process can be seen as a professor, and the synchronization upon completion of a phase can be seen as a committee meeting. Thus, we have n professors that are all members of a single committee. Since there is no mutual exclusion, the phase synchronization can be seen as an extreme case of the committee problem with only condition synchronization present.

Specification

Let $p_1, p_2, \ldots, p_n$ denote the set of processes involved in synchronization. In order to specify the synchronization scheduler, we first define the communication between process p_i and the scheduler. Upon completion of its current phase, p_i sends a message r_i to the scheduler and waits until the execution of the next phase is granted. After the scheduler has learned that all processes have completed their current phases (i.e., all r_i messages have been received), it broadcasts message z to all processes and enables the execution of the next phase. Thus, the scheduler's behavior with respect to process p_i only is

$$\mathsf{constraint}_i \;=\; \mathbf{pref} *[r_i?; z!] \qquad 1 \leq i \leq n$$

The partial behavior described above can be seen as a constraint for the communication between process p_i and the scheduler. The final specification can then be expressed as a conjunction of all constraints:

$$\begin{aligned} \mathsf{phase\_synch}_n = \quad & \mathbf{pref} * [r_1?; z!] \\ & \| \; \mathbf{pref} * [r_1?; z!] \\ & \vdots \\ & \| \; \mathbf{pref} * [r_n?; z!] \end{aligned} \tag{3.1}$$

Note that all subbehaviors synchronize at the common output z.

The specification technique we use when we construct a behavior of a component as a conjunction of subbehaviors is called *constraint-oriented* specification. We will frequently apply it in the rest of the chapter.

In order to visualize the synchronization involved in the phase synchronization, we give a Petri net specification of the scheduler. As in the specification of the JOIN, given in Figure 2.1, we use the transitions in a Petri net to represent receipt of an input symbol or production of an output symbol. A specification is shown in Figure 3.1.

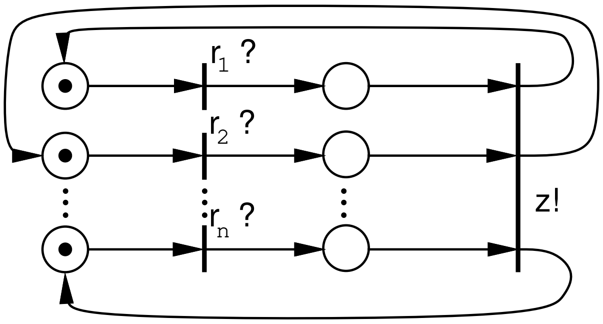


Figure 3.1: Petri net for the phase-synchronization scheduler.

3.2 Mutual exclusion

In another extreme case, the committee synchronization problem collapses to the mutual exclusion problem. In the traditional statement of the mutual exclusion problem n parallel processes compete for access to a shared resource. The resource can serve only one process at a time; therefore the section where a resource is used by a process is called the *critical section* of the process. It is required to design a synchronization protocol that guarantees that

- at most one process executes its critical section at a time, and
- a process that wants to enter its critical section will eventually enter it.

In order to relate the mutual exclusion problem to the committee synchronization we construct the following committee configuration: Process i, $1 \leq i \leq n$, has the role of professor p_i, and a shared resource (i.e., a critical section) has the role of professor cs. Professors are organized in n committees $C_1, \ldots, C_n$ where each committee C_i consists of two professors: $C_i = \{p_i, cs\}$. Since cs is a member of all committees, no two committees can meet concurrently. This corresponds to the requirement that only one process may be executing its critical section at a time. The requirement that every waiting process eventually gains access to its critical section does not correspond to the progress condition from the committee problem ("if all members of the committee are waiting, one of them will attend a meeting of one committee"). Since each committee consists of a professor and a critical section and since the critical section is a member of all committees, the progress condition from the committee problem is satisfied by any meeting of a committee, i.e., by any execution of a critical section. Thus, it is not guaranteed that every waiting process eventually enters its critical section. In Section 6.2.3 we discuss a stronger progress condition for committee meetings, which corresponds to the progress condition from the mutual-exclusion problem.

Specification

Let us examine the communication between process p_i and the scheduler. Process i requests access to its critical section by sending message r_i to the scheduler. The scheduler may later inform process i that it can enter its critical section by sending it message m_i. The scheduler's view of this communication can be specified as

$$\mathsf{constraint}_i \;=\; \mathbf{pref}\ *[r_i?; m_i!] \qquad 1 \leq i \leq n$$

In the communication between the scheduler and a shared resource, the resource tells the scheduler that it is ready to serve another process by sending message cs.

The scheduler may later respond with message m_i and inform the resource that it has to serve process i, where $1 \leq i \leq n$:

$$\mathsf{constraint}_{cs} \;=\; \mathbf{pref} *[cs?;(m_1! \,|\, m_2! \,|\, \ldots \,|\, m_n!)]$$

The behavior of the scheduler is determined by the conjunction of the constraints above:

$$\begin{aligned}
\mathsf{mutex} = \quad & \mathbf{pref} * [r_1?; m_1!] \\
& \| \; \mathbf{pref} * [r_1?; m_2!] \\
& \vdots \\
& \| \; \mathbf{pref} * [r_n?; m_n!] \\
& \| \; \mathbf{pref} * [cs?; (m_1! \,|\, m_2! \,|\, \ldots \,|\, m_n!)]
\end{aligned} \tag{3.2}$$

Note that process p_i and a shared resource synchronize on a common symbol m_i and that the resource can synchronize with at most one process at a time. In Figure 3.2 we give a Petri net specification, which visualizes synchronization between individual processes and a shared resource.

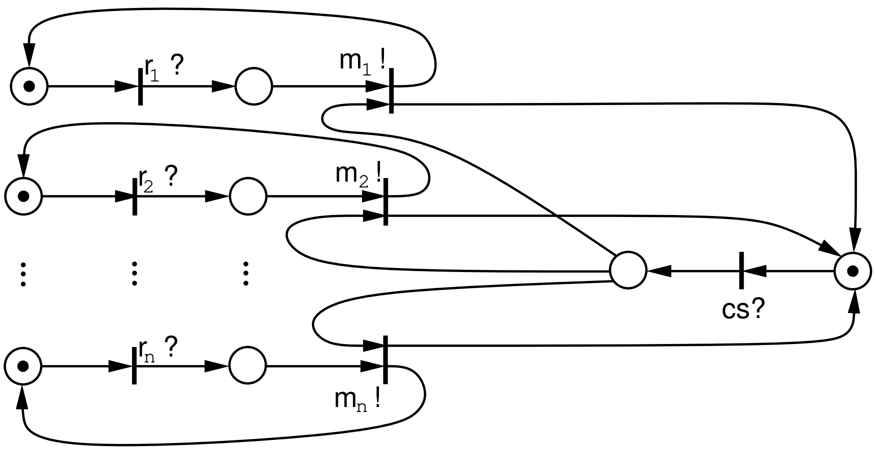


Figure 3.2: Petri net for the mutual-exclusion scheduler.

In the mutual-exclusion problem one tacitly assumes that initially a shared resource is ready to serve a process. Our specification, however, requires that the resource signals its availability before it is being used by any process. This requirement may be interpreted as waiting for an initialization of a device upon a start-up.

3.3 Dining philosophers

The dining philosophers is a canonical synchronization problem. It was originally stated by Dijkstra [Dij71]:

> Five philosophers, numbered from 0 through 4 are living in a house where the table is laid for them, each philosopher having his own place at the table (see Figure 3.3).

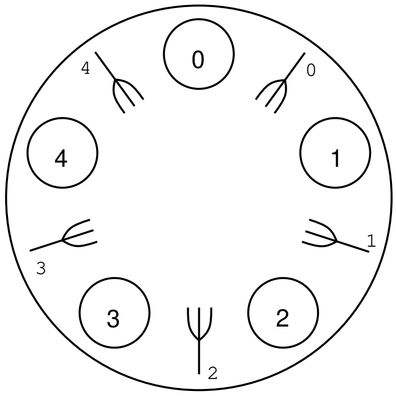


Figure 3.3: Dining philosophers table.

> Their only problem — besides those of philosophy — is that the dish served is a very difficult kind of spaghetti, that has to be eaten

> with two forks. There are two forks at each plate, so that presents no difficulty: as a consequence, however, no two neighbors can be eating simultaneously.

To keep the example simple we consider the case where there are only three philosophers. In order to relate the dining philosophers problem to the committee synchronization problem we construct 3 committees, each consisting of 3 members: forks i and $i+1$ (in this section addition is modulo 3) and philosopher i, for $0 \leq i < 3$. We interpret a meeting of a committee as a philosopher eating his spaghetti. Since every fork is a member of two committees and since there are two forks in each committee, the structure of the committees reflects that each fork can be used by two philosophers and that each philosopher uses two forks to eat spaghetti.

Specification

Once more, we are going to construct the behavior of the committee scheduler by specifying a number of partial behaviors, each representing a constraint on the scheduler's behavior.

In a response to a ready (i.e., hungry) message h_i from philosopher i, the scheduler informs the philosopher that he can eat by sending him a message e_i. Such a subbehavior, seen by the scheduler, can be expressed as

$$\mathbf{pref} *[\, h_i?;\ e_i!\,]$$

Furthermore, the receipt of message f_i means that fork i is free to be used by a philosopher. The scheduler may then assign a fork to a philosopher j, where $j = i$ or $j = i + 1$. This partial behavior can be specified as

$$\mathbf{pref} *[\, f_i?;\ (\, e_i! \mid e_{i+1}!\,)]$$

The conjunction of the partial behaviors for all forks and for all professors gives us the specification for the scheduler:

$$\begin{aligned} \mathsf{dining}_3 = \quad & \mathbf{pref} *[f_0?; (\, e_0! | e_1!\,)] \\ \| \; & \mathbf{pref} *[f_1?; (e_1! \mid e_2!)] \\ \| \; & \mathbf{pref} *[f_2?; (e_2! \mid e_0!)] \\ \| \; & \mathbf{pref} *[h_0?;\ e_0!] \\ \| \; & \mathbf{pref} *[h_1?;\ e_1!] \\ \| \; & \mathbf{pref} *[h_2?;\ e_2!] \end{aligned} \tag{3.3}$$

Recall that a behavior of a weave is in accordance with subbehaviors specified by weavands. Thus, we can see that a weavand

$$\mathbf{pref} *[f_i?; (e_i! \mid e_{i+1}!)]$$

imposes that fork i is used by at most one philosopher at a time, i.e., the fork has to be released before it can be used again. Furthermore, weavands

$$\begin{aligned} & \mathbf{pref} *[h_i?;\ \underline{e_i!}] \\ \| \; & \mathbf{pref} *[f_{i-1}?; (e_{i-1}! \mid \underline{e_i!})] \\ \| \; & \mathbf{pref} *[f_i?; (\underline{e_i!} \mid e_{i+1}!)] \end{aligned}$$

impose that philosopher i can eat (e_i) only when he is hungry and when both fork $i-1$ and fork i are free.

Of course, we can generalize the problem to n philosophers. The specification of the scheduler then becomes

$$\begin{aligned} \mathsf{dining}_n \quad = \quad & (\; \| i :\ 0 \le i < n :\ \mathbf{pref} *[\, f_i?; (e_i! \mid e_{(i+1)\,\mathbf{mod}\,n}!)\,]\;) \\ \| \; & (\; \| i :\ 0 \le i < n :\ \mathbf{pref} *[\, h_i?;\ e_i!\,]\;) \end{aligned}$$

In order to illustrate the synchronization and exclusion expressed in the specification above, the reader may wish to construct a corresponding Petri net.

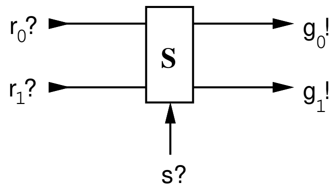
As it was the case with the mutual exclusion scheduler, there is a small difference between our specification and tacit assumptions in the original problem statement: Originally, the forks are assumed to be initially on the table. Our specification, on the other hand, requires that all forks initially signal their availability. The reason for this difference is a (tacit) assumption in the committee problem statement that, in the beginning, none of the professors is ready for a meeting.

We have shown that three known synchronization/exclusion problems can be interpreted as special cases of the committee problem. Now, we focus on some delay-insensitive components and show that they actually behave as schedulers for special cases of the committee problem.

3.4 Sequencers

We begin with the SEQUENCER, one of the basic components we described in the previous chapter. The specification and the schematic of the SEQUENCER are

$$\begin{array}{ll} & \mathbf{pref} *[r_0?; g_0!] \\ \| & \mathbf{pref} *[r_1?; g_1!] \\ \| & \mathbf{pref} *[s?; (\, g_0! \mid g_1! \,)] \end{array}$$



Messages r_0, r_1, and s can be interpreted as ready messages sent by professors, and messages g_0 and g_1 can be seen as signals that the committee meetings can begin. The underlying committee configuration then consists of two committees, $c_0 = \{r_0, s\}$ and $c_1 = \{r_1, s\}$.

We interpreted the SEQUENCER as a basic component that upon receiving (synchronization) signal s arbitrates between pending inputs r_0 and r_1, if the arbitration is necessary. As we will see later, it is often convenient to have a component that upon receiving a special (synchronization) signal arbitrates among up to n pending requests (where $n \geq 2$). Because such a component can be seen as a generalization (in the number of inputs) of the SEQUENCER, we can extend the name SEQUENCER to a class of components, called n-input SEQUENCERs.

$$
\begin{array}{l}
\quad \mathbf{pref} *[r_0?; g_0!] \\
\| \ \mathbf{pref} *[r_1?; g_1!] \\
\vdots \\
\| \ \mathbf{pref} *[r_{n-1}?; g_{n-1}!] \\
\| \ \mathbf{pref} *[s?; (g_0! \mid \ldots \mid g_{n-1}!)]
\end{array}
$$

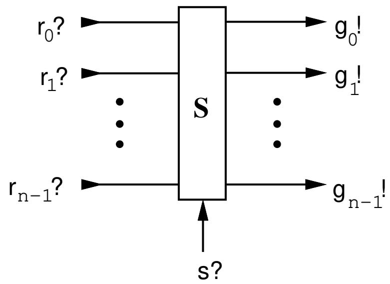


Figure 3.4: n-input SEQUENCER.

The specification of an n-input SEQUENCER shown in Figure 3.4 is an obvious extension of the specification of the 2-input SEQUENCER. The operation of the n-input SEQUENCER can also be interpreted as a special case of the committee problem: inputs r_i, $0 \leq i < n$, and s represent ready signals from professors, and outputs g_i, $0 \leq i < n$, represent signals to begin meetings of committees C_i, that consist of two members: $C_i = \{r_i, s\}$. Since the professor who sends signal s is a member of all committees (probably a department chair), only one committee can meet at a time.

Note that the specification of the n-input SEQUENCER is the same (after the appropriate renaming of the symbols) as the specification of the mutual exclusion scheduler. This confirms that the SEQUENCER is in fact a basic component implementing mutual exclusion. In Section 5.4 we will discuss how an n-input SEQUENCER can be implemented by a network of basic components.

3.5 Arbitrating Test-and-Set

Another delay-insensitive component that can be interpreted as a special case committee problem scheduler is the *Arbitrating Test-and-Set* (ATS) component introduced by Keller [Kel74]. Its specification and schematic are given in Figure 3.5.

$$\mathrm{ATS} \quad = \quad \begin{array}{l} \quad \mathbf{pref} *[t?;(t_0!|t_1!)] \\ \| \ \mathbf{pref} *[r?;t_0!] \end{array}$$

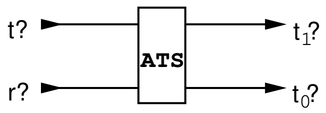


Figure 3.5: Arbitrating Test-and-Set.

During its operation the component may be either "set" or "reset". Upon receiving input t, the component responds with t_0, if it is "reset", or with t_1, if it is "set". Furthermore, after receiving input r, the component becomes "reset" and remains in this state until output t_0 is produced. After outputting t_0 the component becomes "set" again. Note that the component does not necessarily become "reset" immediately upon receiving input r, but may become "reset" after an unspecified delay. Initially the ATS component is "set".

Our specification of the ATS component prescribes a behavior that is different from that in the original definition; state graphs for our specification and for Keller's specification are are shown in Figure 3.6. We can see that Keller's specification requires that, after input r is received, the component immediately becomes "reset", i.e., it responds with t_0 on the first input t it receives after r. In terms of commands, Keller's specification would be written as

$$\mathrm{ATS}_{\mathsf{K}} = \mathbf{pref} *[\ *[t?;t_1!];\ \underline{r?;t?};t_0!]$$

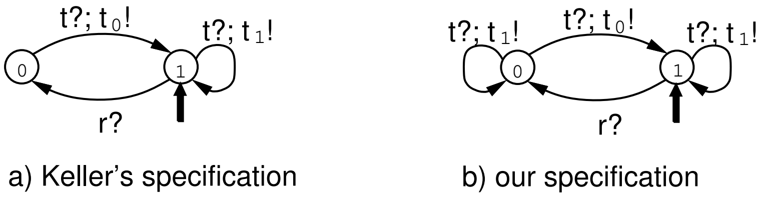


Figure 3.6: ATS — different specifications.

Our specification, however, allows a component to remain "set" after it has received input r.

There is an important reason for our use of a different specification: We want the ATS to be a DI component. Informally, a DI component is a component that is not affected by delays in wires that connect the component to its environment. Because of the two (underlined) consecutive inputs, the component as proposed by Keller is not delay-insensitive. The basic reason is that the order in which Keller's component receives inputs r and t determines its behavior. However, if we insert delays in the input wires, the order in which the component receives r and t may not be the same as the order in which the environment sends these two inputs to the component. Therefore, the behavior of the component may be affected by the delays in the connecting wires.

In order to interpret the behavior of the ATS in the context of the committee scheduler, we consider inputs r and t as the ready messages from professors R and T, and outputs t_0 and t_1 as signals for members of committees C_0 and C_1 to attend a meeting. The rosters of the committees are $C_0 = \{R, T\}$ and $C_1 = \{T\}$.

3.6 General case

Let $P = \{p_0, \ldots, p_{n-1}\}$ be a set of n professors, and let $C = \{c_0, \ldots, c_{m-1}\}$ be a set of m committees. The set of committees in which professor p_i participates is called a *committee set* and is referred to as C_i. We assume that each professor is a member of at least one committee and that each committee has at least one member.

In order to obtain a specification of the general committee scheduler, we apply the same technique as we did with the special cases: We examine the communication between the scheduler and each professor p_i, $0 \leq i < n$, in isolation, and then combine all constraints into a specification. When professor p_i is ready to attend a meeting, he or she sends ready message r_i to the scheduler. After the scheduler has determined that a committee c_j can meet, where $c_j \in C_i$, it sends a message m_j to the professor. This means that, after receiving r_i, the scheduler may send to the professor any of the messages m_j, where $c_j \in C_i$. We specify such a partial behavior as

$$\mathsf{constraint}_i = \mathbf{pref} *[r_i?;\ (\,|j :\ c_j \in C_i :\ m_j!\,)]$$

For example, if professor p_i is a member of committees c_i, c_j, and c_k, then the partial behavior is

$$\mathsf{constraint}_i = \mathbf{pref} *[r_i?;\ (\,m_i! \mid m_j! \mid m_k!\,)]$$

The complete behavior of the scheduler is the conjunction of the n partial behaviors:

$$\begin{aligned} \mathsf{scheduler} = \quad & \mathbf{pref} *[r_0?;\ (\,|j :\ c_j \in C_0 :\ m_j!\,)] \\ & \| \ \mathbf{pref} *[r_1?;\ (\,|j :\ c_j \in C_1 :\ m_j!\,)] \\ & \vdots \\ & \| \ \mathbf{pref} *[r_{n-1}?;\ (\,|j :\ c_j \in C_{n-1} :\ m_j!\,)] \end{aligned} \tag{3.4}$$

The key issue in this specification is synchronization on common symbols. When committee c_i meets, all members of the committee synchronize on signal m_i.

Example: Let $C = \{c_0, c_1, c_2\}$ be the set of committees, and let $P = \{p_0, p_1, p_2, p_3, p_4\}$ be the set of professors. Furthermore, let the committee rosters be

$$c_0 \;=\; \{p_0, p_1, p_2\} \qquad c_1 \;=\; \{p_1, p_3\} \qquad c_2 \;=\; \{p_0, p_4\}$$

The committee sets of the professors are

$$\begin{aligned} C_0 &= \{c_0, c_2\} & \qquad C_3 &= \{c_1\} \\ C_1 &= \{c_0, c_1\} & \qquad C_4 &= \{c_2\} \\ C_2 &= \{c_0\} \end{aligned}$$

This configuration gives us the following specification:

$$\begin{aligned} \mathsf{scheduler} = \quad & \mathbf{pref} *[r_0?;\ (m_0! \,|\, m_2!)\,] \\ & \|\ \mathbf{pref} *[r_2?;\ (m_0! \,|\, m_1!)\,] \\ & \|\ \mathbf{pref} *[r_3?;\ m_0!\,] \\ & \|\ \mathbf{pref} *[r_4?;\ m_1!\,] \\ & \|\ \mathbf{pref} *[r_5?;\ m_2!\,] \end{aligned}$$

□

The scheduler may have to arbitrate among a number of committees that are all ready to meet, but cannot meet at the same time. Suppose, for example, that the scheduler above has received ready messages from professors p_0, p_2, and p_3. If the next ready message comes from professor p_1, there are two committees that could meet, namely c_0 and c_1. However, because professor p_1 is a member of both committees, they cannot meet at the same time. Therefore, the scheduler must arbitrate among two possible meetings. In Sections 6.2.2, 6.2.5 we will discuss two arbitration schemes and their implementations by means of primitive components.

3.6.1 Specification using Petri nets

Petri nets are a formalism that offers an attractive visualization of the synchronization of parallel processes. Since there is a close resemblance between our commands and a Petri net specification [EBG93], we have been using them to provide pictorial equivalents of our specifications. Let us see how the committee scheduler for the example above can be specified by a Petri net, shown in Figure 3.7.

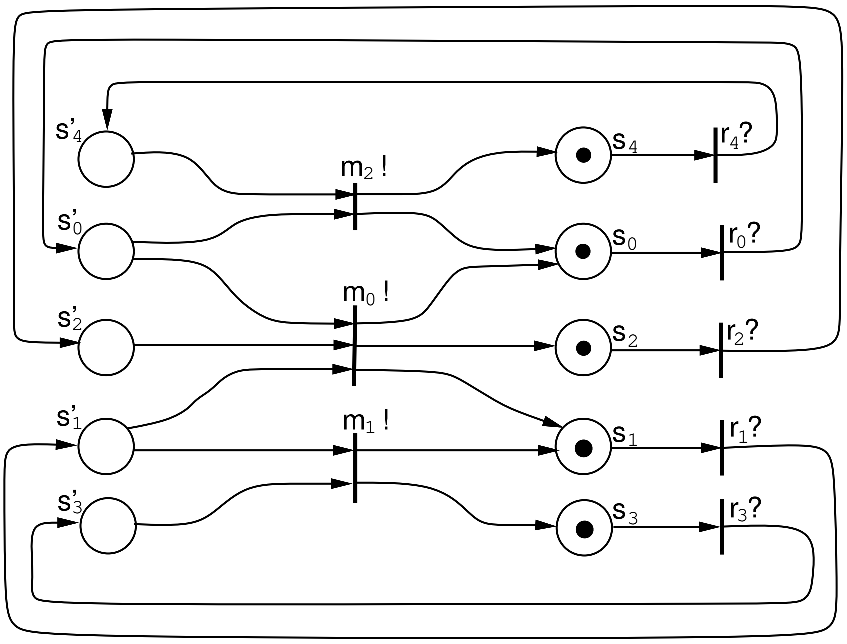


Figure 3.7: Petri net for the committee scheduler.

Again, we use the transitions in a Petri net to denote messages, transmitted or received by the scheduler. There are two sets of transitions, one for the ready messages $r_0, \ldots, r_4$, sent by professors, and the other for signals m_1, m_2, and m_3, sent by the scheduler to indicate beginnings of meetings. Furthermore, we use two sets of places to denote the current states of the professors. Place s_i', where

$i \in \{0, \ldots, 4\}$, holds a token after professor i has transmitted his ready message, which means he or she is waiting to attend a meeting. When place s_i holds a token, where $i \in \{0, \ldots, 4\}$, professor i is attending a meeting, or he or she is resting before signalling his or her readiness to attend the next meeting. Initially, we assign one token to each of the places $s_0, \ldots, s_4$. From their initial places tokens may travel to places s'_0, $\ldots$, s'_4 via corresponding transitions $r_0, \ldots, r_4$. Places $s'_0, \ldots, s'_4$ are inputs for transitions m_0, m_1,and m_2, with place s'_i being an input to all transitions that correspond to committees in the committee set of professor i. Note that transition m_i can fire only if all input places of m_i hold a token, which means that all members of committee c_i are waiting.

Chapter 4

Decomposition rules

After obtaining a specification, we would like to implement it as a network of basic components. In this chapter we formalize this idea by defining the notion of decomposition. We state two theorems, the *Substitution* Theorem and the *Separation* Theorem, that help us to derive and to verify a decomposition. The Substitution Theorem allows us to decompose a specification in a hierarchical way. By means of the Separation Theorem we can decompose parts of a specification separately. We conclude the chapter with a formalization of the concept of delay-insensitivity; the definitions of DI decomposition and DI component are then presented. The definitions and theorems in this chapter are taken from [Ebe89].

4.1 Decomposition

First, we state the conditions for a decomposition of specification E into a network of components E_i, $1 \leq i \leq n$. When we reason about a component we refer to a regular, non-empty, prefix-closed trace structure that prescribes a certain communication behavior, and not to a physical implementation of a specification.

Notation: We denote a network consisting of components $E.1, E.2, \ldots, E.n$ by $(E.1, E.2, \ldots, E.n)$, or by $(i : 1 \leq i \leq n \ : E.i)$. We interpret $(i : 1 \leq i \leq n \ : E.i)$ as an enumeration of components $E.i$, where the order of enumeration is not important. With $E \rightarrow (i : 1 \leq i \leq n \ : E.i)$ we denote the property that E can be decomposed into network $(E.1, E.2, \ldots, E.n)$. □

In Section 2.3 we stipulated that a specification prescribes the behavior of both a component and its environment. More precisely, it specifies when a component may produce outputs and when the environment may produce inputs. Note that inputs and outputs are always considered from the component's point of view. If network $(i : 1 \leq i \leq n \ : E.i)$ is an implementation of specification E we assume that the environment of this network still behaves as prescribed by E. Such an environment can be seen as a component $\overline{E}$, where $\overline{E}$ denotes the *reflection* of E, defined as $\langle \mathbf{o}E, \mathbf{i}E, \mathbf{t}E \rangle$. Since $\mathbf{i}\overline{E} = \mathbf{o}E$, $\mathbf{o}\overline{E} = \mathbf{i}E$, and $\mathbf{t}\overline{E} = \mathbf{t}E$, the reflection can be seen as interchanging the roles of the component and its environment. Consequently, in order to comprise all components that produce outputs relevant to the decomposition, we do not consider $\overline{E}$ and network $(i : 1 \leq i \leq n \ : E.i)$ separately, but we rather consider network $(i : 0 \leq i \leq n \ : E.i)$, where $E.0 = \overline{E}$.

Definition 4.1.1 (Decomposition) *Let $n \geq 1$ and let E and $E.i$, where $0 \leq i \leq n$, be components. Furthermore, let $E.0 = \overline{E}$, and $W = (\|i : 0 \leq i \leq n \ : E.i)$.*

We say that component E can be decomposed into components $E.i$, $1 \leq i \leq n$, denoted by

$$E \rightarrow (i : 1 \leq i \leq n \ : E.i)$$

if the following conditions are satisfied:

(1) *Closed network:* $(\cup i : 0 \leq i \leq n : \mathbf{o}E.i) = (\cup i : 0 \leq i \leq n : \mathbf{i}E.i)$.

(2) *No output interference:* $\mathbf{o}E.i \cap \mathbf{o}E.j = \emptyset$ *for* $0 \leq i, j \leq n$ *and* $i \neq j$.

(3) *Network is free of computation interference: For all traces* t*, symbols* x *and indices* i*, where* $0 \leq i \leq n$*, we have*

$$t \in \mathbf{t}W \ \wedge \ x \in \mathbf{o}E.i \ \wedge \ tx \downarrow \mathbf{a}E.i \in \mathbf{t}E.i \quad \Longrightarrow \quad tx \in \mathbf{t}W.$$

(4) *Network behaves as specified at boundary* $\mathbf{a}E$*:*

$$\mathbf{t}W \downarrow \mathbf{a}E = \mathbf{t}E$$

.

Condition (1) requires that every input in a network be connected to an output and that every output be connected to an input. We call such a network a *closed network*.

Condition (2) requires that outputs of distinct components be not connected to each other. This means that each symbol may be produced by at most one component. We say that such a network is *free of output interference*. Note, however, that inputs may be connected to each other. Conditions (1) and (2) together guarantee that each symbol is produced by at most one component and received by at least one component.

Conditions (1) and (2) are concerned with the structure of a network and are expressed in terms of the alphabets of the trace structures. On the other hand, conditions (3) and (4) are behavioral conditions and are therefore phrased in terms of trace sets and alphabets.

Condition (3) requires that every output that can be produced by a component in a network can also be accepted by another component. In other words, no environment prescription of a component may be violated. Recall, that $t \in \mathbf{t}W$ means that joint behavior t is in accordance with the behaviors of all components in the network. If we want to extend t with symbol x, there must be a component $E.i$ that can produce x after a subbehavior $t \downarrow \mathbf{a}E.i$, i.e.,

$$(t \downarrow \mathbf{a}E.i)x \in \mathbf{t}E.i \ \wedge \ x \in \mathbf{o}E.i$$

Note that we use projection to extract the behavior of a single component from joint behavior t. Furthermore, for all components $E.j$, $0 \le j \le n$, it must be true that

$$tx \downarrow \mathbf{a}E.j \in \mathbf{t}E.j\,,$$

that is, for all j, $0 \le j < n$, if $x \in \mathbf{i}E.j$, then $(t \downarrow \mathbf{a}E.j)x \in \mathbf{t}E.j$.

Operationally speaking, a violation of an environment prescription of a component may cause hazardous behavior. For example, if the WIRE, given by **pref** $*[a?;b!]$, receives two inputs a without producing output b, we have computation interference caused by the environment. An operational interpretation of this computation interference is that more than one transition is propagating along a physical wire, which can cause a hazardous behavior and must be avoided.

Condition (4) says that a network must behave as specified by $\mathbf{t}E$ at the boundary $\mathbf{a}E$. In other words, we require that the network communicates with its environment in exactly the same way as it is prescribed in the specification. This condition excludes decompositions into a so-called "accept-everything-do-nothing" module, that is, into a component that accepts everything and never produces any output. On the other hand, satisfying condition (4) does not guarantee that every trace will occur. It allows, for example, a decomposition that has a possibility of deadlock or livelock. We encounter this problem in Chapters 5 and 6.

We use these four conditions as a correctness criteria for a decomposition. Although the conditions are simple to verify, the verification may be tedious for large components and networks. Fortunately, we can use an automatic verifier VERDECT [EG93b], described in Section 4.6. The time complexity of a straightforward (or flat) verification can be exponential in the number of components in a network, but we can decrease it significantly by applying the Substitution Theorem and Separation Theorems, introduced later in this chapter.

We remark that we see a decomposition as a goal-directed activity. We start with specification E and try to find its decomposition, all under the assumption that an environment behaves as specified in E. We do not start with a network of components, trying to find out what a network could do without any requirement for its environment.

Before we give an example of the decomposition, let us introduce the notation we use in the derivation of a decomposition:

$$\begin{array}{ll} & E \\ = & \{\ \text{Introduction of internal symbols A}\ \} \\ & |[\, A :: EE \,]| \\ \rightarrow & \{\ \text{Syntax-directed decomposition}\ \} \\ & (E.1, E.2). \end{array}$$

Each step in the derivation is explained by a short hint given between braces. The hint "Introduction of internal symbols" indicates that EE is an expansion of E obtained by inserting the internal symbols from A in E. Finding a proper expansion is often the most important step in the derivation of a decomposition.

The hint "Syntax-directed decomposition" indicates that the conditions for decomposition can be verified by inspecting the syntax of EE and of E_1 and E_2. The conditions for a closed network and absence of output interference can be verified by checking the alphabets. Absence of computation interference can be verified by checking that EE represents all network behaviors (note that the network consists of $\overline{EE}$, E_1, and E_2) and that $\mathbf{t}EE \downarrow \mathbf{a}E_i \subseteq \mathbf{t}E_i$, where $i \in \{1,2\}$. Complete behavior of the network is then automatically satisfied, since EE is an expansion of E.

Example: Consider the following component:

$$E = \mathbf{pref} *[a?; b!; c?; d!].$$

We observe that every output b is preceded by input a and that every output d is preceded by input c. Furthermore, the environment is not allowed to produce c before the component produces output b, and is not allowed to produce a before the component produce output d (except in the beginning). From this we can conclude that component E can be decomposed into a network consisting of two WIREs:

$$\begin{array}{ll}
 & E \\
= & \quad \{ \text{ By definition } \} \\
 & \mathbf{pref} *[a?; b!; c?; d!] \\
\rightarrow & \quad \{ \text{ Syntax-directed decomposition } \} \\
 & (\, \mathbf{pref} *[a?; b!], \mathbf{pref} *[c?; d!] \,)
\end{array}$$

□

4.2 Substitution Theorem

The Substitution Theorem enables a top-down approach in obtaining a decomposition, that is, it allows us to decompose a specification by stepwise refinement. The Substitution Theorem applies to problems of the following kind: Suppose that specification E can be decomposed into a number of components, F being one of them. Moreover, component F can be further decomposed into a number of components. The question is: Under what conditions can we substitute component F by its decomposition in a decomposition of E ?

Theorem 4.2.1 (Substitution Theorem, [Ebe89]) *For components E, $E.1$, $E.2$, $E.3$, and F we have:*

$$\begin{aligned} & (\, E \rightarrow (E.1, F)\,) \;\wedge\; (\, F \rightarrow (E.2, E.3)\,) \\ \Longrightarrow\;\; & E \rightarrow (E.1, E.2, E.3) \end{aligned}$$

if the following condition is satisfied.

$$(\mathbf{a}E.1 \cup \mathbf{a}F) \;\cap\; (\mathbf{a}E.2 \cup \mathbf{a}E.3) \;=\; \mathbf{a}F \tag{4.1}$$

Condition 4.1 states that the only symbols that decompositions of E and F have in common are symbols from $\mathbf{a}F$. This is in fact a trivial condition since it can always be satisfied by an appropriate renaming of internal symbols in the decomposition of F.

Note that Theorem 4.2.1 considers a decomposition into two components only. The generalization to decompositions into n, $n \geq 2$, components is straightforward and omitted here.

Example: The following is an example of a decomposition by stepwise refinement. Each of the individual steps can be verified in isolation.

$$\begin{array}{ll} & \textbf{pref } *[b?;(q! \mid p1!;a0?;q!)\,] \\ \rightarrow & \quad \{ \text{ definition of decomposition } \} \\ & (\textbf{ pref } *[b?;(q1! \mid p1!;a0?;q0!)]\ ,\ \textbf{pref } *[(q1?|q0?);q!]\ ) \end{array}$$

and

$$\begin{array}{ll} & \textbf{pref } *[b?;(q1! \mid p1!;a0?;q0!)\,] \\ \rightarrow & \quad \{ \text{ definition of decomposition } \} \\ & (\textbf{ pref } *[b?;(q1!|p1!)]\ ,\ \textbf{pref } *[a0?;q0!]\ ) \end{array}$$

Since we did not introduce any new symbols in the second decomposition, Condition 4.1 is automatically satisfied. Thus we can apply the Substitution Theorem to these two decompositions and get

$$\begin{array}{ll} & \textbf{pref } *[b?;(q! \mid p1!;a0?;q!)] \\ \rightarrow & \quad \{ \text{ the first decomposition } \} \\ & (\textbf{pref } *[b?;(q1! \mid p1!;a0?;q0!)]\ ,\ \textbf{pref } *[(q1?|q0?);q!]) \\ \rightarrow & \quad \{ \text{ the second decomposition, the Substitution Theorem } \} \\ & (\underbrace{\textbf{pref } *[b?;(q1!|p1!)]}_{\text{SELECTOR}}\ ,\ \underbrace{\textbf{pref } *[a0?;q0!]}_{\text{WIRE}}\ ,\ \underbrace{\textbf{pref } *[(q1?|q0?);q!]}_{\text{MERGE}}) \end{array}$$

We are already familiar with the WIRE and the MERGE, two of the components that form the final decomposition. The third component, the SELECTOR, is sometimes also considered as one of the basic components. Its behavior is nondeterministic; after receiving an input (b) from its environment it nondeterministically chooses one of its two output terminals ($q1$, $p1$) and then produces an output at the chosen terminal. This behavior may then repeat. □

4.3 Separation Theorem

The Separation Theorem allows finding a decomposition of a specification by partwise refinement. The theorem applies to specifications that are stated as a weave $E.0 \| R.0$. We can first try to find decompositions for $E.0$ and $R.0$ separately, and then obtain a decomposition for $E.0 \| R.0$ by collecting commands in the separate decompositions and weaving those that have common outputs.

Theorem 4.3.1 (Separation Theorem, [Ebe89]) *For components $E.i$ and $F.i$, with $0 \le i \le n$, $n \ge 1$, we have*

$$\begin{array}{ll} & E.0 \to (i:\ 1 \le i \le n:\ E.i) \ \wedge\ F.0 \to (i:\ 1 \le i \le n:\ F.i) \\ \Longrightarrow & E.0 \| F.0 \to (i:\ 1 \le i \le n:\ E.i \| F.i) \end{array}$$

if the following conditions are satisfied.

$$(\cup i:\ 1 \le i \le n:\ \mathbf{a}E.i) \cap (\cup i:\ 1 \le i \le n:\ \mathbf{a}F.i) \subseteq \mathbf{a}E.0 \cap \mathbf{a}F.0; \tag{4.2}$$

$$\begin{array}{lclcll} (\mathbf{o}E.i \cup \mathbf{o}F.i) & \cap & (\mathbf{o}E.j \cup \mathbf{o}F.j) & = & \emptyset & \text{for } 1 \le i,j \le n \text{ and } i \ne j \\ (\mathbf{i}E.0 \cup \mathbf{i}F.0) & \cap & (\mathbf{o}E.i \cup \mathbf{o}F.i) & = & \emptyset & \text{for } 1 \le i \le n. \end{array} \tag{4.3}$$

Condition 4.2 states that all the symbols that the decompositions of $E.0$ and $F.0$ have in common are external symbols, that is, elements of $\mathbf{a}E.0 \cap \mathbf{a}F.0$. In other words, condition 4.2 requires that the sets of internal symbols of the two decompositions be disjoint. Condition 4.2 is in fact trivial because it can be always satisfied with an appropriate renaming of internal symbols.

Condition 4.3 states that pairs of woven commands, $E.i \| F.i$, have disjoint outputs. This condition may be satisfied by rearranging commands in decompositions of $E.0$ and $F.0$ and by introducing ε, the empty trace, at appropriate places. Note that ε has the role of the identity in a decomposition, that is,

$$\text{if} \quad E \to (E.1, E.2) \ \text{ then also } \quad E \to (E.1, E.2, \varepsilon).$$

Note also, that the order in which we list the components in network $(E.1, \ldots, E.n)$ does not affect the correctness of the decomposition $E \rightarrow (E.1, \ldots, E.n)$.

Example: Let us show how we can apply the Separation Theorem to obtain a decomposition of an "initialized" JOIN specified by

$$\mathbf{pref}(a?; c!; *[(a?\|b?); c!])$$

The initialized JOIN first waits for input a only. After the first c has been produced, it behaves as our standard JOIN. One can verify that the specification above can be rewritten as

$$\mathbf{pref} *[a?; c!] \;\|\; \mathbf{pref} *[c!; b?]$$

The second specification of the initialized JOIN is a weave of two commands, which suggests the Separation Theorem. The weavands can be decomposed as

$$\begin{aligned} \mathbf{pref} *[a?; c!] &\rightarrow (\mathbf{pref} *[a?; c!], \varepsilon) \\ \mathbf{pref} *[c!; b?] &\rightarrow (\mathbf{pref} *[y?; c!], \mathbf{pref} *[y!; b?]) \end{aligned}$$

Since there are no internal symbols in the first decomposition (y is an internal symbol in the second decomposition), the decompositions do not have a common internal symbol. Therefore, the first condition in the Separation Theorem is satisfied. In order to satisfy the second condition, we have to weave the commands with a common output. Since the first decomposition consists of one command only, we extend it with the identity ε to get $\mathbf{pref} *[a?; c!] \rightarrow (\mathbf{pref} *[a?; c!], \varepsilon)$. Finally, we can apply the Separation Theorem and obtain the decomposition

$$\mathbf{pref} *[a?; c!] \;\|\; \mathbf{pref} *[c!; b?] \rightarrow (\underbrace{\mathbf{pref} *[a?; c!] \,\|\, \mathbf{pref} *[y?; c!]}_{\text{JOIN}}, \underbrace{\mathbf{pref} *[y!; b?]}_{\text{IWIRE}})$$

We can recognize the components we get in the decomposition above as two of our basic components, IWIRE and JOIN. A schematic of the decomposition is shown in Figure 4.1.□

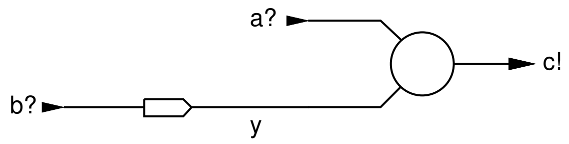


Figure 4.1: Decomposition of an "initialized" JOIN.

According to condition 4.2, components $E.0$ and $F.0$ may not have a common internal symbol if we want to apply the Separation Theorem. This restriction may be inconvenient when we deal with complex components. Fortunately, there exists a more general version of the Separation Theorem [Ebe91b].

In the following $|[\, A \,::\, E, F \,]|$ denotes a *network* consisting of components E and F, where the symbols in A are *local* to the network.

Theorem 4.3.2 (Separation Theorem, generalized [Ebe91b])

For components $E.i$, and $F.i$, with $0 \le i \le n$, $n \ge 1$, and for a list of symbols A we have the following. If

$$\begin{array}{lcl} |[\, A \,::\, E.0 \,]| & \to & |[\, A \,::\, (i : 1 \le i \le n : E.i) \,]| \\ |[\, A \,::\, F.0 \,]| & \to & |[\, A \,::\, (i : 1 \le i \le n : F.i) \,]| \end{array} \tag{4.4}$$

with complete network behaviors $E.0$ and $F.0$

and if

$$\begin{array}{l} (\mathbf{o}E.i \cup \mathbf{o}F.i) \cap (\mathbf{o}E.j \cup \mathbf{o}F.j) = \emptyset \text{ for } 1 \le i,j \le n \text{ and } i \ne j \\ (\mathbf{i}E.0 \cup \mathbf{i}F.0) \cap (\mathbf{o}E.i \cup \mathbf{o}F.i) = \emptyset \text{ for } 1 \le i \le n \end{array} \tag{4.5}$$

then

$$|[\, A \,::\, E.0 \| F.0 \,]| \to |[\, A \,::\, (i : 1 \le i \le n : E.i \| F.i) \,]|$$

Condition 4.5 is the same as Condition 4.3 and states that pairs of woven commands have disjoint outputs. Condition 4.4 does not require that the sets of internal symbols of the decompositions of $E.0$ and $F.0$ be disjoint. Rather, it puts a stronger requirement on network behaviors: Normally, a network is a decomposition if it *externally* exhibits the same behaviors as given in the specification. Here we strengthened this requirement to one where the network behaviors of a decomposition and the behaviors of the specification $E.0$ (or $F.0$) have to be the same with respect to the external symbols *and* the symbols in A.

Both versions of the Separation Theorem can be generalized in a natural way to decompositions of components that are specified as weaves of more than two trace structures. In the following we do not explicitly indicate which version of the Separation Theorem we are applying; it can be easily determined from a context.

4.4 Delay-insensitivity

We introduced the notion of decomposition in order to formalize the implementation of a specification by a network of components. Operationally, we may see a decomposition as an implementation of a circuit by a number of subcircuits. One of our assumptions was that there may be an arbitrary delay in a subcircuit between receipt of an input and production of an output. We also assumed that sending and receipt of a signal between two subcircuits coincide, i.e., a communication is instantaneous. Such circuits are called *speed-independent* circuits, because their functional correctness of is not affected by speed of the components (i.e., by delays in response times).

Physical circuits, however, are connected by wires with unspecified delays. This makes our assumption about instantaneous communication unrealistic, unless con-

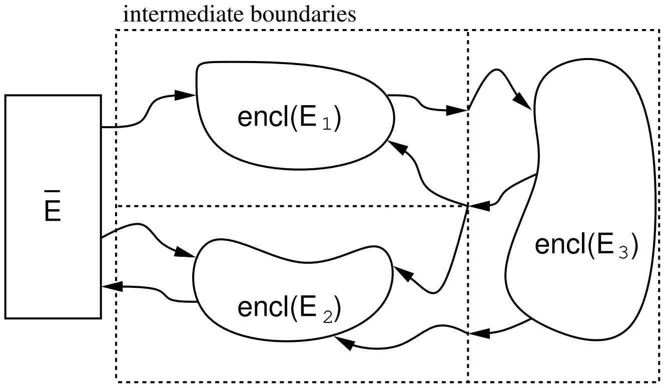


Figure 4.2: DI decomposition.

nection wires are explicitly included in a network of communicating components. For this reason, we introduce *DI decomposition*, where components are not connected directly, but rather through one or two WIREs via an intermediate terminal. The set of intermediate terminals is called *intermediate boundary* of a component. The intermediate boundary is shown as a dotted line in Figure 4.2.

Operationally speaking, WIREs introduce delays in communications. These delays may affect the correctness of a decomposition, because, for example, the order in which a component receives inputs is not the same as the order in which the inputs were sent. If a network with additional WIREs satisfies the 4 conditions from the definition of decomposition, then we call such a network a *delay-insensitive* circuit.

In order to formalize delay-insensitivity, we begin with a definition of the *enclosure* of component E, denoted by $enc(E)$. The enclosure of a component is a component that has its external symbols renamed to their localized versions, so it can be seen as a component enclosed into its intermediate boundary.

Definition 4.4.1 (Enclosure) *Let E be a trace structure. The enclosure of E, denoted by $enc(E)$ is a trace structure such that*

$$\begin{array}{rcl} \mathbf{i}(enc(E)) & = & \{\mathcal{R}(a) \mid a \in \mathbf{i}E\} \\ \mathbf{o}(enc(E)) & = & \{\mathcal{R}(a) \mid a \in \mathbf{o}E\} \\ \mathbf{t}(enc(E)) & = & \{\mathcal{R}(t) \mid t \in \mathbf{t}E\} \end{array}$$

and $\mathcal{R}$ is a homomorphism defined as

$$\mathcal{R}(a) = \begin{cases} a^o & \text{if } a \in \mathbf{o}E \\ a^i & \text{if } a \in \mathbf{i}E \end{cases}$$

Because $enc(E)$ can be connected to its environment only by additional WIREs, we add WIRE $Wire(a)$ for each terminal $a \in \mathbf{a}E$; it connects the enclosure of the component and its intermediate boundary:

$$Wire(a) = \begin{cases} \mathbf{pref} *[a^o?; a!] & \text{if } a \in \mathbf{o}E \\ \mathbf{pref} *[a?; a^i!] & \text{if } a \in \mathbf{i}E \end{cases}$$

A list of all WIREs that connect $enc(E)$ and its intermediate boundary is denoted by $Wires(E)$. Now we can state our definition of a DI decomposition.

Definition 4.4.2 (DI decomposition) *We say that components $E.k$, $1 \leq i \leq n$, form a DI decomposition of component $E.0$, denoted by*

$$E.0 \xrightarrow{DI} (i : 1 \leq i \leq n : E.i)$$

if and only if

$$E.0 \rightarrow (i : 1 \leq n \leq i : enc(E.i), Wires(E.i)) \tag{4.6}$$

The name DI decomposition suggests that it is a special case of decomposition. Unfortunately, this is not true, because there are DI decompositions that are not decompositions. For example, consider components $E.0 = \mathbf{pref} *[a!\|b!]$ and $E.1 = \mathbf{pref} *[a!; b!]$. It is true that $E.1$ is a DI decomposition of $E.0$, but it is not a decomposition of $E.0$. The reason is that additional WIREs enlarge the set of possible behaviors. Two outputs, for example, that are ordered at the component's boundary may be unordered at its intermediate boundary, because of delays in the added WIREs.

However, there is a special case when the two decompositions are equivalent; this is true when all constituting components are so-called DI components. We have

Definition 4.4.3 (DI component) *Component E is a* DI component, *if*

$$E \rightarrow (enc(E), \mathit{Wires}(E))$$

Theorem 4.4.4 *If all components $E.i$, $1 \le i \le n$, are DI components, then*

$$E.0 \rightarrow (i:\ 1 \le i \le n : E.i) \quad \equiv \quad E.0 \xrightarrow{DI} (i:\ 1 \le i \le n : E.i)$$

The insensitivity to delays in communications between a component and environment, formalized in Definition 4.4.3, is often stated as the so-called *Foam Rubber Wrapper* (FRW) property. We can imagine a component as being wrapped in a foam rubber with flexible boundaries. Operationally speaking, the FRW property says that a component is invariant under the deformation of its boundary. In our case, the FRW boundaries are formed by $\mathbf{a}(enc(E))$ and $\mathbf{a}E$, as shown in Figure 4.3. The FRW principle originates from Charles Molnar and was formalized in terms of trace structures by Udding [Udd86].

There are a number of ways to verify whether a given component is a DI component: we can use Definition 4.4.3, we can use the rules given by Udding [Udd86],

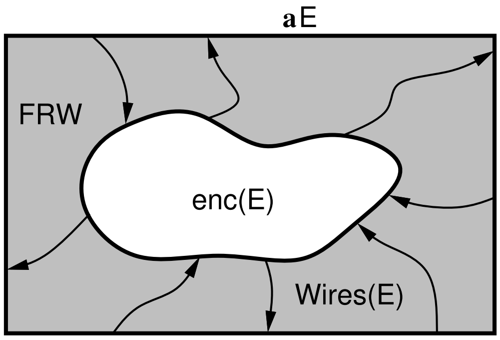


Figure 4.3: The Foam Rubber Wrapper property.

or, we can use the DI grammar [Ebe89], which generates commands representing DI components. Definition 4.4.3 and Udding's rules are equivalent, while for a DI grammar it is a conjecture that the grammar generates commands for a proper subset of DI components. Let us mention that all basic components we described so far are DI components.

4.5 Performance analysis

We define two criteria to measure the efficiency of a decomposition: the *area complexity* and the *response time*. We use standard $\mathcal{O}$, Ω, and Θ notations to denote asymptotic upper bound, asymptotic lower bound, and asymptotically tight bound for a function.

Definition 4.5.1 (Area complexity) *Let N be a network $(i : 1 \leq i \leq n : E.i)$. Then the area complexity of N, denoted by $\alpha(N)$, is defined as*

$$\alpha(N) = (\,+i : 1 \leq i \leq n : \alpha(E.i)\,)$$

We assume that the area complexity of a basic component is bounded by a constant. Furthermore, because we deal with delay-insensitive circuits where WIREs may be inserted (or deleted) without affecting the correctness of a decomposition, we assume that the area complexity of a WIRE is 0. This means that our definition of area complexity corresponds to an ideal situation, where wires take no additional space. This is, of course, a very crude measure because chips consist mostly of wires. Another deficiency in our assumptions is that we do not precisely define area complexities of basic components. The underlying reason is that there is no generally accepted set of basic components, and that there are no generally accepted implementations of basic components, which makes comparisons difficult. For these reasons, we see the area complexity more as a topological rather than a geometric measure.

We define the response time $\mathcal{T}(N)$ of network N as the *maximum* delay between any output and the immediately preceding input, where the response time is measured in response times of basic components. Thus, the response time is a *worst case* complexity and gives an upper bound on actual response times. Again, we assume that the response time of a *nonarbitrating* basic component is bounded by a constant and we do not define it precisely for each particular component. Because there is no physical implementation of an arbiter that has a bounded response time, we do not make any assumptions about the response time of the SEQUENCER. Because we deal with delay-insensitive circuits, we also assume that a WIRE has response time 0. As was the case with area complexity, the response time is not necessarily an accurate estimate of the actual response time of the circuit and should be seen more as a topological than as a time-metrical measure.

4.6 Verification tool: VERDECT

The verification of the four conditions of our definition of a decomposition may be quite tedious for a large specification or a large network; therefore, there is a strong need for an automatic verifier.

A verification tool called VERDECT [EG93b] has been developed to verify the four conditions of a decomposition. Its inputs are specifications written in terms of commands described in Chapter 2. The simplest usage of VERDECT is to do a flat verification of a decomposition. However, the time and the space complexity of such a verification can be exponential in the number of components in a network: The verifier constructs a deterministic finite automaton for all network behaviors, and the number of states of such an automaton may be exponential in the number of components. Fortunately, the application of the Substitution and the Separation Theorems can make a verification much more efficient. In [EG93b] experimental results show that a significant reduction of the number of states can be achieved when stepwise (Substitution Theorem) and partwise (Separation Theorem) verification is used.

Unless stated otherwise, all decomposition examples in the following chapters were verified using VERDECT.

Chapter 5

Schedulers for special cases

In this chapter we propose decompositions of schedulers for the special cases of the committee problem described in Chapter 3. Decompositions of the special-case schedulers are more efficient than a decomposition of a general scheduler. In Chapter 6 we show how decompositions for some special cases can be used as parts of a general decomposition. In Section 5.1 we relate phase synchronization to a generalized JOIN; in Section 5.2 we show a decomposition of the scheduler for the dining philosophers problem; in Section 5.3 we decompose the ATS component; finally, in Section 5.4 we discuss decompositions of an n-input SEQUENCER.

5.1 Phase synchronization

In order to obtain a decomposition for the phase synchronization scheduler we begin with a simple instance of the problem, where we consider only two processes. A specification of the corresponding scheduler is

$$\begin{aligned} \mathsf{phase\_synch}_2 = \quad & \mathbf{pref} * [r_1?; z!] \\ \| \; & \mathbf{pref} * [r_2?; z!] \end{aligned}$$

We recognize the specification above as the specification of the JOIN, one of our basic components. This suggests that the scheduler for the phase synchronization problem can be implemented as the n-input JOIN. The schematic of the n-input JOIN is shown in Figure 5.1. The n-input JOIN can be implemented easily as a

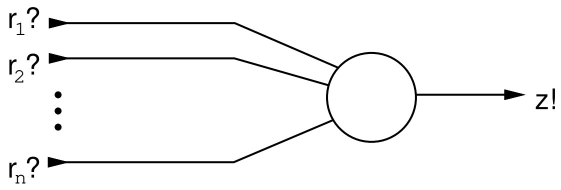


Figure 5.1: n-input JOIN.

network of 2-input JOINs connected as a binary tree. The inputs of the leaves are the inputs of the n-input JOIN, and the output of the root is the output of the n-input JOIN. Each node in the tree is a component that synchronizes the outputs of its subtrees (see Figure 5.2).

As an example, Figure 5.3 shows a decomposition of the 4-input JOIN. Such a decomposition is not difficult to derive. A specification of the 4-input JOIN is given by

$$\begin{aligned} \mathrm{JOIN}_4 = \quad & \mathbf{pref} * [r_1?; z!] \\ \| \; & \mathbf{pref} * [r_2?; z!] \\ \| \; & \mathbf{pref} * [r_3?; z!] \\ \| \; & \mathbf{pref} * [r_4?; z!] \end{aligned}$$

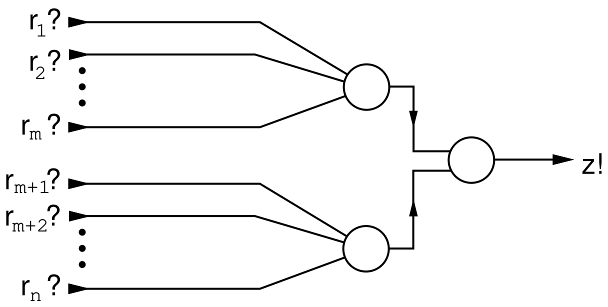


Figure 5.2: A decomposition of the n-input JOIN.

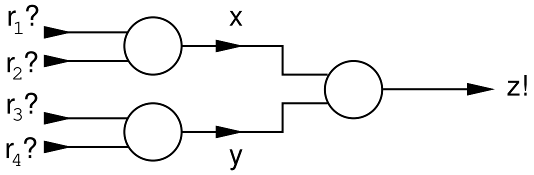


Figure 5.3: A decomposition of the 4-input JOIN.

The specification can be seen as a weave $E \| F$, where

$$E \;=\; \textbf{pref} *[r_1?; z!] \;\|\; \textbf{pref} *[r_2?; z!]$$

and

$$F \;=\; \textbf{pref} *[r_3?; z!] \;\|\; \textbf{pref} *[r_4?; z!]$$

This form of specification suggests an application of the Separation Theorem. First, we can decompose the two weavands as

$$\begin{array}{lcl} E & \to & (\,\textbf{pref} *[r_1?; x!] \,\|\, \textbf{pref} * [r_2?; x!],\;\; \varepsilon,\;\; \textbf{pref} *[x?; z!]) \\ F & \to & (\varepsilon,\;\; \textbf{pref} *[r_3?; y!] \,\|\, \textbf{pref} * [r_4?; y!],\;\; \textbf{pref} *[y?; z!]\,) \end{array}$$

Now, we apply the Separation Theorem and get the decomposition

$$\begin{aligned}\text{JOIN}_4 \;\rightarrow\; (\;\; & \mathbf{pref}*[r_1?;x!] \,\|\, \mathbf{pref}*[r_2?;x!],\\ & \mathbf{pref}*[r_3?;y!] \,\|\, \mathbf{pref}*[r_4?;y!],\\ & \mathbf{pref}*[x?;z!] \,\|\, \mathbf{pref}*[y?;z!] \quad)\end{aligned}$$

We can recognize all three components in the decomposition above as 2-input JOINs. We can derive a decomposition of the n-input JOIN in a similar way. Because a balanced binary tree with n leaves has $\mathcal{O}(n)$ nodes, the area complexity of such a decomposition is $\mathcal{O}(n)$. The height of a balanced binary tree with n leaves is $\mathcal{O}(\log n)$. This means that a delay between an output and an immediately preceding input corresponds to the sum of delays in $\mathcal{O}(\log n)$ basic components, which gives the response time $\mathcal{O}(\log n)$.

5.2 Dining philosophers

Let us recall the specification of the scheduler for the dining philosophers problem we gave in Section 3.3:

$$\begin{aligned}\mathsf{dining}_3 = \quad & \mathbf{pref}*[f_0?;(e_0!|e_1!)]\\ \| \; & \mathbf{pref}*[f_1?;(e_1!\,|\,e_2!)]\\ \| \; & \mathbf{pref}*[f_2?;(e_2!\,|\,e_0!)]\\ \| \; & \mathbf{pref}*[h_0?;\; e_0!]\\ \| \; & \mathbf{pref}*[h_1?;\; e_1!]\\ \| \; & \mathbf{pref}*[h_2?;\; e_2!]\end{aligned}$$

A decomposition of the scheduler is shown in Figure 5.4. The decomposition resembles Dijkstra's solution of the dining-philosophers problem [Dij71] using semaphores: Each of the SEQUENCERs in fact implements a binary semaphore. Furthermore, the

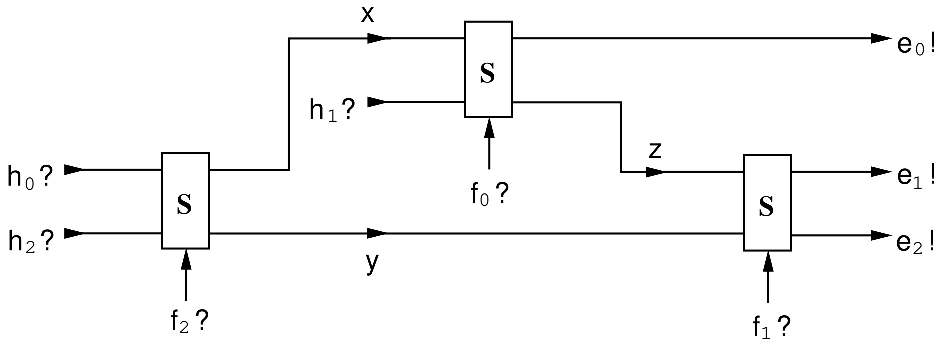


Figure 5.4: A decomposition of the dining philosophers scheduler.

asymmetric usage of the semaphores, which prevents the occurrence of deadlock in Dijkstra's solution, is also present in our decomposition: Philosopher 2 acquires the fork on his left side first, and the other two philosophers first acquire forks to their right.

The decomposition given in Figure 5.4 was obtained by taking an idea from a known algorithm and using it as a guideline to design a circuit. Given a tentative circuit, we have to verify that all the correctness conditions are satisfied. This can be done, for example, by using an automatic verifier. On the other hand, we could approach the problem formally and give a derivation of the decomposition. The derivation we give below can be generalized easily and then used to obtain a decomposition of the dining-philosophers scheduler for an arbitrary number of philosophers.

Derivation

Since we specified the scheduler as a weave of commands, we aim to apply the Separation Theorem in order to obtain a decomposition. First, we rewrite the specification as

$$\begin{array}{lll}
\mathsf{dining}_3 = & \textbf{pref}\, *[f_0?;(\,e_0!|e_1!\,)] \;\|\; \textbf{pref}\, *[h_1?;\; e_1!] & \{E.0\} \\
& \|\; \textbf{pref}\, *[f_1?;(e_1!\,|\,e_2!)] & \{E.1\} \\
& \|\; \textbf{pref}\, *[f_2?;(e_2!\,|\,e_0!)] \;\|\; \textbf{pref}\, *[h_0?;\; e_0!] \;\|\; \textbf{pref}\, *[h_2?;\; e_2!] & \{E.2\}
\end{array}$$

Now we have the specification written as a weave of 3 trace structures:

$$\mathsf{dining}_3 = E.0 \,\|\, E.1 \,\|\, E.2$$

We can decompose the weavands as follows:

$$\begin{array}{llllll}
E.0 & \rightarrow & (& \textbf{pref}\, *[f_0?;\; (\,e_0!\,|\,z!\,)] \,\|\, \textbf{pref}\, *[h_1?; z!], & \textbf{pref}\, *[z?; e_1!], & \varepsilon \;\;) \\
E.1 & \rightarrow & (& \varepsilon, & \textbf{pref}\, *[f_1?;\; (e_1!\,|\,e_2!)], & \varepsilon \;\;) \\
E.2 & \rightarrow & (& \textbf{pref}\, *[x?; e_0!], & \textbf{pref}\, *[y?; e_2!], & \\
& & & \multicolumn{3}{r}{\textbf{pref}\, *[f_2?;\; (x!\,|\,y!)] \,\|\, \textbf{pref}\, *[h_0?; x!] \,\|\, \textbf{pref}\, *[h_2?; y!] \;\;)}
\end{array}$$

All decompositions above are straightforward to verify.

According to the Separation Theorem, we can weave the components with common outputs and obtain the final decomposition.

Decomposition 5.2.1 (Dining philosophers)

$$\begin{array}{lll}
\mathsf{dining}_3 = & (& \textbf{pref}\, *[x?; e_0!] \,\|\, \textbf{pref}\, *[h_1?; z!] \,\|\, \textbf{pref}\, *[f_0?;\; (\,e_0!\,|\,z!\,)], \\
& & \textbf{pref}\, *[z?; e_1!] \,\|\, \textbf{pref}\, *[y?; e_2!] \,\|\, \textbf{pref}\, *[[f_1?;\; (e_1!\,|\,e_2!)], \\
& & \textbf{pref}\, *[h_0?; x!] \,\|\, \textbf{pref}\, *[h_2?; y!] \,\|\, \textbf{pref}\, *[f_2?;\; (x!\,|\,y!)] \qquad)
\end{array}$$

We can recognize all three components in Decomposition 5.2.1 as the SEQUENCERs connected as shown in Figure 5.4. Generalization of the decomposition to the case with n philosophers is straightforward; Figure 5.5, for example, shows a decomposition of a scheduler for 5 philosophers.

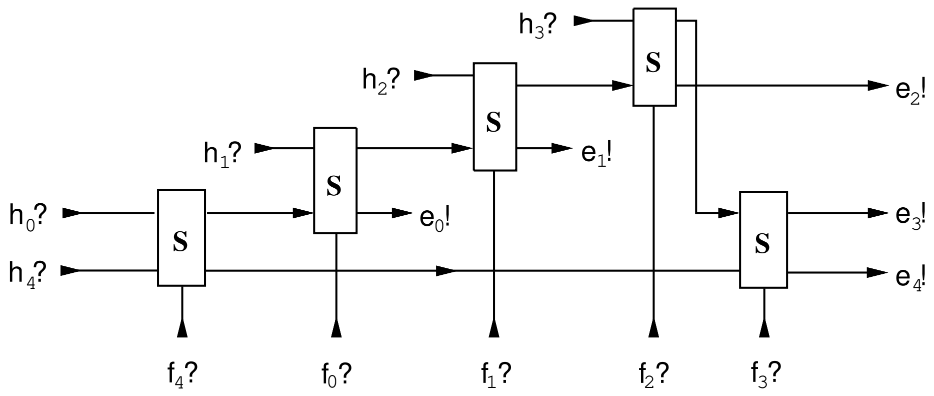


Figure 5.5: A scheduler for 5 dining philosophers.

5.3 Arbitrating Test-and-Set

We can treat the ATS as one of the basic components, or we can implement it using the components given in Figure 2.2. Its specification is very similar to the specification of the SEQUENCER. Therefore, in our first step we "expand" the ATS to the SEQUENCER and add an additional component to satisfy the correctness conditions:

$$
\begin{array}{ll}
 & \text{ATS} \\
= & \quad \{ \text{ specification } \} \\
 & \mathbf{pref} *[t?;\, (\, t_0! \mid t_1!\,)] \parallel \mathbf{pref} *[r?; t_0!] \\
= & \quad \{ \text{ introducing internal symbol } x \text{ } \} \\
 & |[\, x :: \\
 & \quad \mathbf{pref} *[t?;\, (t_0! \mid t_1)\,] \\
 & \ \parallel \mathbf{pref} *[x;\, t_1!] \\
 & \ \parallel \mathbf{pref} *[r?; t_0!] \\
 &]|
\end{array}
$$

$\rightarrow$ { identifying a sequencer }

$$
\begin{array}{ll}
(\ \mathbf{pref} *[t?; (t_0! \mid t_1!)] \parallel \mathbf{pref} *[r?; t_0!] \parallel \mathbf{pref} *[x?; t_1!], & \{\text{SEQUENCER}\} \\
\ \ \mathbf{pref} *[x!; t1?] \ \ ) & \{\text{IWIRE}\}
\end{array}
$$

$\rightarrow$ { introducing a WIRE }

$$
\begin{array}{ll}
(\ \mathbf{pref} *[t?; (t_0! \mid z!)] \parallel \mathbf{pref} *[r?; t_0!] \parallel \mathbf{pref} *[x?; z!], & \{\text{SEQUENCER}\} \\
\ \ \mathbf{pref} *[x!; z?] & \{\text{IWIRE}\} \\
\ \ \mathbf{pref} *[z?; t_1!] \ \ ) & \{\text{WIRE}\}
\end{array}
$$

A schematic of the decomposition is shown in Figure 5.6. Note that z is an output of the SEQUENCER and also serves as an input of the IWIRE and the WIRE. Such a fork is usually called an *isochronic fork* and cannot always be replaced by a genuine FORK. In our case, however, all the components to which the isochronic fork is connected are DI components; hence, we can replace the isochronic fork by a genuine FORK [Ebe89].

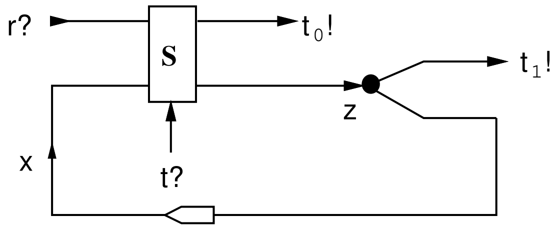


Figure 5.6: A decomposition of the ATS.

We remark that trace $xzx\ldots$ in the network above can be seen as the traveling of a token around a ring. Our interpretation of the specification of the SEQUENCER (Section 2.3) does not guarantee that request r will be granted eventually. This means that a token may keep circulating around the ring. For example, $r(txzt_1)^*$

is a subset of the possible behaviors of the network. A situation in which request r may never be granted is usually called *starvation*. This example demonstrates that fairness (i.e., there is no starvation) is not implied by our correctness conditions.

5.4 Sequencers

The n-input SEQUENCER can be seen as a generalized SEQUENCER. It can also be seen as a committee scheduler for the mutual exclusion problem, as we showed in Section 3.4. Here we discuss two approaches to the decomposition of the n-input SEQUENCER. Both approaches demonstrate similarities between the design of parallel programs and the design of delay-insensitive circuits.

5.4.1 Decompositions using tokens

Recall, that the n-input SEQUENCER is a component which, upon receiving signal s, arbitrates between pending requests r_i, $1 \leq i \leq n$, if arbitration is necessary. Let us begin with a derivation of a decomposition of the 3-input SEQUENCER:

$$
\begin{array}{ll}
 & \text{SEQUENCER}_3 \\
= & \quad \{ \text{ specification } \} \\
 & \quad \mathbf{pref} *[r_1?; g_1!] \parallel \mathbf{pref} *[r_2?; g_2!] \parallel \mathbf{pref} *[r_3?; g_3!] \\
 & \parallel \mathbf{pref} *[s?;\ (g_1! \mid g_2! \mid g_3!)\,] \\
= & \quad \{ \text{ introducing internal symbol } t_1 \text{ } \} \\
 & |[\ t_1 :: \\
 & \quad\ \mathbf{pref} *[r_1?; g_1!] \parallel \mathbf{pref} *[r_2?; g_2!] \parallel \mathbf{pref} *[r_3?; g_3!] \\
 & \quad \parallel \mathbf{pref} *[s?;\ (\,g_1! \mid t_1; (g_2! \mid g_3!\,)\,)\,] \\
 &]|
\end{array}
$$

$$
\begin{array}{lll}
\rightarrow & \{ \text{ syntax-directed decomposition } \} & \\
& (\ \mathbf{pref}\ *[r_1?; g_1!] \parallel \mathbf{pref}\ *[s?; (g_1! \mid t_1!)], & \{\text{ATS}\} \\
& \ \ \mathbf{pref}\ *[r_2?; g_2!] \parallel \mathbf{pref}\ *[r_3?; g_3!] \parallel \mathbf{pref}\ *[t_1?; (g_2! \mid g_3!)]\ ) & \{\text{SEQUENCER}\}
\end{array}
$$

Thus, we have decomposed the 3-input SEQUENCER into a network consisting of a SEQUENCER and an ATS; the latter can be treated as a basic component or can be decomposed further. Using the same reasoning, we can decompose the SEQUENCER:

$$
\begin{array}{lll}
& \mathbf{pref}\ *[r_2?; g_2!] \parallel \mathbf{pref}\ *[r_3?; g_3!] \parallel \mathbf{pref}\ *[t_1?; (g_2! \mid g_3!)] & \\
= & \{ \text{ introducing internal symbol } t_2 \ \} & \\
& |[\ t_2 :: & \\
& \quad \mathbf{pref}\ *[r_2?; g_2!] \parallel \mathbf{pref}\ *[r_3?; g_3!] & \\
& \ \parallel \mathbf{pref}\ *[t_1?; (g_2! \mid t_2; g_3!\,)\,] & \\
&]| & \\
\rightarrow & \{ \text{ syntax-directed decomposition } \} & \\
& (\ \mathbf{pref}\ *[r_2?; g_2!] \parallel \mathbf{pref}\ *[t_1?; (g_2! \mid t_2!)], & \{\text{ATS}\} \\
& \ \ \mathbf{pref}\ *[r_3?; g_3!] \parallel \mathbf{pref}\ *[t_2?; g_3!]\ \ ) & \{\text{JOIN}\}
\end{array}
$$

At this point we can apply the Substitution Theorem and get the following decomposition of the 3-input SEQUENCER:

$$
\begin{array}{llll}
\text{SEQUENCER}_3 \ \rightarrow & (& \mathbf{pref}\ *[r_1?; g_1!] \parallel \mathbf{pref}\ *[s?; (g_1! | t_1!)], & \{\text{ATS}\} \\
& & \mathbf{pref}\ *[r_2?; g_2!] \parallel \mathbf{pref}\ *[t_1?; (g_2! \,|\, t_2!)], & \{\text{ATS}\} \\
& & \mathbf{pref}\ *[r_3?; g_3!] \parallel \mathbf{pref}\ *[t_2?; g_3!] \qquad) & \{\text{JOIN}\}
\end{array}
$$

The decomposition is shown in Figure 5.7. A generalization of the decomposition to n inputs is straightforward.

The network above satisfies all our criteria for a decomposition of a specification. However, there is a possibility of an undesired behavior: namely, unnecessary

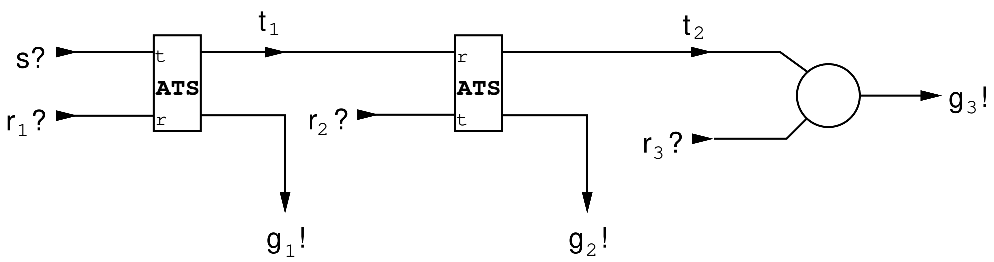


Figure 5.7: A token-bus decomposition of the 3-input SEQUENCER.

blocking. Consider the network's behavior after trace st_1t_2. At this point requests r_1 and r_2 cannot be granted before issuing a grant for request r_3, which has not arrived. This means, that if r_3 never arrives, r_1 and r_2 will never be granted.

Before we try to "correct" the decomposition, let us describe it informally. The ATS components and the JOIN can be seen as the controllers C_i for requests r_i, $1 \leq i \leq 3$ (n in general), and the receipt of input s can be seen as a receipt of a token. When controller C_i has a token and request r_i is pending, it may either grant the request and destroy the token, or it may pass the token to controller C_{i+1}. If request r_i is not pending, controller C_i may send the token to the next controller. The only exception is controller C_3, which cannot send a token to its successor: it waits until r_3 arrives and then grants the request. Such a structure can be called a *token bus*.

It seems that the problem with the token-bus decomposition lies in the fact that, once a token has passed a controller, it never comes back. We can overcome this problem by allowing the last controller in the bus to send a token back to the first controller, i.e. by using a token ring instead of a token bus. This modification gives us the following decomposition for the 3-input SEQUENCER

$$
\begin{array}{llll}
\text{SEQUENCER}_3 & \rightarrow & (& \mathbf{pref} *[r_1?; g_1!] \parallel \mathbf{pref} *[t_0?; (g_1!|t_1!)], & \{\text{ATS}\} \\
 & & & \mathbf{pref} *[r_2?; g_2!] \parallel \mathbf{pref} *[t_1?; (g_2! \mid t_2!)], & \{\text{ATS}\} \\
 & & & \mathbf{pref} *[r_3?; g_3!] \parallel \mathbf{pref} *[t_2?; (g_3! \mid t_3!)], & \{\text{ATS}\} \\
 & & & \mathbf{pref} *[(s? \mid t_3?); t_0!] \quad) & \{\text{MERGE}\}
\end{array}
$$

The decomposition is shown in Figure 5.8. To obtain the token-ring decomposition

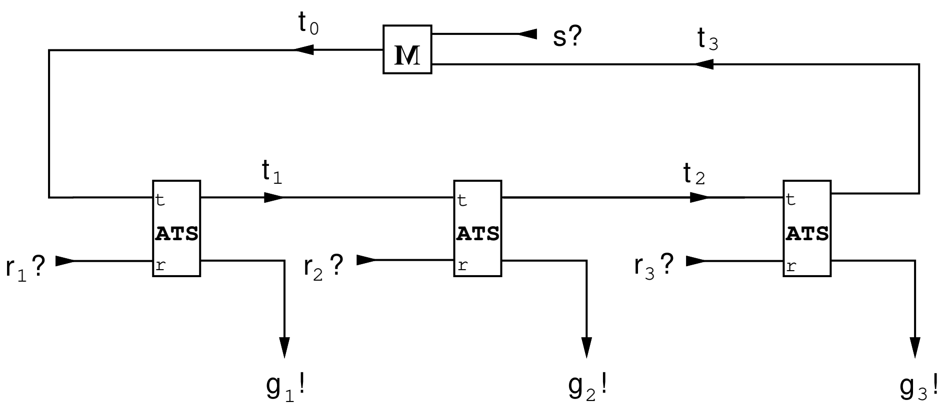


Figure 5.8: A token-ring decomposition of the 3-input SEQUENCER.

we made two changes in the token-bus decomposition: (1) we replaced the JOIN by an ATS that allows the last controller to pass on the token, and (2), we added the MERGE, which enables the passing of a token from the last ATS to the first. Again, it is straightforward to generalize the decomposition to n requests.

With the introduction of the ring we eliminated the possibility of unnecessary blocking in the network. However, the new decomposition now suffers from livelock: Once a token is generated it may circulate around the ring without granting any of the pending requests. Recall (Section 2.3) that our correctness criteria do not allow an implementation of an arbiter that always grants the same request. However, our correctness criteria also do not guarantee that a certain trace will occur during the operation of a component. Therefore, the behavior of the ATS which consists of always passing a token to the next controller is valid according to our specification.

The area complexity of both token-based decompositions is $\mathcal{O}(n)$, where n is a number of inputs of the SEQUENCER. Because of a possible metastable behavior of the ATS components, nothing can be said about the response time. Finally, we remark that the token-ring decomposition is very similar to the decomposition of the LOCK-UNLOCK MODULE, given in [Kel74].

5.4.2 Tree decomposition

Since we cannot be satisfied with the token-based decompositions, we take a different approach, namely, a tree decomposition. The idea for such a decomposition originates from [Sei80] and was used to obtain a decomposition of the n-input SEQUENCER in [Ebe89].

Recall that the n-input SEQUENCER is a component which, upon receipt of the synchronization input s, grants one of the pending requests. If there are more than one pending requests, arbitration takes place.

First, we divide requests $r_1, \ldots, r_n$ into two sets, $L = \{r_i \mid 1 \leq i \leq m\}$ and $R = \{r_i \mid m < i \leq n\}$, with $1 < m < n$ (we assume $n > 2$, otherwise the decomposition is trivial). Now, we apply the divide-and-conquer paradigm and decompose the n-input SEQUENCER into a network consisting of three components: the SEQUENCER, and components A_m and A_{n-m} (see Figure 5.9).

Informally, the role of component A_m is to select one of the pending inputs r_i, where $r_i \in L$, and send output x as a request to the SEQUENCER. Similarly, component A_{n-m} selects one of the pending inputs r_j, where $r_j \in R$, and sends request y

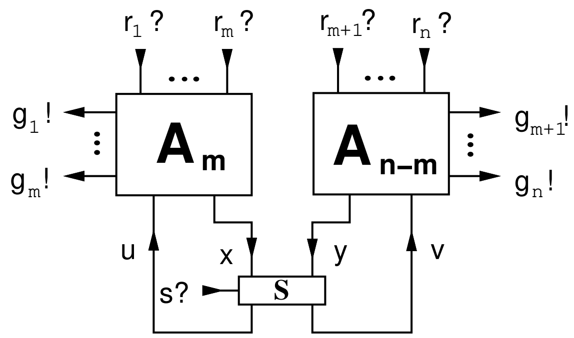


Figure 5.9: A divide-and-conquer decomposition of an n-input SEQUENCER.

to the SEQUENCER. Upon receipt of the synchronization signal s, the SEQUENCER grants either request x (by sending output u), or request y (by sending output v). When component A_m receives input u, it outputs the grant for the selected request and then selects the next pending request. The behavior of component A_{n-m} is similar.

We implemented the selection of a pending request in two steps. First, components A_m and A_{n-m} make a selection from the sets L and R, respectively. Then, if necessary, a selection is made between the two sets.

We begin a derivation of this decomposition by introducing internal symbols x_i and f_i for every request r_i. Furthermore, we separate the requests in two groups by introducing internal symbols u and v:

$$
\begin{array}{ll}
 & \text{SEQUENCER}_n \\
= & \quad \{ \text{ introducing internal symbols } \} \\
 & |[\; (i \,:\, 1 \le i \le n \,:\, x_i, f_i), x, y, u, v \;:: \\
 & \left.\begin{array}{l} \quad (\| \, i \,:\, 1 \le i \le m \,:\, \mathbf{pref} * [\, r_i?; x_i; f_i; g_i! \,] \,) \\ \; \| \; \mathbf{pref} * [\, | \, i \,:\, 1 \le i \le m \,:\, x_i; x; u; f_i \,] \end{array} \right\} E.0
\end{array}
$$

$$\left.\begin{array}{l} \| \, (\| \, j \,:\, m < j \leq n \,:\, \mathbf{pref} *[\, r_j?; x_j; f_j; g_j!] \,) \\ \| \, \mathbf{pref} *[\, | \, j \,:\, m < j \leq n \,:\, x_j; y; v; f_j \,] \end{array}\right\} E.1$$

$$\left.\begin{array}{l} \| \, \mathbf{pref} *[\, s?; \, (\, (\, | \, i \,:\, 1 \leq i \leq m \,:\, u; g_i! \,) \\ \qquad\qquad | \, (\, | \, j \,:\, m < j \leq n \,:\, v; g_j! \,) \,) \,] \end{array}\right\} E.2$$

$$]|$$

Since the specification is a weave of three trace structures, we aim to apply the Separation Theorem. First we have to decompose each of the three weavands. In order to decompose $E.0$ we observe that x and u alternate in every trace in $\mathbf{t}(E.0)$. This observation leads to the following decomposition:

$$\begin{array}{ll} |[\, (i \,:\, 1 \leq i \leq m \,:\, x_i, f_i), x, u \;:: & \{E.0\} \\ \quad (\| \, i \,:\, 1 \leq i \leq m \,:\, \mathbf{pref} *[\, r_i?; x_i; f_i; g_i!] \,) & \\ \quad \| \, \mathbf{pref} *[\, | \, i \,:\, 1 \leq i \leq m \,:\, x_i; x; u; f_i \,] & \\ ]| & \\ \rightarrow \quad \{ \text{ syntax-directed decomposition } \} & \\ (& \\ \quad |[\, (i \,:\, 1 \leq i \leq m \,:\, x_i, f_i) \;:: & \\ \qquad (\| \, i \,:\, 1 \leq i \leq m \,:\, \mathbf{pref} *[\, r_i?; x_i; f_i; g_i!] \,) & \\ \qquad \| \, \mathbf{pref} *[\, | \, i \,:\, 1 \leq i \leq m \,:\, x_i; x!; u?; f_i \,] & \\ \quad]|, & \\ \quad \mathbf{pref} *[x?; u!] & \\ ) & \end{array}$$

As with the decomposition above, we observe that y and v alternate in each trace in $\mathbf{t}(E.1)$. This gives the following decomposition:

$$\begin{array}{ll} |[\, (j \,:\, m < j \leq n \,:\, x_j, f_j), y, v \;:: & \{E.1\} \\ \quad (\| \, j \,:\, m < j \leq n \,:\, \mathbf{pref} *[\, r_j?; x_j; f_j; g_j!] \,) & \end{array}$$

$$
\begin{array}{l}
\quad \| \ \mathbf{pref} *[\,|\,j \,:\, m < j \leq n \,:\, x_j; y; v; f_j\,] \\
]| \\
\rightarrow \quad \{ \text{ syntax-directed decomposition } \} \\
(\\
\quad |[\ (j \,:\, m < j \leq n \,:\, x_j, f_j) \ :: \\
\qquad (\|\, j \,:\, m < j \leq n \,:\, \mathbf{pref} *[\, r_j?; x_j; f_j; g_j!]\,) \\
\quad\ \ \| \ \mathbf{pref} *[\,|\,j \,:\, m < j \leq n \,:\, x_j; y!; v?; f_j\,] \\
\quad]|, \\
\quad \mathbf{pref} *[y?; v!] \\
)
\end{array}
$$

Note that in the behavior of $E.2$ every production of g_i, where $1 \leq i \leq m$, is preceded by a production of u, and every production of g_j, where $m < j \leq n$, is preceded by a production of v. Furthermore, every production of u od v is preceded by a production of s. These observations lead to the following decomposition:

$$
\begin{array}{ll}
|[\ u, v :: & \{E.2\} \\
\quad \mathbf{pref} *[\, s?; (\,(\,|\,i \,:\, 1 \leq i \leq m \,:\, u; g_i!\,) & \\
\qquad\qquad |\,(\,|\,j \,:\, m < j \leq n \,:\, v; g_j!\,)\,)\,] & \\
]| & \\
\rightarrow \quad \{ \text{ syntax-directed decomposition } \} & \\
(& \\
\quad \mathbf{pref} *[u?; (|\ i \,:\, 1 \leq i \leq m \,:\, g_i!)\,], & \\
\quad \mathbf{pref} *[v?; (|\ j \,:\, m < j \leq n \,:\, g_j!)\,], & \\
\quad \mathbf{pref} *[s?; (u! \mid v!),] & \\
) &
\end{array}
$$

Now we apply the Separation Theorem and obtain the following decomposition:

$$
\begin{array}{ll}
& \text{SEQUENCER}_n \\
\rightarrow & \{ \text{ decomposition by the Separation Theorem } \} \\
& (\\
& |[\, (i \,:\, 1 \le i \le m \,:\, x_i, f_i) :: \qquad\qquad \{X_m\} \\
& \quad (\| \, i \,:\, 1 \le i \le m \,:\, \textbf{pref} *[\, r_i?; x_i; f_i; g_i!]\,) \\
& \quad \| \, \textbf{pref} *[\, |\, i \,:\, 1 \le i \le m \,:\, x_i; x!; u?; f_i\,] \\
& \quad \| \, \textbf{pref} *[u?; (|\, i \,:\, 1 \le i \le m \,:\, g_i!)\,] \\
&]|, \\
& |[\, (i \,:\, m < j \le n \,:\, x_j, f_j) :: \qquad\qquad \{X_{n-m}\} \\
& \quad (\| \, j \,:\, m < j \le n \,:\, \textbf{pref} *[\, r_j?; x_j; f_j; g_j!]\,) \\
& \quad \| \, \textbf{pref} *[\, |\, j \,:\, m < j \le n \,:\, x_j; y!; v?; f_j\,] \\
& \quad \| \, \textbf{pref} *[v?; (|\, j \,:\, m < j \le n \,:\, g_j!)\,] \\
&]|, \\
& \quad \textbf{pref} *[x?; u!] \qquad\qquad \{\text{SEQUENCER}\} \\
& \| \, \textbf{pref} *[y?; v!] \\
& \| \, \textbf{pref} *[s?; (u! \mid v!)\,] \\
&)
\end{array}
$$

We can see that one of the weavands in component X_k, given by

$$
\begin{array}{lll}
X_k & = & |[(i \,:\, 1 \le i \le k \,:\, x_i, f_i) \;:: \\
& & \quad (\, \| \, i \,:\, 1 \le i \le k \,:\, \textbf{pref} *[r_i?; x_i; f_i; g_i!]\,) \\
& & \quad \| \, \textbf{pref} *[\, (\, |\, i \,:\, 1 \le i \le k \,:\, x_i; x!; u?; f_i)\,] \\
& & \quad \| \, \textbf{pref} *[u?; (|\, i \,:\, 1 \le i \le k \,:\, g_i!)\,] \qquad \{ \longleftarrow \text{ redundant } \} \\
& &]|
\end{array}
$$

is redundant. Thus, X_k can be decomposed into a “simpler” component A_k, given by

$$\begin{array}{rcl} A_k & = & |[(i \,:\, 1 \le i \le k \,:\, x_i, f_i) \;:: \\ & & \quad (\,\|\, i \,:\, 1 \le i \le k \,:\, \mathbf{pref} * [r_i?; x_i; f_i; g_i!] \,) \\ & & \quad \| \; \mathbf{pref} * [\,(\,|\, i \,:\, 1 \le i \le k \,:\, x_i; x!; u?; f_i)\,] \\ & &]| \end{array}$$

In the decomposition above we may apply the Substitution Theorem and replace X_m and X_{n-m} by A_m and A_{n-m}. After this replacement we get

Decomposition 5.4.1 (n-input SEQUENCER, step 1)

$$\begin{array}{lll} & \textsc{sequencer}_n & \\ \to & \{ \text{ decomposition, first step } \} & \\ & (& \\ & \mathbf{pref} * [x?; u!] \,\|\, \mathbf{pref} * [y?; v!] \,\|\, \mathbf{pref} * [s?; \, (u! \,|\, v!)\,], & \{\textsc{sequencer}\} \\ & |[\, (i \,:\, 1 \le i \le m \,:\, x_i, f_i) \;:: & \{A_m\} \\ & \quad (\| \, i \,:\, 1 \le i \le m \,:\, \mathbf{pref} * [r_i?; x_i; f_i; g_i!] \,) & \\ & \quad \| \; \mathbf{pref} * [(\,|\, i \,:\, 1 \le i \le m \,:\, x_i; x!; u?; f_i \,)] & \\ &]|, & \\ & |[\, (j \,:\, m < j \le n \,:\, x_j, f_j) \;:: & \{A_{n-m}\} \\ & \quad (\| \, j \,:\, m < j \le n \,:\, \mathbf{pref} * [\, r_j?; x_j; f_j; g_j!] \,) & \\ & \quad \| \; \mathbf{pref} * [\,(\,|\, j \,:\, m < j \le n \,:\, x_j; y!; v?; f_j \,)\,] & \\ &]| & \\ &) & \end{array}$$

Because components A_m and A_{n-m} are not among our basic components, we decompose them further. Since the only difference between the specifications of A_m and A_{n-m} is the number of requests and grants, one can conclude they belong to the same class of components; we call them subSEQUENCERs.

One can verify that in every trace of component A_1, given by

$$A_1 = |[\, x_1, f_1 \;::\; \mathbf{pref} *[r_1?; x_1; f_1; g_1!] \parallel \mathbf{pref} *[x_1; x!; u?; f_1] \;\;]|$$

every input r_1 is followed by output x, and every input u is followed by output g_1. Therefore, A_1 can be decomposed into a network consisting of two WIREs:

Decomposition 5.4.2 (1-input subSEQUENCER)

$$A_1 \;\;\rightarrow\;\; (\quad \mathbf{pref} *[r_1?; x!], \quad \mathbf{pref} *[u?; g_1!] \quad)$$

The decomposition of subSEQUENCER A_2, given by

$$\begin{aligned} A_2 \;\;=\;\; & |[\, x1, x2, f_1, f_2 \;:: \\ & \qquad \mathbf{pref} *[r_1?; x_1; f_1; g_1!] \\ & \quad \parallel \mathbf{pref} *[r_2?; x_2; f_2; g_2!] \\ & \quad \parallel \mathbf{pref} *[x_1; x!; u?; f_1 \mid x_2; x!; u?; f_2] \\ &]| \end{aligned}$$

is more complicated (see Figure 5.10). The reason is that A_1 does not perform any arbitration, whereas A_2 may need to arbitrate between two requests. Let us explain the decomposition intuitively. The basic component we can use for arbitration between two signals is the SEQUENCER with inputs r_1 and r_2, and with outputs x_1 and x_2. Initially, the SEQUENCER's synchronization signal z must be present, so we connect an IWIRE to it. Furthermore, after a request has been selected, the subSEQUENCER may produce output x. Therefore, we connect the outputs of the SEQUENCER to a MERGE with output x. Since the SEQUENCER grants only one request at a time, the environment constraints for the MERGE are not violated. We also have to remember which request was selected by the SEQUENCER, because upon receipt of input u, the subSEQUENCER may output the grant for the selected

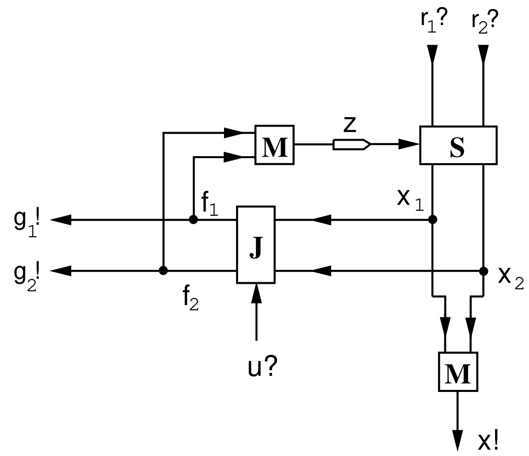


Figure 5.10: A decomposition of subSEQUENCER A_2.

request. Thus, we connect x_1, x_2, and u as inputs to the 2-by-1 JOIN, with u being the synchronization signal. Finally, after the subSEQUENCER has produced one of the grants, the next request may be selected, i.e., we have to provide the next synchronization signal for the SEQUENCER. We can do this by connecting g_1 and g_2 as inputs to the MERGE and connect the output of the MERGE to the IWIRE.

Formally, the decomposition above can be derived using the Separation Theorem. First, we decompose each of the three weavands from the specification of A_2:

$$|[\ x_1, f_1\ ::\ \mathbf{pref}\ *[r_1?; x_1; f_1, g_1!]\ \ ]|$$
$$\rightarrow \quad \{\ \text{syntax-directed decomposition, complete network behavior}\ \}$$
$$|[\ x_1, f_1\ ::\ \mathbf{pref}\ *[r_1?; x_1!],\ \mathbf{pref}\ *[x_1?; f_1!],\ \mathbf{pref}\ *[f_1?; g_1!]\ ]|$$

$$|[\ x_2, f_2\ ::\ \mathbf{pref}\ *[r_2?; x_2; f_2; g_2!]\ \ ]|$$
$$\rightarrow \quad \{\ \text{syntax-directed decomposition, complete network behavior}\ \}$$
$$|[\ x_2, f_2\ ::\ \mathbf{pref}\ *[r_2?; x_2!],\ \mathbf{pref}\ *[x_2?; f_2!],\ \mathbf{pref}\ *[f_2?; g_2!]\ ]|$$

$$
\begin{array}{ll}
& |[\, x_1, x_2, f_1, f_2 \;::\; \mathbf{pref} *[x_1; x!; u?; f_1 \mid x_2; x!; u?; f_2] \;]| \\
= & \quad \{ \text{ introduce internal symbol } z \ \} \\
& |[\, x_1, x_2, f_1, f_2, z \;::\; \mathbf{pref} *[\, z;\ (x_1; x!; u?; f_1 \mid x_2; x!; u?; f_2)\,] \;]| \\
\to & \quad \{ \text{ syntax-directed decomposition, complete network behavior } \} \\
& |[\, x_1, x_2, f_1, f_2, z \;:: \\
& \quad \mathbf{pref} *[\,(x_1? \mid x_2?);\ x!\,], \\
& \quad \mathbf{pref} *[\, u?;\ (f_1! \mid f_2!)\,] \parallel \mathbf{pref} *[\, x_1?; f_1! \mid x_2?; f_2!\,], \\
& \quad \mathbf{pref} *[z?;\ (x_1! \mid x_2!)\,], \\
& \quad \mathbf{pref} *[z!;\ (f_1? \mid f_2?)\,] \\
&]|
\end{array}
$$

Using the Separation Theorem we can obtain a decomposition by collecting all commands in the separate decompositions and weaving those with common outputs. After some simplifications, our decomposition is

Decomposition 5.4.3 (2-input subSEQUENCER)

$$
\begin{array}{lll}
A_2 \to |[\ x_1, x_2, f_1, f_2 \;:: & & \\
& \mathbf{pref} *[\, r_1?; x_1!\,] \parallel \mathbf{pref} *[\, r_2?; x_2!\,] \parallel \mathbf{pref} *[\, z?; (x_1! \mid x_2!)\,], & \{\text{SEQUENCER}\} \\
& \mathbf{pref} *[\,(x_1? \mid x_2?);\ x!], & \{\text{MERGE}\} \\
& \mathbf{pref} *[\, x_1?; f_1! \mid x_2?; f_2!] \parallel \mathbf{pref} *[\, u?;\ (f_1! \mid f_2!)\,], & \{\text{2-by- 1 JOIN}\} \\
& \mathbf{pref} *[\, z!;\ (f_1? \mid f_2?)\,], & \{\text{ ``initialized'' MERGE}\} \\
& \mathbf{pref} *[f_1?; g_1!], & \\
& \mathbf{pref} *[f_2?; g_2!] & \\
\quad]| & &
\end{array}
$$

The “initialized” MERGE can be further decomposed into a MERGE and a IWIRE (see Figure 5.10). Note that in Decomposition 5.4.3 some outputs are connected to multiple inputs (f_1 for example). This means these elements should be implemented as isochronic forks. All these inputs, however, are connected to DI-components, so we can safely replace the isochronic forks by genuine FORKs [Ebe89].

After we have decomposed both basic subSEQUENCERs, A_1 and A_2, we consider subSEQUENCER A_m with m requests, where $m > 2$. We can find a decomposition in a similar way we did for the 2-input subSEQUENCER: we split requests in two groups and do the arbitration in two steps. It turns out that subcomponents that do the first step of the arbitration are smaller subSEQUENCERs, and the component that arbitrates between the two groups of requests is a 2-input subSEQUENCER (see Figure 5.11).

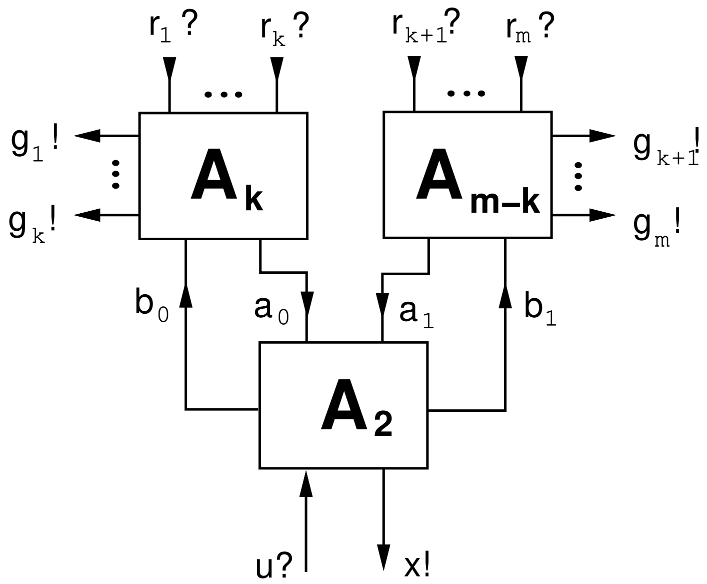


Figure 5.11: A decomposition of subSEQUENCER A_k.

Formally, the decomposition can be written as

Decomposition 5.4.4 (m-input subSEQUENCER) *Let $m > 2$ and $0 < k < m$. Then*

$$
\begin{array}{llll}
A_m \to (& |[& (i : 1 \le i \le k : x_i, f_i) :: & \{A_k\} \\
& & (\,\|\, i : 1 \le i \le k : \textbf{pref} *[r_i?; x_i; f_i, g_i!]\,) & \\
& & \|\ \textbf{pref}\,! *[(\,|\, i : 1 \le i \le k : x_i; a_0!; b_1?; f_k)\,] & \\
&]|, & & \\
& |[& (i : k < i \le m : x_i, f_i) :: & \{A_{m-k}\} \\
& & (\,\|\, i : k < i \le m : \textbf{pref} *[r_i?; x_i; f_i; g_i!]\,) & \\
& & \|\ \textbf{pref}\,! *[(\,|\, i : k < i \le m : x_i; a_1!; b_2?; f_i)\,] & \\
&]|, & & \\
& |[& l_1, l_2, h_1, h_2 :: & \{A_2\} \\
& & \quad \textbf{pref} *[a_0?; l_1; h_1; b_1!] \,\|\, \textbf{pref} *[a_1?; l_2; h_2; b_2!] & \\
& & \|\ \textbf{pref} *[l_1; x!; u?; h_1 \mid l_2; x!; u?; h_2\,] & \\
&]| & & \\
) & & &
\end{array}
$$

Since a derivation of this decomposition is very similar to the derivation of Decomposition 5.4.1, it is omitted.

Decompositions 5.4.1, 5.4.2, 5.4.3, and 5.4.4 allow us to decompose the n-input SEQUENCER into a binary tree with the SEQUENCER as a root and the subSEQUENCERs A_1 and A_2 as leaves and internal nodes. As an example, Figure 5.12 shows a decomposition of a 7-input SEQUENCER. Because such a decomposition consists of subsequencers A_2 connected as a binary tree, the number of basic components used is of the same order as the number of nodes in a tree. This implies that the area complexity of this decomposition is $\mathcal{O}(n)$.

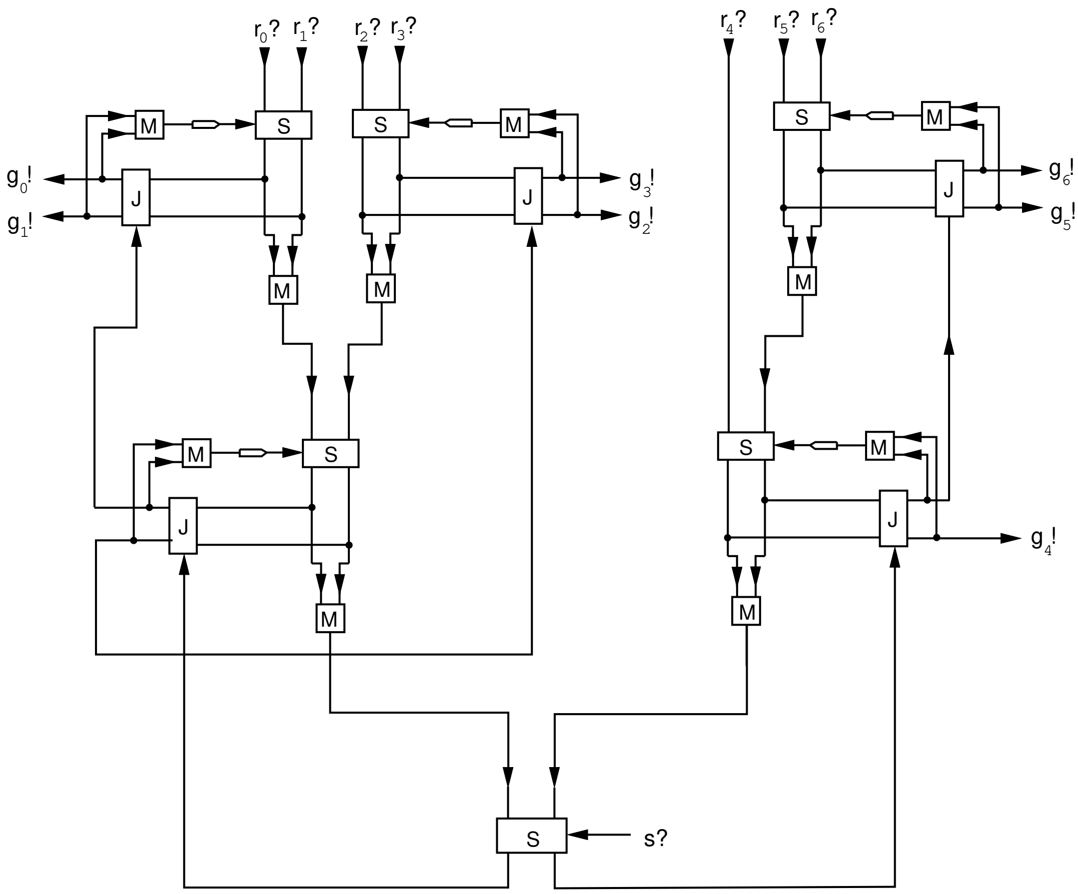


Figure 5.12: A decomposition of the 7-input SEQUENCER.

Chapter 6

Designing a general scheduler

In this chapter we discuss some decompositions of a general committee scheduler. We begin with a naive, straightforward decomposition, which yields a network where deadlock can occur. The next decomposition of the committee scheduler is based on a sequential finite automaton. We show that the number of states of the finite automaton may be exponential in the number of professors. Therefore, a general method of decomposing a finite automaton with area complexity linear in the number of states (and hence exponential in the number of professors) is not suitable for our needs. For this reason we exploit the structure of the committee problem and propose an efficient design of a finite automaton for the committee scheduler. The proposed scheduler has a fair arbitration scheme in terms of committee meetings and does not suffer from deadlock or livelock. Finally, we show that our decomposition rules are in conflict with fair behavior of any network implementation.

6.1 Naive decomposition

Recall that $P = \{p_0, \ldots, p_{n-1}\}$ denotes a set of professors and $C = \{c_0, \ldots, c_{m-1}\}$ denotes a set of committees. The set of committees in which professor p_i participates is called a committee set and is referred to as C_i. Similarly, P_i denotes the roster of committee c_i; sometimes we refer to P_i as the *professor set* of committee c_i. We assume that each professor is a member of at least one committee and that each committee has at least one member.

A decomposition of the scheduler, specified as

$$
\begin{array}{ll}
& \mathsf{scheduler} \\
= & \quad \{ \text{ specification } \} \\
& \quad \textbf{pref} *[r_0?;\ (\,|j :\ c_j \in C_0 :\ m_j!\,)] \\
& \| \ \textbf{pref} *[r_1?;\ (\,|j :\ c_j \in C_1 :\ m_j!\,)] \\
& \vdots \\
& \| \ \textbf{pref} *[r_{n-1}?;\ (\,|j :\ c_j \in C_{n-1} :\ m_j!\,)]
\end{array}
$$

can be derived as follows. First, we introduce some internal symbols:

$$
\begin{array}{ll}
= & \quad \{ \text{ introduce internal symbols } \} \\
& |[\,(\,i, j :\ 0 \le i < n \ \wedge\ c_j \in C_i :\ x_{i,j}\,)\ \ :: \\
& \qquad \textbf{pref} *[r_0?;\ (\,|j :\ c_j \in C_0 :\ x_{0,j};\ m_j!\,)] \\
& \quad \| \ \textbf{pref} *[r_1?;\ (\,|j :\ c_j \in C_1 :\ x_{1,j};\ m_j!\,)] \\
& \quad \vdots \\
& \quad \| \ \textbf{pref} *[r_{n-1}?;\ (\,|j :\ c_j \in C_{n-1} :\ x_{n-1,j};\ m_j!\,)] \\
&]|
\end{array}
$$

The rationale behind the internal symbols is a separation of the problem into two phases: the receipt of a professor's ready message and the output of a permission to begin a committee meeting. Since the specification is a weave of commands, we aim to apply the Separation Theorem. A weavand can be decomposed as

$$
\begin{array}{ll}
& \textbf{pref} *[r_i?;\ (\,|j : c_j \in C_i : x_{i,j};\ m_j!\,)] \\
\rightarrow & \quad \{ \text{ syntax-directed decomposition } \} \\
& (\quad \textbf{pref} *[r_i?;\ (\,|j : c_j \in C_i : x_{i,j}!;\,)],\ (\,j : c_j \in C_i : \textbf{pref} *[x_{i,j}?; m_j!]\,)\quad).
\end{array}
$$

When we collect commands in separate decompositions and weave those that have common outputs, we get the following decomposition:

Decomposition 6.1.1 (General scheduler; naive)

$$
\begin{array}{lll}
& \mathsf{scheduler} & \\
\rightarrow & \quad \{ \text{ decomposition by the Separation Theorem } \} & \\
& (& \\
& \ (\,i : 0 \le i < n : \textbf{pref} *[r_i?;\ (\,|j : c_j \in C_i : x_{i,j}!\,)]\,), & \{\textsc{selectors}\} \\
& \ (\,j : 0 \le j < m : (\,\|\,i : p_i \in P_j : \textbf{pref} *[x_{i,j}?;\ m_j!]\,)\,) & \{\textsc{joins}\} \\
&) &
\end{array}
$$

We can see that we have one k-input JOIN for every committee of size k and one SELECTOR for every professor. The SELECTOR, shown in Figure 6.1, is a component we briefly mentioned in Section 4.2. It has one input and a number of outputs; after its environment has produced the input, the SELECTOR produces an output on a nondeterministically selected output terminal. After the SELECTOR has produced an output, the environment is allowed to produce the next input. Such a behavior may be specified as

$$\textsc{selector} \quad = \quad \textbf{pref} *[r?;\ (g_0!|g_1!)\,]$$

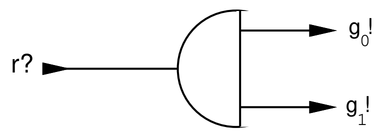


Figure 6.1: SELECTOR.

In this chapter we use the SELECTOR as a basic component. It is easy to see that the n-output SELECTOR can be decomposed into a network consisting of 2-output SELECTORs connected in the form of a binary tree.

Informally, the role of a SELECTOR in the decomposition above is to nondeterministically select the committee in which a corresponding professor is going to meet. The role of a JOIN is to ensure that all members of a committee are ready to meet when a meeting begins. Since we cannot guarantee that the SELECTORs always makes a right choice, there is a possibility of deadlock.

For example, consider professors p_0, p_1, and p_2 and committees c_0, c_1, and c_2, such that

$$c_0 = \{p_0, p_1\} \quad c_1 = \{p_1, p_2\} \quad c_2 = \{p_0, p_2\}$$

A decomposition of the corresponding committee scheduler is shown in Figure 6.2. One of the traces that leads to deadlock is $r_0\, r_1\, r_2\, x_0\, x_2\, x_5$. We can see that one of the inputs has arrived at each JOIN. From this point on, however, nothing can happen. Intuitively, since the SELECTORs make their decisions independently of each other, the scheduler does not take into account which ready messages have already arrived when it decides which meeting a professor will be waiting for. Note that here a professor actually waits for a meeting of one committee only. This naive decomposition can be seen as an extreme, because all ready messages may be processed entirely in parallel.

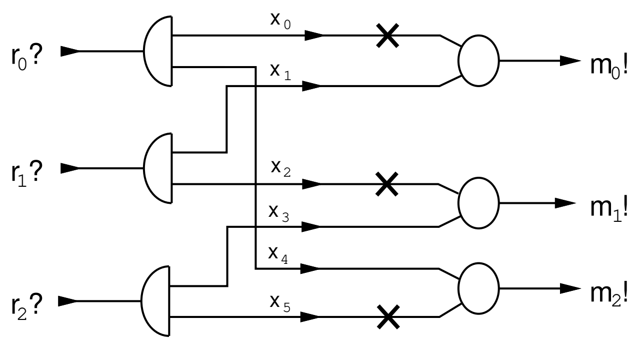


Figure 6.2: Deadlock in a naive decomposition.

6.2 Decomposition using a finite automaton

From our first attempt to decompose the general committee scheduler we learned that we cannot decide in which committee a professor is going to meet without taking into account ready messages from other professors. In this section we show how we can design the committee scheduler as a finite automaton and how we can obtain its decomposition. The solution is based on a nondeterministic, sequential Mealy automaton [HU79].

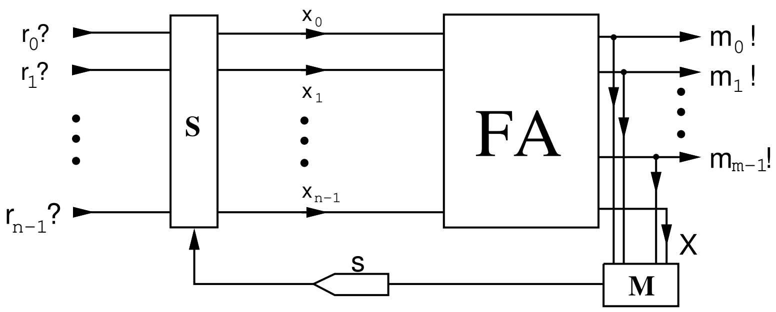


Figure 6.3: Outline of a finite-automaton-based decomposition.

The first problem we have to address is that a sequential finite automaton (FA) can process only one input symbol at a time, but the committee scheduler has to be able to accept ready messages in parallel. A solution is to use an n-input SEQUENCER as an interface between ready messages coming from professors and the FA that processes the messages. Initially, the synchronization signal of the SEQUENCER has to be present, because upon receipt of a ready message from a professor, the SEQUENCER has to be able pass the ready message to the FA. When the FA processes one ready message, the next synchronization input for the SEQUENCER has to be produced. We can do this by connecting all the outputs of the FA to a MERGE, and then connecting the output of the MERGE to the IWIRE that produces the synchronization signals for the SEQUENCER. An outline of this design is shown in Figure 6.3. Note that the FA produces a special output X when it does not allow a meeting of a committee; otherwise, the next synchronization signal for the SEQUENCER would not have been produced. This design can be seen as another extreme, opposite to the naive decomposition; here, all ready messages are processed in a sequential manner.

Obtaining a finite automaton

How can we obtain an FA for the committee scheduler ? Figure 6.4 shows a finite automaton for the scheduler with two committees:

$$c_0 = \{p_0\} \quad c_1 = \{p_0, p_1\}$$

The automaton is nondeterministic: When input r_0 is received in state q_1, the automaton may either allow a meeting of committee c_0, or it may allow a meeting of committee c_1. Note that a special output X is produced when none of the two committees can meet.

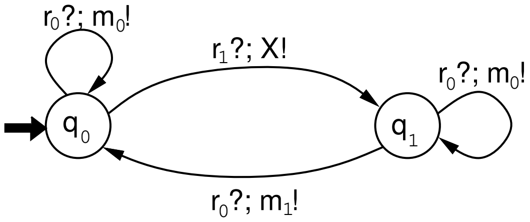


Figure 6.4: Example: an automaton for a simple scheduler.

In general, each state of an automaton corresponds to a set of professors that have sent their ready messages and are waiting for a meeting. In the example above, state q_0 corresponds to the situation where no professor is waiting; state q_1 corresponds to the situation where professor p_1 is waiting for a meeting.

Let us estimate the number of states for an FA corresponding to the committee scheduler for n professors ($P = \{p_0, \ldots, p_{n-2}, x\}$) and m committees ($C = \{c_0, \ldots, c_{m-1}\}$). Because a state represents an element of the power set of P, we know that the number of states cannot be larger than 2^n. The following example shows that the number of states can be of $\mathcal{O}(2^n)$ even when the number of committees is linear in the number of professors.

We consider the following set of committees

$$
\begin{aligned}
c_0 &= \{p_0, x\} \\
c_1 &= \{p_1, x\} \\
&\vdots \\
c_{n-2} &= \{p_{n-2}, x\}
\end{aligned}
$$

The automaton for the scheduler for the committees above has to keep track of every possible subset of professors $p_0, \ldots, p_{n-2}$ because, upon receipt of a ready message from professor x, it may allow a meeting of any committee c_i, where $0 \leq i \leq n-2$, if professor p_i has already submitted a ready message. Therefore, the number of states in the automaton is of $\mathcal{O}(2^n)$.

6.2.1 Decomposition of a finite automaton

In order to obtain a decomposition of the committee scheduler as shown in Figure 6.3, we have to decompose the SEQUENCER and the finite automaton. In Section 5.4 we gave a decomposition of the n-input SEQUENCER as a special case of the committee scheduler, so the next question we address is how to decompose an FA. We define a nondeterministic Mealy automaton as a six-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0)$ where Q is a set of states, q_0 is the initial state, Σ is an input alphabet, Γ is an output alphabet, and δ is a transition and output function that maps from $Q \times \Sigma$ to $2^{Q \times \Gamma}$. Note that we modified the usual definition of the Mealy automaton [HU79] slightly by introducing nondeterminism.

Generalized JOIN

The basis of our first decomposition of an FA is the n-by-m JOIN, which is a generalization of the 2-by-1 JOIN. Let us first introduce the 2-by-2 JOIN, given by

$$\begin{array}{lll} \text{2-by-2 JOIN} = & \textbf{pref} *[& (r_0?\|q_0?); g_{0,0}! \\ & & \mid (r_0?\|q_1?); g_{0,1}! \\ & & \mid (r_1?\|q_0?); g_{1,0}! \\ & & \mid (r_1?\|q_1?); g_{1,1}! \\ &] & \end{array}$$

The schematic of the component is given in Figure 6.5. Intuitively, the 2-by-2 JOIN is a generalization of the 2-by-1 JOIN in a way that the synchronization is performed between one "row" input, namely r_0 or r_1, and one "column" input, namely q_0 or q_1 (recall that the 2-by-1 JOIN has only one column input). The environment is restricted to produce only one of the inputs r_0 and r_1 and only one of the inputs q_0 and q_1. When the 2-by-2 JOIN produces an output a new cycle is enabled. We treat the 2-by-2 JOIN as a basic component, since it is a conjecture that it cannot be decomposed into a network consisting only of components given in Figure 2.2 [Ebe93].

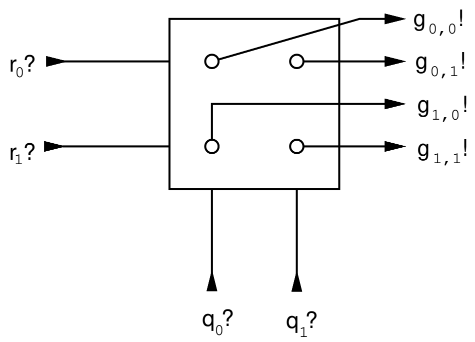


Figure 6.5: 2-by-2 JOIN.

The 2-by-2 JOIN can be further generalized so that two synchronized signals are members of two arbitrarily large sets of inputs. The schematic of such a component, usually referred to as the n-by-m JOIN, is shown in Figure 6.6.

A specification of the n-by-m JOIN is

$$n\text{-by-}m \text{ JOIN} = \quad \textbf{pref} *[(\mid i,j :\ 0 \leq i \leq n \wedge 0 \leq j \leq m :\ (\,r_i? \parallel q_j?\,); g_{i,j}!)]$$

Intuitively, the n-by-m JOIN behaves as follows: Its environment is allowed to produce one of the "row" inputs, say r_i, and one of the "column" inputs, say q_j. After the n-by-m JOIN has received the two inputs, it may produce the output that corresponds to the intersection of the i^{th} row and the j^{th} column, namely $g_{i,j}$.

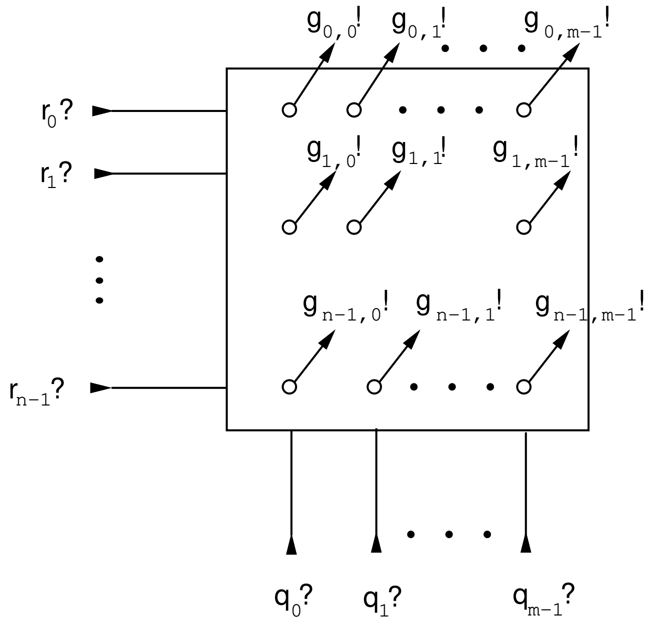


Figure 6.6: n-by-m JOIN.

At this point a new cycle may begin. We mention without proof that the n-by-m JOIN can be decomposed into a network consisting of MERGEs and 2-by-2 JOINs (or 2-by-1 JOINs if $m = 1$ or $n = 1$) in such a way that the area complexity of the decomposition is $\mathcal{O}(mn)$ and the time complexity is $\mathcal{O}(\min\{m \log n, n \log m\})$ [Ebe93].

Decomposing an arbitrary FA

We can use an n-by-m JOIN as a basis for a decomposition of an arbitrary FA. For example, Figure 6.7 shows a decomposition of a nondeterministic Mealy automaton

N shown in Figure 6.4. Formally, the automaton is defined as

$$Q = \{q_0, q_1\} \quad \Sigma = \{r_0, r_1\} \quad \Gamma = \{m_0, m_1, X\}$$
$$\delta(q_0, r_0) = \{(q_0, m_0)\} \qquad\qquad \delta(q_0, r_1) = \{(q_1, X)\}$$
$$\delta(q_1, r_0) = \{(q_0, m_1), (q_1, m_0)\}$$

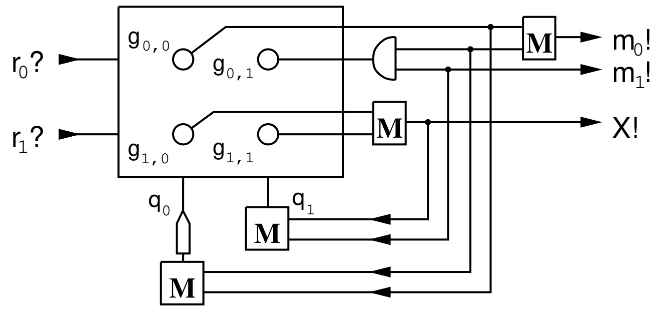


Figure 6.7: Example: a decomposition of a finite automaton.

How did we obtain the decomposition in Figure 6.7 ? We used the "row" inputs of the 2-by-2 JOIN to accept the inputs to the automaton, and the "column" inputs to indicate the current state of the automaton. To ensure that the automaton begins the computation in the initial state we connected the IWIRE to the input corresponding to state q_0. Output $g_{i,j}$ corresponds to transition $\delta(q_i, r_j)$. Each transition is associated with a change of the state and with an output. Therefore we connect $g_{i,j}$ to a MERGE with an output that corresponds to the new state and to a MERGE with an output that corresponds to the output associated with the transition. When we have a nondeterministic transition, we first connect $g_{i,j}$ to a SELECTOR which chooses a transition. Note that automaton N is not complete, because we did not define transition $\delta(q_1, r_1)$. Since the environment (according to our specification) is not allowed to produce input r_1 when N is in state q_1 and since, according to our interpretation of the specification, we do not guarantee anything if the environment does not behave as specified, a decomposition does not have to distinguish between transitions $\delta(q_1, r_0)$ and $\delta(q_1, r_1)$.

We can take the same approach and obtain a decomposition of an arbitrary Mealy automaton by using an n-by-m JOIN, where n is the size of the input alphabet and m is the number of states. The complexity of such a decomposition is dominated by the decomposition of the n-by-m JOIN; the area complexity is $\mathcal{O}(mn)$ and the time complexity is $\mathcal{O}(\min\{m \log n, n \log m\})$.

In Section 6.2.1 we showed that, when the number of committees is $\mathcal{O}(n)$, the number of states of an FA for the committee scheduler is $\mathcal{O}(2^n)$. This means that the approach described above would give us a decomposition with area complexity $\mathcal{O}(n2^n)$. Because we cannot be satisfied with such an inefficient decomposition, we depart from decomposing a general finite automaton. In our next attempt we exploit the nature of the committee problem in order to get a more efficient decomposition.

6.2.2 A simple arbitration scheme

The committee scheduler (and thus a corresponding FA) has two basic functions: (1) to keep track of received ready messages and (2) to allow a meeting of a committee when all its members are waiting. Thus, we divide an automaton into two modules: a memory module and an arbitration module. We first concentrate on the memory module.

The data structure we use to build a memory module is a Boolean $n \times m$ matrix B with each row corresponding to a professor and each column corresponding to a committee. Entry $b_{i,j}$ is present if professor p_i is a member of committee c_j. Figure 6.8 shows a memory matrix for professors p_0, p_1, and p_2 and committees c_0, c_1, and

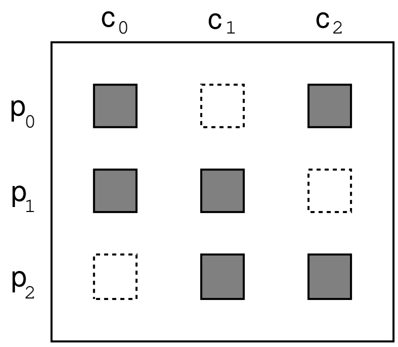


Figure 6.8: A memory matrix.

c_2, such that

$$c_0 = \{p_0, p_1\} \quad c_1 = \{p_1, p_2\} \quad c_2 = \{p_0, p_2\}$$

The dotted squares represent the matrix entries that are not present.

Initially, all entries in the matrix have the value **false**. When the automaton accepts a ready message from professor p_i, it sets all the entries in row i to **true**. When the scheduler allows a meeting of committee j, it sets to **false** all the entries in the rows that correspond to members of committee j. The fact that a professor is waiting to attend a meeting is recorded in every column that corresponds to a committee in which the professor is a member. This means that our storage scheme is redundant; the redundancy, however, pays off with a simpler decomposition.

Because each entry of the matrix stores one bit, a decomposition of a matrix could be based on a grid of one-bit-memory cells. A cell (see the schematic in Figure 6.9) has seven terminals: Input r corresponds to a read request to which a cell answers with output r_0, if the content of the cell is **false**, or with r_1 otherwise; inputs w_0 and w_1 correspond to requests to write either **false** or **true** into the cell. After the contents of the cell have been modified, the cell produces an acknowledgement:

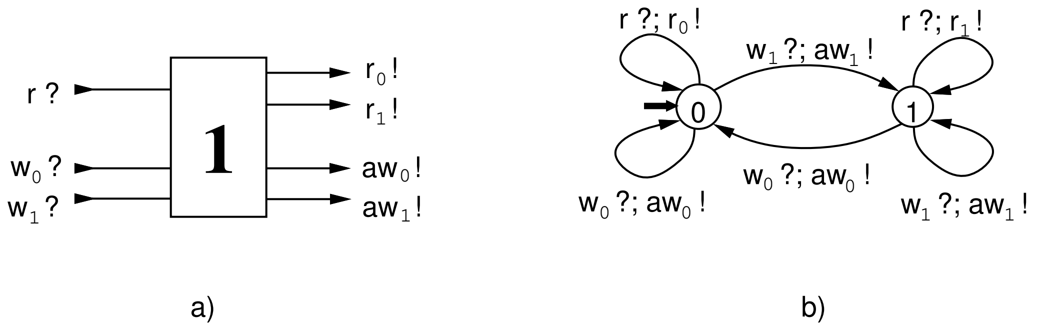


Figure 6.9: A one-bit memory cell: a) schematic b) state graph.

aw_0 for request w_0 and aw_1 for w_1. We stipulate that every cell is initialized to **false**.

A one-bit-memory cell can be seen as a simple Mealy automaton with two states, each corresponding to one of two possible values stored in a cell (Figure 6.9). Since the automaton is small, we can decompose it using a 3-by-2 JOIN (3 input symbols, 2 states) as we learned in Section 6.2.1.

After receiving a ready message, an FA updates its memory module (see the schematics in Figure 6.10) and then queries whether any of the committees can meet. Since ready messages trigger updates in the memory module, the module has one input terminal for receiving a ready message for each of the professors. When the memory module has been updated, it activates the arbitration module by producing output *arb*. The arbitration module initiates a check whether committee c_i can meet by sending output mc_i to the memory module. If all the entries in a column corresponding to c_i are **true**, the memory module sets all the entries corresponding to c_i to **false** and produces okc_i as input to the arbitration module; if c_i cannot meet, the memory module produces output $notc_i$. If the meeting of c_i has been confirmed, the arbitration module outputs m_i (sent to professors) and

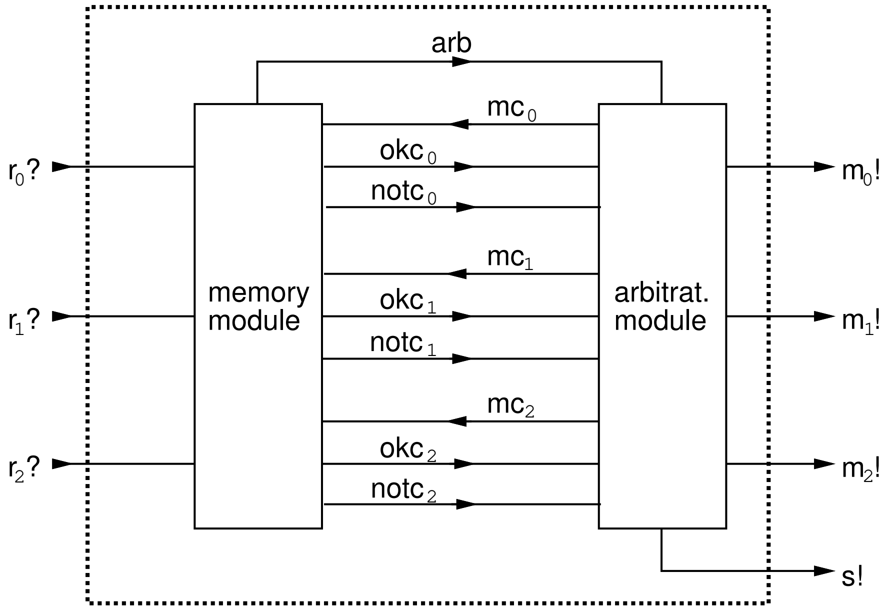


Figure 6.10: A memory and an arbitration module form a scheduler.

s (the next synchronization signal for the SEQUENCER); otherwise the arbitration module proceeds with a query for a meeting of the next committee.

In Figure 6.11 we give a decomposition of the memory module for our example. We can see that, after the memory module has received a ready message, all entries in the corresponding row are concurrently updated. After all the updates have been completed, acknowledgement signals (aw_1) from the memory cells are synchronized using a JOIN and then output arb is produced, i.e., the arbitration module is activated.

What happens when a query whether a committee can meet is received? For example, suppose the memory module receives mc_0. Then a read request for the first entry in the first column (the one that corresponds to committee c_0) is produced. If professor p_0 has submitted a ready message (i.e., the entry is **true**), a read request for the next cell is produced. Otherwise, committee c_0 cannot meet; thus the

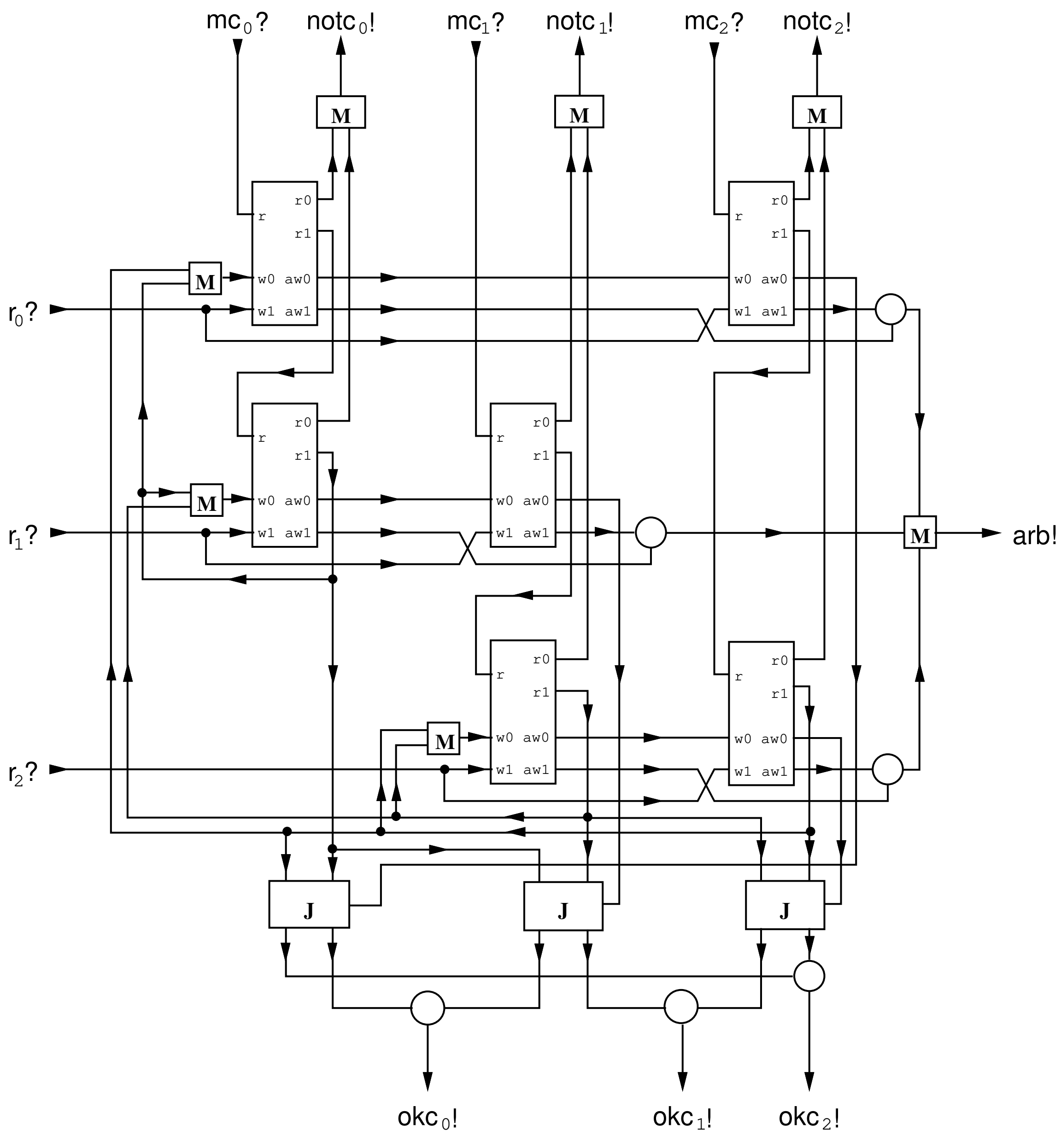


Figure 6.11: A decomposition of the memory module.

memory module produces output $notc_0$. If all the entries in the column are **true**, committee c_0 may meet. Therefore we have to set all the entries in the first and the second row of the matrix (the ones that correspond to members of c_0) to **false**. To accomplish this, we generate inputs w_0 to the first cells in both rows and then use acknowledgements aw_0 to generate inputs w_0 for subsequent cells. When all the updates have been completed, the acknowledgements are synchronized using k-by-1 JOINs, where $k \leq m$, and output okc_0 is produced. Note that all entries could be updated concurrently. However, in order to keep the example simple, we give a decomposition in which the entries in each row are updated sequentially. Concurrency could be achieved as in the updating the entries upon receipt of a ready message.

The area complexity of the memory module is $\mathcal{O}(mn)$: We have at most mn memory cells, each of area complexity $\mathcal{O}(1)$, n k-by-1 JOINs with area complexity $\mathcal{O}(m)$ and, finally, m JOINs with the area complexity $\mathcal{O}(n)$. To determine the response time, we observe that there are two basic activities of the memory module, an update when a new ready message arrives and a query whether a committee can meet. A query is initiated by the arbitration module. For the update we have to generate write-**true** signals for all the cells in the appropriate row, and, after having the memory cells updated, synchronize the acknowledgements before output arb is produced. Because the update is done in parallel, its response time is dominated by JOINs which are used to synchronize acknowledgements from at most m cells; this gives the response time of $\mathcal{O}(\log m)$. The second activity, a query, consists of two phases. In the first phase we determine whether all memory cells in a column are set to **true**. Since there are at most n cells to test, the time complexity of this phase is $\mathcal{O}(n)$ (note that, if a committee cannot meet, a response is directed through a MERGE with the response time $\mathcal{O}(\log n)$). If a committee can meet, the second phase takes place. First, a write-**false** signal has to be distributed among all the cells that correspond to the members of a committee; this can be done in $\mathcal{O}(\log mn)$ time. Then it takes $\mathcal{O}(\log m)$ time to collect acknowledgements for each row (in parallel), and, finally, $\mathcal{O}(\log mn)$ to synchronize the acknowledgements and produce the output. Thus, the response time of the memory module is $\mathcal{O}(n + \log m)$.

The next step is designing the arbitration module. A simple way of determining which committee will meet is to keep a list of committees. Each time, when a ready message is received and the memory module is updated, the arbitration module traverses the list and queries the memory module whether a committee that corresponds to the current position in the list can meet. If a committee can meet, the arbitration module produces an output that allows a meeting, and then waits for the next *arb* message; otherwise, it moves to the next item in the list. If none of the committees can meet, the module waits for the next *arb* message and produces the next synchronization signal for the SEQUENCER.

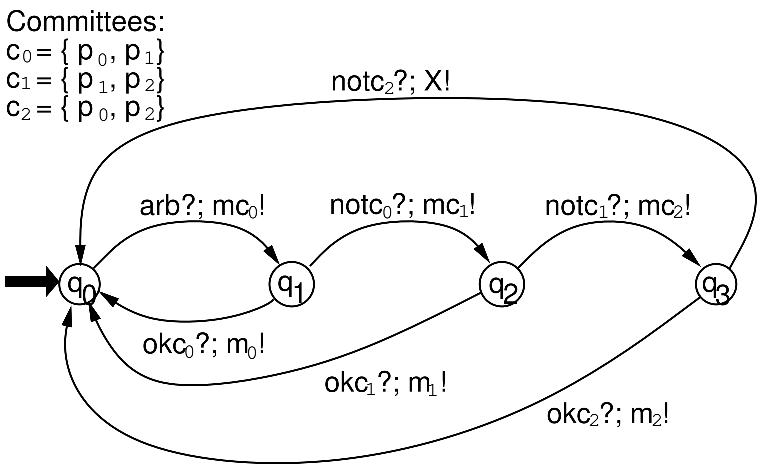


Figure 6.12: An FA for a simple arbitration module.

Such a strategy is easy to specify by a finite automaton; an FA for our example is shown in Figure 6.12. Since the automaton is very simple — from each state we have transitions only to the next and to the initial state — we do not have to follow the decomposition scheme from Section 6.2.1. A more efficient decomposition is shown in Figure 6.13.

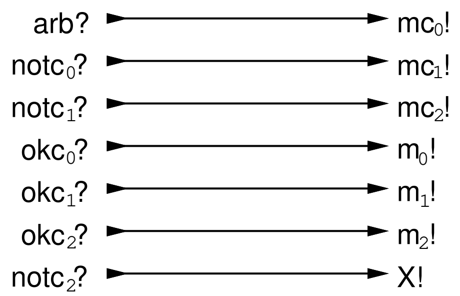


Figure 6.13: A decomposition of a simple arbitration module.

The area complexity of a network consisting of the memory and the simple arbitration module is $\mathcal{O}(mn)$. The response time is $\mathcal{O}(mn + m \log m)$, because there are at most m queries and every query has the response time $\mathcal{O}(n + \log m)$.

6.2.3 Starvation

The committee scheduler as designed above satisfies all our conditions for a decomposition, but it can still exhibit an undesired behavior. For example, consider committees c_0 and c_1 such that

$$c_0 = \{p_0, p_1\} \quad c_1 = \{p_1, p_2\}$$

and suppose that, in a sequence of queries, committee c_0 appears before committee c_1. Note that the sequence of queries does not change; thus committee c_0 is *always* checked before committee c_1. This means that if the FA first receives ready messages from professors p_0 and p_2, and then a ready message from professor p_1, the FA allows the meeting of committee c_0. Now suppose that all subsequent ready messages from professor p_1 arrive at the SEQUENCER after the ready messages from professor p_0, so that the SEQUENCER always advances a ready message from p_0 before a ready message from p_1. When the FA receives a ready message from professor p_1, both committees are ready to meet, but, because of the fixed sequence of queries, committee c_0 is always chosen. Thus, committee c_1 might never meet, regardless of the fact that it could meet every time a ready message from professor p_1 was received. Such a situation is referred to as *starvation*. The reason for starvation is a fixed sequence of queries. It does not matter in which order the committees are placed in the list; if two of them have a common member, starvation can occur.

6.2.4 Subcommittee problem

The example below illustrates another problem with the arbitration module as designed above. Let us try to design an FA-based scheduler for the following committee configuration

$$c_0 = \{p_0\} \quad c_1 = \{p_1\} \quad c_2 = \{p_0, p_1\}.$$

First, we have to design a finite automaton. Because each state in the automaton corresponds to a set of waiting professors, our automaton needs three states, corresponding to $\emptyset$ (state $q_\emptyset$), $\{p_0\}$ (state q_0), and $\{p_1\}$ (state q_1), with $q_\emptyset$ being the initial state. When the automaton is in state $q_\emptyset$ and receives a ready message from professor p_0, it may immediately initiate a meeting of committee c_0. However, if the scheduler allows a meeting of committee c_0 always when it receives a ready message from professor p_0 and if it works similarly when it receives a ready message from professor p_1, then committee c_2 can never meet. The first idea for a solution might be as follows. Upon receipt of a ready message from p_0, the automaton nondeterministically decides whether it will initiate a meeting of committee c_0, or it will rather wait for a ready message from p_1 and then initiate a meeting of committee c_2. A schematic of such an automaton is shown in Figure 6.14. This automaton is

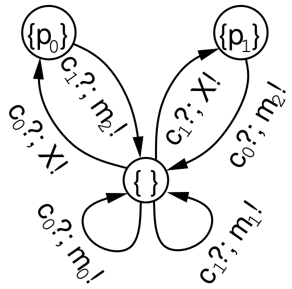


Figure 6.14: An automaton with the subcommittee problem.

an acceptable solution if it is guaranteed that professor p_1 will eventually submit a ready message. However, if professor p_1 never becomes available for a meeting, the automaton never receives an input in state q_0. This means that committee c_0 may never meet, in spite of the fact that all the members of the committee are waiting for a meeting; this also means that professor p_0 may never attend a meeting.

The problem with the above committee configuration is that either there exists a committee that can never meet, or there is a possibility that the scheduler waits for a ready message that may never arrive, while there is a committee which could meet. This problem occurs with a committee configuration in which some subsets of a committee are committees themselves; therefore, we call it the *subcommittee problem*.

Definition 6.2.1 (Subcommittee problem) *The subcommittee problem occurs if and only if the following condition is satisfied*

$$\exists A : A \in C : (\forall S : S \subseteq A \wedge |S| = |A| - 1 : (\exists K : K \in C \wedge K \subseteq S)) \quad (6.1)$$

Informally, Condition 6.1 states that if $|A| = k$, then after every possible sequence of $k-1$ ready messages there is some committee K with all members waiting for a meeting. This means that the scheduler has to decide whether it will wait for the missing ready message (which may never arrive) and then initiate a meeting of committee A, or it will immediately allow a meeting of K.

A way to avoid the subcommittee problem is to modify the committee configuration so that Condition 6.1 is not satisfied. For example, we can augment every committee K which is a subset of a "larger" committee A by a fictive committee member. After the modification of the committee configuration from the example above we would get committees

$$c_0 = \{p_0, f_0\} \quad c_1 = \{p_1, f_1\} \quad c_2 = \{p_0, p_1\}$$

where f_0 and f_1 are fictive committee members. Obviously, after such a modification, Condition 6.1 cannot be satisfied because no subset of a "large" committee is a committee itself.

How does a fictive committee member f behave ? Initially f sends a ready message after an arbitrary delay. After a committee of which f is a member meets, f waits for a certain time and then submits the next ready message. An implementor of the scheduler has the freedom to choose how long a fictive committee members waits before it submits the next ready message. If the delay is small then the scheduler favors "small" committees; if the delay is large then the possibility that a meeting of a large committee is initiated increases. In the following decompositions we do not pay any attention to fictive committee members, because their introduction in a decomposition is trivial.

Note that the original statement of the problem (given in Section 1.1) allows starvation and even allows solutions where some committees cannot meet at all.

Consequently, starvation and the subcommittee problem may occur in solutions to the committee problem proposed in [Bag87], [CM88], and [Bag89]. We remark that the only original requirement for the committee scheduler is that, if all members of a committee are waiting, one of them should attend *some* meeting. In the next section we strengthen this requirement and show a simple arbitration scheme that prevents starvation.

6.2.5 An arbitration scheme using a queue

The major problem with our previous arbitration scheme is the fixed order in which the arbitration module queries whether committees can meet. If committees c_i and c_j have common members and committee c_i meets, we want committee c_j to appear before c_i in a sequence of queries. Moreover, we want c_j to appear in the sequence before c_i until c_j meets, and then the order should be changed again to prevent starvation of c_i. A way of realizing this idea is to maintain a list of committees as we did in our previous solution, but, when a committee is allowed to meet, we move this committee to the end of the list. As shown in Figure 6.15, we first store the content of the current cell in a buffer, then shift the "tail" of the list one place to the left, and finally move the content of the buffer into the last cell. Because a committee, when it meets, moves to the end of the list (i.e., to the end of the sequence of queries), such a list in fact implements a queue.

What happens if all the members of committees c_i and c_j are ready to meet and c_i and c_j have a common member ? In such a situation we say that committees c_i and c_j are in a conflict. Suppose that the position of committee c_i in the queue is $pos(c_i)$ and that of c_j is $pos(c_j)$. Furthermore, suppose that $pos(c_i) < pos(c_j)$. This means that, in the current sequence of queries, committee c_i appears before

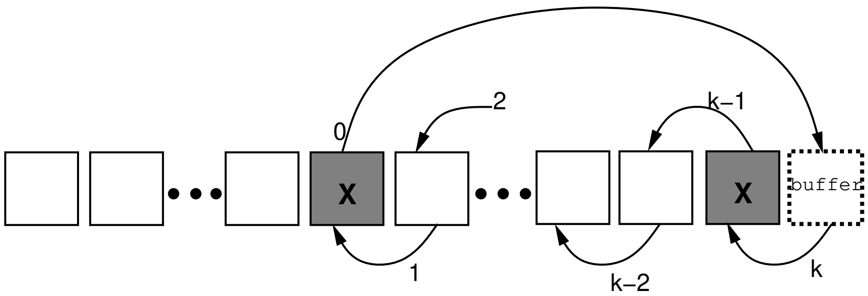


Figure 6.15: Moving a committee to the end of the queue.

committee c_j; thus, the arbitration module will allow committee c_i to meet and will move it to the end of the queue. The new position of c_j in the queue will be $pos(c_j) - 1$. Because the length of the queue is n, we can see that committee c_j can lose the arbitration in at most $n - 1$ conflicts. In other words, it cannot happen that a committee is ready, but not allowed to meet for an unbounded number of times. Therefore, the arbitration scheme is fair.

Since our decomposition of the arbitration module that is based on the idea above is rather complex, it is omitted here. A reader can find the decomposition and its analysis in Appendix A. We remark that the area complexity of the queue-based arbitration module is $\mathcal{O}(m^3)$, which means that the area complexity of a complete FA is $\mathcal{O}(mn + m^3)$. The response time of the FA is then $\mathcal{O}(m(m + n))$.

6.2.6 Discussion

Now we have a committee scheduler with a fair arbitration scheme and without deadlock and livelock. However, fairness of the scheduler does not depend solely on its arbitration scheme; remember that the scheduler consists of a SEQUENCER and of a finite automaton. The SEQUENCER serves as a “filter” for ready messages comming from professors; thus, if it behaves unfairly, it is possible that a ready message from a professor will never reach the finite automaton. From the above we can conclude that the behavior in which a committee always loses in a "conflict" (now at the SEQUENCER) is possible in our decomposition; therefore, the starvation is still possible. We remark, however, that such a behavior of the SEQUENCER is unlikely to occur in practice.

We can make another remark on fairness of a decomposition. Consider an arbitrary network that implements a fair committee scheduler. Since fairness means that none of the committees can "lose" the arbitration for an unbounded number of times, a trace in which, for example, only one committee always wins the arbitration cannot occur in the scheduler's behavior. From this we can conclude that no fair scheduler qualifies as a decomposition according to our conditions; the conditions are obviously too strong. On the other hand, as we saw in the case of our naive decomposition (see Section 6.1), the four conditions are too weak to detect deadlock. This dilemma may be the price we have to pay for having such a simple semantic domain as prefix-closed, non-empty trace structures.

From Section 6.2 onwards our description of components and the design of "decompositions" have been rather informal. The reasons for this informality are as follows. First, the components we were trying to decompose became quite complex. For example, the only specifications we found for a memory module and for an arbitration module were finite automata. Unfortunately, the length of such specifications is linear in the number of states, which may be exponential in the number of professors.

Second, the derivation of a decomposition can be compared to the derivation of a program: Such derivations may be feasible for small cases, but become hard when a problem becomes "large". On the basis of the comparison with program design we believe that a number of general decomposition paradigms should be developed. Such paradigms would not only make obtaining a decomposition easier but would also have a positive impact on proving the correctness of a decomposition. A good basis for such paradigms could be the existing knowledge about concurrent programming. Furthermore, we believe that it would be useful to have a number of hints on the proposed use of a component. For example, we often used n-by-m JOINs as switches to direct a signal in the proper direction.

Chapter 7

Conclusions

The committee problem, also known as n-party rendezvous, captures the essence of synchronization in many concurrent systems. It combines basic forms of synchronization, namely, mutual exclusion and condition synchronization. In this thesis we explored how the committee problem can be solved in the context of delay-insensitive circuits.

In Chapter 2 we described trace theory and a CSP-like language, called **commands**, that we used for specifying the communication behavior of a component. We also introduced a number of primitive delay-insensitive components that were the ultimate target in later decompositions.

In Chapter 3 we showed that some canonic synchronization problems are in fact special cases of the committee problem. We gave specifications for the committee schedulers for mutual exclusion, phase synchronization, and the dining philosophers problem. We also showed that some known delay-insensitive components, namely the ATS, the JOIN, and the SEQUENCER, are schedulers for special cases of the committee problem. Finally, we specified a general committee scheduler in terms of the

commands and in terms of a Petri net. Finding a specification in terms of **commands** was made very simple by using a constraint-based specification technique.

In Chapter 4 we stated the requirements that have to be satisfied if a network is to be a decomposition of a given specification. We also stated two theorems, the Substitution Theorem and the Separation Theorem that are helpful in finding and verifying a decomposition.

In Chapter 5 we proposed decompositions for the committee schedulers for the special cases mentioned above. The common denominator of all decompositions is that they are intuitively easy to understand and obtain, applying the knowledge and heuristics we knew from concurrent programming. All decompositions also have simple derivations. Since the committee scheduler can be specified as a weave (a parallel composition with synchronization on common symbols) of trace structures, we were able to exploit the power of the Separation Theorem [Ebe89] in order to derive each of the decompositions.

We gave three decompositions of the n-input SEQUENCER. All three satisfy all our correctness conditions. However, a token-bus decomposition suffers from unnecessary blocking, and a token-ring decomposition suffers from livelock. This again shows that not all progress conditions are included in our correctness criteria. The third decomposition is based on a binary tree structure and exhibits neither deadlock nor livelock.

In Chapter 6 we discussed decompositions of a general committee scheduler. We began with a naive decomposition, where deadlock can occur. Since the committee scheduler can be seen as a finite automaton, we discussed decompositions of a nondeterministic Mealy automaton. The general approach we described uses an n-by-m JOIN, where n is the size of the input alphabet and m is the number of states of the automaton. We showed that the number of states for the committee scheduler can be exponential in the number of professors, even when the number of committees is linear in the number of professors. This means that the number of components used to decompose a finite automaton (the area complexity) for the committee scheduler could be exponential in the number of professors. Therefore, we took another approach and designed a finite automaton that makes use of the special structure of the committee scheduler. First we divided the automaton into two parts: a memory part, which stores current states of professors and committees, and the arbitration part, that decides which committee will meet. We designed the memory part as a matrix of one-bit memory cells. The area complexity of such a decomposition of the memory part is $\mathcal{O}(mn)$, where n is the number of professors and m is the number of committees. Its response time is $\mathcal{O}(\log m)$ for an update and $\mathcal{O}(n + \log m)$ for a query whether a committee can meet.

The arbitration part of the committee scheduler is based on a list of committees. The scheduler traverses the list, and queries whether committees are ready to meet. We showed that, if the list is fixed, then starvation may occur. Therefore we designed an arbitration part where the list is not fixed, and showed that the arbitration scheme is fair.

Because the scheduler consists of a SEQUENCER and a finite automaton, such a network can still exhibit all possible behaviors, described by a specification of the committee scheduler. Which means that starvation is still possible, although unlikely to occur in practice. We also proved that a trace set of any fair committee scheduler is smaller than the trace set of the specification. This is another demonstration that progress and fairness are not included in our correctness criteria.

We designed a general scheduler in an informal way, which was similar to design of a concurrent program. This design example indicates, once more, that the design of delay-insensitive circuits is a discipline closely related to concurrent programming. It also suggests that the existing knowledge of concurrent programming can be successfully applied to the design of delay-insensitive circuits. For example, our decomposition of the scheduler for the dining philosophers closely resembles Dijkstra's solution of the problem [Dij71].

7.1 Future work

During the design of a committee scheduler we encountered a number of problems which still remain to be resolved.

Decomposition heuristics: A derivation of a decomposition has the advantage of eliminating the verification from the design process. However, a derivation requires a lot of intuition and experience. A collection of heuristics would make the derivation of a decomposition easier. Since complex components are very hard to handle formally, we also believe that it would be good to have a collection of hints, similar to the standard paradigms in program design, that would help with an informal design of a decomposition. Also, a library of efficient decompositions of canonic problems, which are yet to be determined, might prove to be handy.

Progress and fairness: Through a number of examples we demonstrated that our correctness criteria do not prevent the occurrence of deadlock, livelock, and starvation. For example, deadlock is possible in a naive decomposition of the general committee scheduler. Modification of the correctness criteria aimed at making them capable of coping with these phenomena is a topic of current research.

High-level description: The commands language proved to be suitable for efficient description of "small" components. However, for some "large" components we found only lengthy specifications. We believe that a language that would enable specifications and design on a yet higher level would be useful. We would then also require an automatic translation from a high-level language to the commands language; at the level of commands we could then efficiently design "small" components.

Appendix A

Decomposition of a queue-based arbitration module

We give a decomposition of the arbitration module that is based on the idea introduced in Section 6.2.5: the arbitration module uses a queue to determine the sequence of queries. In order to get a complete decomposition of a finite automaton for the committee scheduler we also use the memory module which was introduced in Section 6.2.2 and can be decomposed as shown in Figure 6.11.

We design a queue as a sequence of cells, each representing one position in the queue, and an additional cell that serves as a buffer when we move a committee to the end of the queue. Because each committee can take any place in the queue, every cell has to be able to query for each committee as to whether it may meet. Since the memory module does not distinguish cells from which a query was submitted, we can use MERGEs to collect requests from different cells, and we can use m-by-2 JOINs (where m is the number of committees) as switches to direct responses from the memory module to a cell that submitted a request for a query. Furthermore,

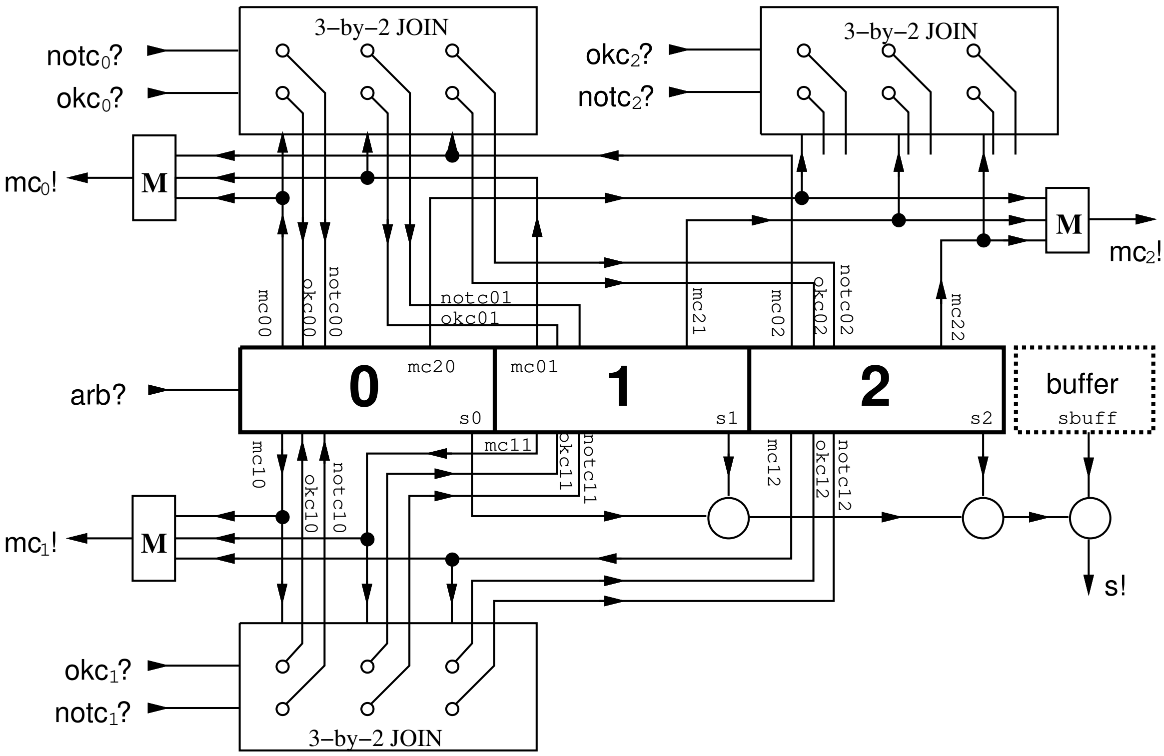


Figure A.1: Manipulation of query-requests.

since the contents of the cells may change, we can use JOINs (connected as a binary tree) to collect signals from queue cells that have completed their updates, and to finally generate the next synchronization signal for the SEQUENCER (see Figure 6.3). In Figure A.1 we show how to use 3-by-2 JOINs to manipulate query-requests from the cells and how to direct answers from the memory module; the underlying committee configuration consists of three committees, c_0, c_1, and c_2.

The next step is to decompose a queue-cell. First we describe how it communicates with its environment. As an example, Figure A.2 shows input and output terminals for cell k that deals with committees c_0, c_1, and c_2. In the following X denotes one of the committees that a cell may query. As in the simple arbitration

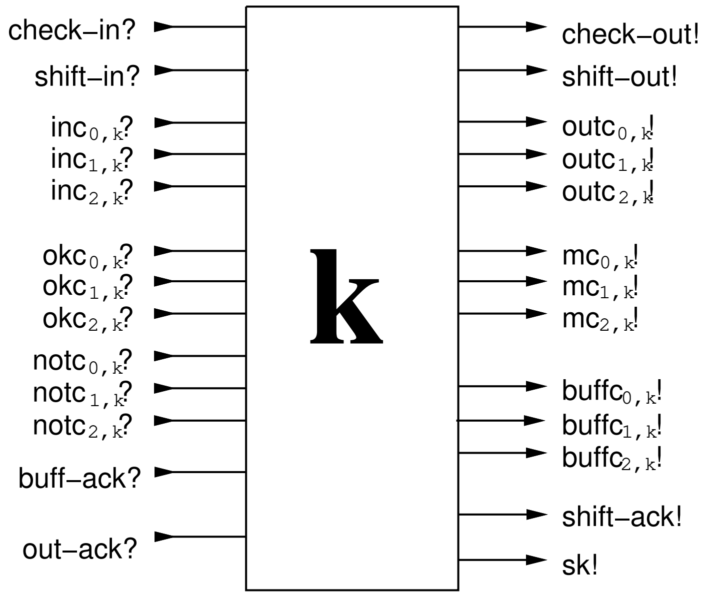


Figure A.2: Queue-cell: input and output terminals.

scheme, described in Section 6.2.2, the arbitration module traverses the queue and checks for every item in the queue whether a corresponding committee is ready to meet. Thus, each cell needs an input that activates the querying (we call this input *check-in*) and an output to activate the next cell (we call this input *check-out*). Furthermore, a cell has output terminals for query requests (mXk) and input terminals to receive a result of a query: $okXk$, if a committee can meet, or $notXk$ if a committee cannot meet.

When we move the contents of a cell to the end of a queue, we first store it in a buffer, then shift the contents of all the cells in the "tail" of the queue one place to the left, and finally write the content of the buffer into the last cell of the queue. Output *buffXk* sends the content of the cell to the buffer; input *buff-ack* serves as an acknowledgement that the buffer has received the data. Furthermore, input *shift-in* serves as a request to send the content of the cell to the left, which is done by one of the outputs *outX*. The receipt of the data is confirmed by input *out-ack*.

When the contents of the cell is moved to the left (or stored in the buffer), the cell requests data from its right neighbor by outputting *shift-out*. The data from the right neighbor is received by one of the inputs *inXk* and is confirmed by sending output *shift-ack*.

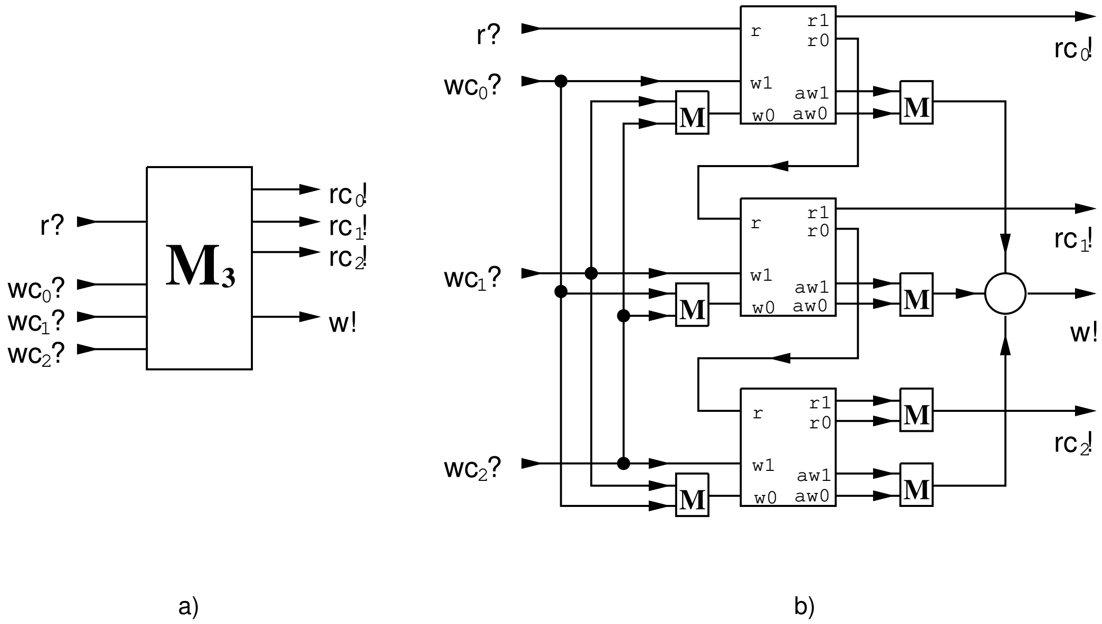


Figure A.3: Memory part of a queue cell: a) schematic b) decomposition.

Note that every queue-cell has to remember for which committee it is responsible. It means that in the decomposition we will need a memory component to store the name of the committee. A memory component can be very similar to the one-bit memory cell we used in Section 6.2.2 (see Figure 6.9); the difference is that now it has to be able to store m different values instead of only two, therefore it has m input and m output terminals. Furthermore, instead of having a separate acknowledgement signal for every possible value stored in a cell, we use a common acknowledgement for every write. A schematic and a decomposition of a memory component that can store three different values are shown in Figure A.3. Note that we use one-bit-memory cells in the decomposition; when we write **true** in a one-bit cell, we have to write **false** into all other cells to preserve the invariant that the memory component stores only one value at a time. Note also that one of the one-bit memory cells has to be initialized to **true** in order to establish an initial position of the committees in the queue. We have to guarantee that each committee is initially assigned to at least one cell.

The area complexity of the memory part of a queue cell is dominated by the MERGEs that collect write-**false** signals. There are m MERGEs, each of them with the area complexity $\mathcal{O}(m)$; thus the area complexity of the memory part in a queue cell is $\mathcal{O}(m^2)$. The response time depends on the operation we perform. To complete a read operation we may need to search through all m one-bit memory cells, therefore the response time is $\mathcal{O}(m)$. To do a write we have to update the content of all one-bit memory cells; this means that we have to generate write-**false** signals for $m-1$ memory cells which takes $\mathcal{O}(\log m)$ time.

Queue cell

Now we are ready to give a decomposition of a queue cell. Figure A.4 shows a decomposition of the k^{th} cell of the queue that stores one of 3 committees, c_0, c_1, and c_2. Both, *check-in* (initiation of a query) and *shift-in* (request to shift the content to the left neighbor) cause a read from the memory part. If a read is triggered by *check-in*, its result should be directed to the memory part of the scheduler as a query, and if read is caused by *shift-in*, the result should be directed to the left-neighboring cell as the shifted data. In order to direct the result of the read to the proper destination we can use an m-by-2 JOIN (m is the number of

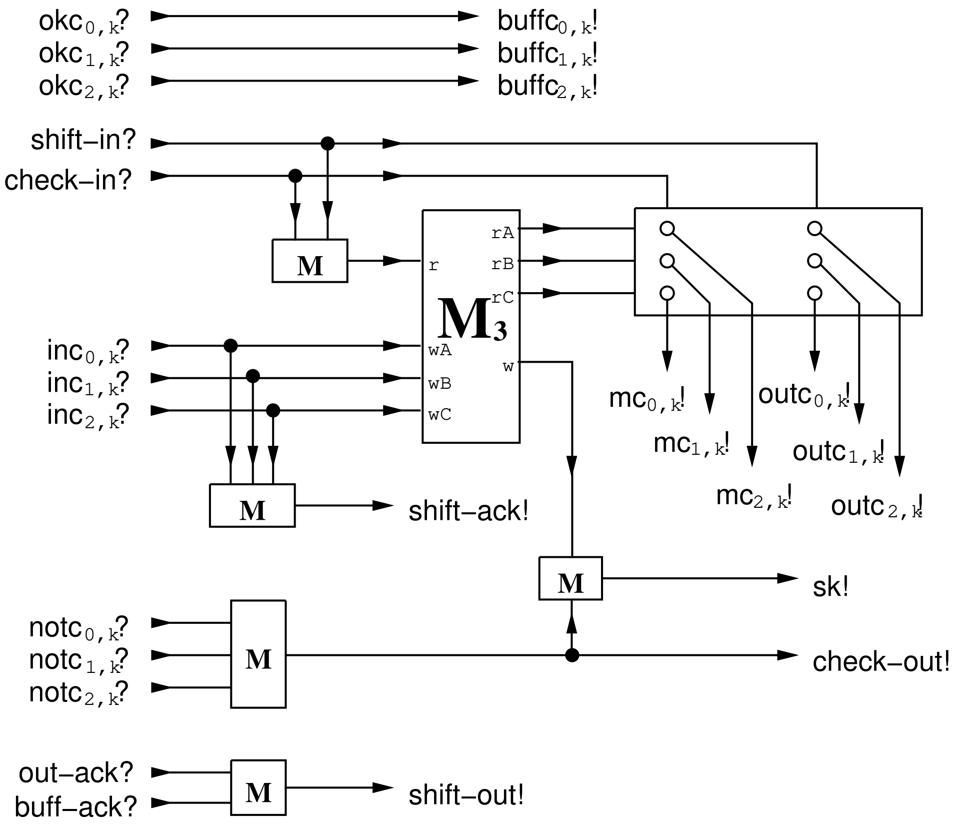


Figure A.4: Decomposition of a queue cell.

committees). If the result of a query is that committee X can meet(where X was stored in the queue cell), then input *okXk* eventually arrives at the cell and causes the cell to write its content to the buffer, i.e., the cell produces output *buffXk*. Note that we do not need an extra read from the memory part of the cell, because the information about the content of the cell is implicitly contained in the signal *okXk*. When the buffer acknowledges that it has stored the content of the cell (input *buff-ack*), the cell reads the content of its right neighbor; it outputs request *shift-out* and eventually receives the data by one of the inputs *inXk*. The cell then, in parallel, acknowledges the receipt of the content of the right neighbor (and thus triggers the next shift along the queue) and writes the new content in the memory.

When the write is completed, the cell outputs *sk* and thus signals that it has been updated.

If the corresponding committee is not ready to meet, the cell activates its right neighbor by producing *check-out* and signals that it has completed its task by producing *sk*.

If a read from the memory part was caused by *shift-in*, a cell exhibits the following behavior. After the content of the cell has been shifted to its left neighbor, the cell eventually receives acknowledgement *out-ack*. A receipt of *out-ack* causes the cell to request a shift from its right neighbor by producing *shift-out*. When the content of the right neighbor has been received (one of the inputs *inXk*), the cell produces acknowledge *shift-ack* and, after updating its memory part, produces output *sk*.

The area complexity of a queue cell is $\mathcal{O}(m^2)$ (it is dominated by the memory part). Because a queue cell communicates with the buffer and with the memory module it is hard to define its response time. For this reason we postpone the performance analysis until we discuss the timing analysis for the complete FA.

Buffer cell

The last missing part of the queue-based committee scheduler is the buffer. As shown in Figures 6.15 and A.1, the buffer can be seen as the (additional) last cell in the queue. There are three essential differences between the functioning of a buffer cell and the functioning of an "ordinary" queue cell:

1. Upon receipt of input *check-in* (which means that none of the committees is ready to meet) it does not perform a query on the memory part of the

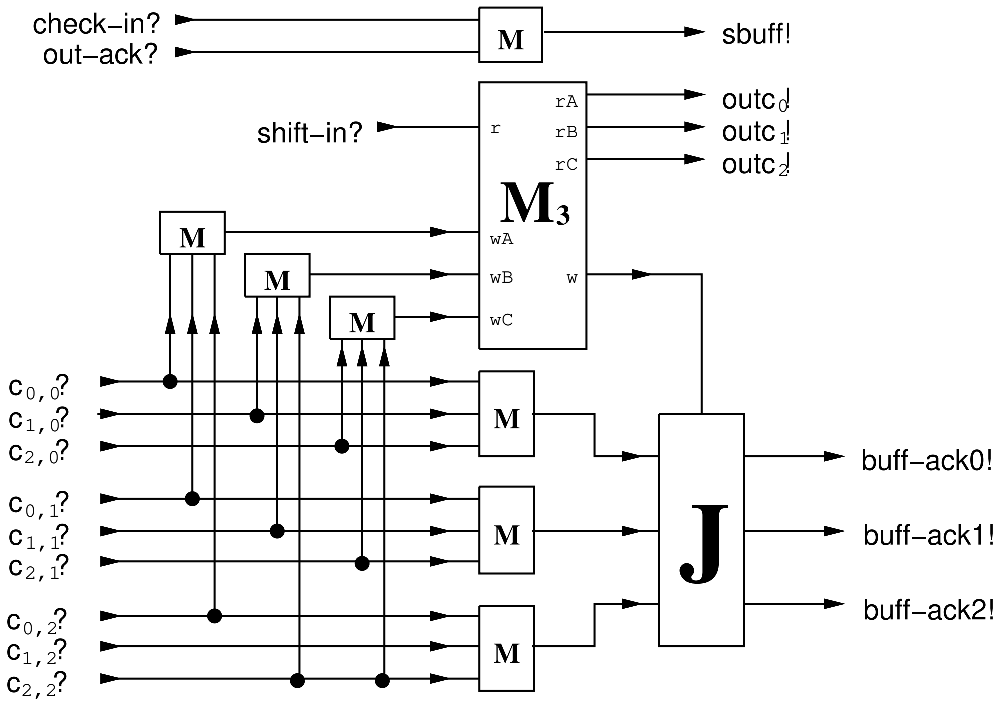


Figure A.5: Decomposition of a buffer cell.

committee scheduler, but immediately produces output *sbuff*, the last in the chain of signals that a cell has completed its work.

2. Upon receipt of an acknowledgement that the content of the buffer has been stored in its left neighbor (input *out-ack*) the buffer does not read the content of the next cell in a queue (there is none), but rather produces output *sbuff* and thus signals that all the cells in the queue have been updated.

3. When the content of a queue-cell is stored in the buffer-cell, the buffer has to produce an acknowledgement and direct it to the cell that sent the data. Again, we can use the n-by-1 JOIN and a number of MERGEs as a switch. Note that an acknowledgement cannot be produced before the data has been actually stored; if the acknowledgement was produced earlier, the read request

(input *shift-in*) could arrive before the memory module is ready to receive it, possibly causing computation interference.

A decomposition of a buffer-cell for a queue that handles three committees, c_0, c_1, and c_2, is shown in Figure A.5.

The area complexity of the buffer cell is $\mathcal{O}(m^2)$: the area complexity of the memory part is $\mathcal{O}(m^2)$ and there are $2m$ MERGEs, each with the area complexity $\mathcal{O}(m)$. From the complexity properties of its memory part we can see that the buffer cell has a response time $\mathcal{O}(m)$ for a read operation and $\mathcal{O}(\log m)$ for a write operation.

Now we can conclude that the area complexity of the arbitration module is $\mathcal{O}(m^3)$, because of the $m+1$ cells with complexity $\mathcal{O}(m^2)$. The area complexity of a complete FA is $\mathcal{O}(mn+m^3)$. The activities that contribute to the response time of the FA can be divided into two phases: (1) an update of the memory module with response time $\mathcal{O}(\log m)$, and (2) traversing the queue and searching for a committee that is ready to meet. Searching for the committee is initiated by queue cells and can be further divided into two basic operations:

querying of the memory module for one committee with response time $\mathcal{O}(m+n)$

shifting the content of one cell with response time $\mathcal{O}(m)$.

Since a query can be more time consuming than a shift and since queries have to be performed sequentially, the worst case occurs when we have to perform a query for every committee. Therefore, the response time of the FA is $\mathcal{O}(m(m+n))$.

Finally, we mention we could introduce more parallelism into the queue. Namely, when we perform a shift, we move the content of the cells to the left one by one, until we reach the end of the queue. However, we could apply the idea used in [EG93a]

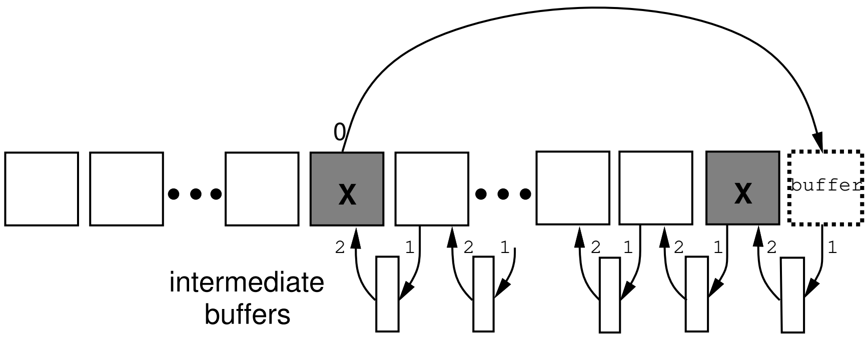


Figure A.6: A shift can be done in parallel.

and introduce an intermediate buffer cell between every pair of consecutive queue-cells (see Figure A.6). Then we perform a shift in three phases: 0) move the content of the current cell (X in Figure A.6) to the buffer at the end of the queue, 1) move the content of every cell in the "tail" to its left buffer, and 2) move the content of the buffer cells to the appropriate queue cells. The design of such a queue would be almost the same as the design of the queue with the sequential shift; therefore, we omit the details. The area complexity would be larger, but still of the same order. The worst-case response time would also not change, because the queries would still have to be performed sequentially. Therefore, it is questionable whether this modification would bring any practical improvement. To obtain a more accurate answer we should consider the expected response time, which means that we should make some assumptions about behaviors of the professors.

Bibliography

[And91] Gregory W. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummins, Redwood City, California, 1991.

[Bag87] Rajive Bagrodia. A Distributed Algorithm to Implement N-Party Rendezvous. *Foundations of Software Technology and Theoretical Computer Science*, 7:139–152, 1987.

[Bag89] Rajive Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, 15(9):1053–1065, 1989.

[BB87] Tomaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BS89] J. A. Brzozowski and C.-J. Seger. A unified framework for race analysis of asynchronous networks. *Journal of the ACM*, 36(1):20–45, 1989.

[Cha87] Arthur Charlesworth. The Multiway Rendezvous. *ACM Transactions on Programming Languages and Systems*, 9(2):350–366, 1987.

[CM73] Thomas J. Chaney and Charles E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22:421–422, 1973.

[CM88] Mani K. Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, Massachusetts, 1988.

[CO89] Michael Coffin and Ronald A. Olsson. An SR Approach to Multiway Rendezvous. *Computer Languages*, 14(4):255–262, 1989.

[DB89] P. Dembinski and S. Budkowski. Specification language Estelle. In M. Diaz, J.-P. Ansart, P. Courtiat, P. Azema, and V. Charbi, editors, *The Formal Description Technique Estelle*, pages 35–75. North-Holland, 1989.

[Dij68] Edsger W. Dijkstra. Cooperating Sequential Processes. In Genuys F., editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.

[Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[Dil89] David L. Dill. *Trace Theory for Automatic Hierachical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[Ebe89] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*. CWI Tract 56, Centre for Mathematics and Computer Science, Amsterdam, 1989.

[Ebe91a] Jo C. Ebergen. A Formal Approach to Designing Delay-Insensitive Circuits. *Distributed Computing*, 5(3):107–119, 1991.

[Ebe91b] Jo C. Ebergen. Separation theorem. Unpublished manuscript, 1991.

[Ebe93] Jo C. Ebergen. Personal communication, 1993.

[EBG93] Jo C. Ebergen, Philippe F. Bertrand, and Sylvain Gingras. Solving a mutual exclusion problem with the RGD arbiter. In S. Furber and M. Edwards, editors, *IFIP WG 10.5 Working Conference on Asynchronous Design Methodologies*. Elsevier, Amsterdam, 1993.

[EG93a] Jo C. Ebergen and Sylvain Gingras. An asynchronous stack with constant response time. Technical report, University of Waterloo, 1993.

[EG93b] Jo C. Ebergen and Sylvain Gingras. A verifier for network decompositions of command-based specifications. In *Proceedings of the 26^{th} Annual Hawaii International Conference on System Sciences*, pages 310–318, 1993.

[FHT86] Nissim Francez, Brent Hailpern, and Gadi Taulbert. Script: a Communication Abstraction Mechanism and its Verification. *Science of Computer Programming*, 6(1):35–88, 1986.

[GvB91] Qiang Gao and Gregor v. Bochmann. Distributed implementation of LOTOS multi-Rendezvous. In *Proceedings of the 9^{th} IFIP Symposium on Protocol Specification, Testing, and Verification*, 1991.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[ISO89] ISO IS 8807. Information processing systems — open systems interconnection: LOTOS - a formal description technique based on the temporal ordering of observational behavior, 1989.

[Kel74] Robert M. Keller. Towards a theory of universal speed-independent modules. *IEEE Transactions on Computers*, C-23(1):21–33, 1974.

[Kum90] D. Kumar. An Implementation of N-Party Synchronization Using Tokens. In *1990 IEEE 10th International Conference on Distributed Computing Systems*, pages 320–327, 1990.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, 1978.

[Mar86] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1:226–234, 1986.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[Mis91] Jayadev Misra. Phase Synchronization. *Information Processing Letters*, 38:101–105, 1991.

[Pee90] Ad Peeters. Decomposition of delay-insensitive circuits. Computing Science Notes 90/04, Dept. of Math. and C.S., Eindhoven Univ. of Technology, April 1990.

[Ray86] M. Raynal. *Algorithms for Mutual Exclusion.* MIT Press, Cambridge, Massachusetts, 1986.

[SCV91] Riccardo Sisto, Luigi Ciminiera, and Adriano Valenzano. A Protocol for Multirendezvous of LOTOS Processes. *IEEE Transactions on Computers*, 40(1):437–447, 1991.

[Sei80] Charles L. Seitz. Ideas about arbiters. *Lambda*, First quarter(1):10–14, 1980.

[Sjö91] Peter Sjödin. A distributed algorithm for synchronous process communication at ports. In *Proceedings of the 9^{th} IFIP Symposium on Protocol Specification, Testing, and Verification*, 1991.

[SPG91] Avi Silberschatz, Jim Peterson, and Peter Galvin. *Operating Systems Concepts*. Addison-Wesley, 1991.

[Sut89] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, January 1989.

[Udd86] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, 1:197–204, 1986.

[vBNRS88] C. H. (Kees) van Berkel, Cees Niessen, Martin Rem, and Ronald W. J. J. Saeijs. VLSI Programming and Silicon Compilation; A Novel Approach from Phillips Research. In *Proc. International Conf. Computer Design (ICCD)*, pages 150–151, Rye Brook, New York, 1988. Computer Science Press.

[vdS85] Jan L. A. van de Snepscheut. *Trace Theory and VLSI Design*. Springer-Verlag, Heidelberg, 1985.