Delay-Insensitive Solutions to the Committee Problem

Igor Benko, Jo C. Ebergen
Department of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1, Canada

Abstract

The committee problem involves the specification and design of a scheduler for committee meetings. It is a general resource allocation problem that combines both synchronization and mutual exclusion. We give a simple specification of a scheduler and present three delay-insensitive implementations. Our last implementation contains a high degree of parallelism and is simpler than previously proposed implementations.

1 Introduction

The committee problem captures many synchronization and exclusion problems that arise in a parallel system. In [4] the problem was described in terms of professors and committees. Here is a slightly edited version.

> Professors in a certain university organize 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. She waits until a meeting of some committee of which she 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; (2) no two committees may meet simultaneously if they have a common member. Given that all meetings terminate in a finite time, the problem is to devise a committee-meeting scheduler that satisfies these restrictions. The scheduler must also guarantee that when all members of a committee are waiting, at least one of them will attend some meeting.

It is clear that the problem includes aspects of both synchronization and mutual exclusion. Synchronization is required for calling a committee meeting, because all members of the committee have to be waiting; mutual exclusion is required, because two committees with a common member cannot meet at the same time.

We are asked to specify a committee scheduler and to design an implementation for the scheduler in terms of an asynchronous circuit. The scheduler can be considered a component that receives "ready" messages from professors when they become available for a meeting, and that sends a "call-for-meeting" message to all members of a committee when that committee is ready to meet. All professors communicate concurrently with the scheduler.

Besides being an intellectually challenging problem, the committee problem is of interest for several other reasons. First, the committee problem is a general resource allocation problem occurring in many parallel systems. Many resource allocation problems, like mutual exclusion and dining philosophers, can be seen as special instances of the committee problem. Second, specifying the scheduler in a particular parallel programming language may indicate the level of support in that language for specifying (hardware) components that exhibit a behavior involving both synchronization and mutual-exclusion. Third, finding implementations of a scheduler that are simple enough so that they can be mapped into an asynchronous circuit is of interest to the design of a compiler that maps parallel programs into asynchronous circuits. Finally, since a high degree of internal non-determinism may be involved, implementations for the committee scheduler are particularly interesting for studying progress properties like absence of deadlock and livelock.

The contributions of this paper are the presentation of a concise specification of a committee scheduler in terms of a CSP-like language called *commands*, and three implementations for the scheduler in terms of a delay-insensitive circuit. The first implementation is a straightforward sequential solution. The final implementation is a new and simple hardware implementation and has a high degree of parallelism. The progress properties for this solution will be briefly discussed. All solutions are presented in an informal way.

1.1 Related work

Because of its generality, studies in a number of areas have encountered the committee problem. The committee problem is sometimes also called the *n-party rendezvous* [2] or the *multirendezvous* [17], since it can be seen as a generalization of the *rendezvous* type of synchronization of two processes.

In the area of concurrent programming much work was devoted to solving special cases of the committee problem for various models of a concurrent system. The special cases include the mutual exclusion problem [15], the dining philosophers problem [1, 7], and the barrier synchronization [1, 13]. Various solutions for the general case of the committee problem were also proposed: [2] proposes a token-ring based solution, [12] and [14] propose solutions that are based on a resource ordering strategy [16]; [4] adopts a solution to the dining philosophers problem ("hygienic philosophers") to solve the committee problem. Our solutions to the general case are simpler than any of these previously proposed solutions and can be implemented easily in an asynchronous circuit.

In the area of programming languages various synchronization constructs have been proposed to solve special cases of the committee problem. Two examples are the *compact* [5] and the *script* [10]. An elegant solution for programming special cases of the committee problem exploiting features of the language SR was proposed in [6]. In [5] it is argued that none of the languages CSP, Ada, SR, and Distributed processing provides adequate support for programming the n-party rendezvous. We show that a simple CSP-like language allows a simple specification of the committee problem.

In the area of formal description techniques the committee problem is inherently present in the language LOTOS, which has been used mostly for formal descriptions of OSI protocols. A solution to the committee problem is therefore a key issue in developing tools for executing LOTOS "programs". In this context [17] proposes an algorithm where processes are organized in a tree and every parental process acts as a synchronization manager for its children. Our solution is less general, but much simpler than the algorithm proposed in [17].

2 Some notation and rules of the game

In order to specify the behavior of a component we use a simple language called *commands*, which is similar to Dijkstra's "guarded commands" extended with input and output actions. We illustrate the basic constructs of the language by giving specifications for some of our basic components. Let us begin with the specification of the WIRE. A WIRE with input terminal a and output terminal b is specified as

$$\text{WIRE} = *[\,a?;b!\,]$$

This specification is interpreted as follows. The environment is initially allowed to produce an input on terminal a. After receiving this input, the WIRE may produce an output on terminal b, then the behavior may repeat. The set of all behaviors that the WIRE can exhibit, also called the *trace set* of the WIRE, is $\{\varepsilon, a, ab, aba, \ldots\}$. The trace set that corresponds to the behavior of a component is always nonempty and prefix-closed.

The specification above shows several constructs to compose commands: "$a?$" denotes the receipt of input a, "$b!$" denotes sending of output b, concatenation is denoted by a semicolon, and repetition of a behavior described by command E is denoted by "$*[\,E\,]$". For obtaining the semantics of a command we always take the prefix-closure of the trace set of the expression.

Another operation on commands is *selection*, denoted by "$E \mid F$", where E and F are commands. For example, the MERGE is a component with two inputs, a and b, and one output, c. Initially, the environment is allowed to produce *only one input*, either a or b. After an input has been received, the MERGE can produce output c, then the behavior may repeat. Such a behavior can be described as:

$$\text{MERGE} = *[\,(a? \mid b?);\ c!\,]$$

The last operation we consider is *weaving*, denoted by $E||F$, where E and F are commands. Weaving can be described as a parallel composition with synchronization on common symbols. It is used to express the conjunction of behaviors described by commands E and F. For example, the JOIN is a component with two inputs, say a and b, and one output, c. After the environment has produced *both* inputs a and b, the component may produce output c, then the behavior may repeat. There is no restriction on the order in which the environment may produce inputs a and b. This behavior can be seen as a *conjunction* of two subbehaviors: In one behavior input a and output c alternate, which is specified by $*[a?;c!]$; in another behavior input b and output c alternate, which is specified by $*[b?;c!]$. The conjunction of the two behaviors gives us the specification of the JOIN :

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

The synchronization on common symbols plays a key role in the specification above: Any behavior of the JOIN must be in accordance with both subbehaviors.

The goal of our game is to give a specification of a component E and to implement this component as a network of smaller and simpler components $E_1, \ldots, E_n$. The network of components must satisfy conditions similar to those described in [9]. In particular, we do not allow any dangling terminals, and outputs of distinct components must not be connected. Furthermore, in the network there must be absence of computation interference. In other words, if an output may be produced by a component, all components connected to the corresponding output terminal must be ready to accept an input. The network must also satisfy some progress conditions. Although absence of deadlock is not required in [9], we do investigate deadlock properties of our implementations.

In our implementations we do not make any assumptions about delays incurred in components. We assume, however, that each communication occurs instantaneously. Because varying the speed of constituent components cannot affect the correctness of the implementation, this type of network is called a *speed-independent* circuit. If the correctness of the implementation is also invariant under delays in interconnecting wires, the network is called a *delay-insensitive* (DI) circuit. Satisfaction of our correctness concerns in general do not imply a DI circuit. However, if all components in a network are DI components, i.e., the specification of the component is insensitive to delays in input and output wires, the implementation can be considered a delay-insensitive circuit. Since we only use DI components, all our implementations are delay-insensitive.

Table 1 shows a number of components that we use as basic building blocks for our implementations. All the components shown are DI components. For a formalization of a speed-independent circuit, a delay-insensitive circuit, and DI components we refer to [9].

Specification	Schematic
WIRE $= *[a?;b!]$	
IWIRE $= *[b!;a?]$	
JOIN $= *[a?;c!] \;\|\; *[b?;c!]$	
MERGE $= *[(a? \mid b?);c!]$	
SEQUENCER $= *[a?;ga!] \;\|\; *[b?;gb!] \;\|\; *[d?;(ga! \mid gb!)]$	
CALL $= *[\ c1?;c!;\ (t?;t1! \mid f?;f1!)\ \mid c2?;c!;\ (t?;t2! \mid f?;f2!)\ ]$	

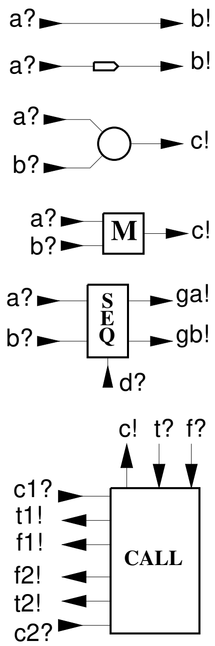

Table 1: Some basic components

3 Specification

In this section we give a specification of a committee-meeting scheduler. We first consider schedulers for two special cases of the committee problem and then we give a specification for a general-case scheduler.

Consider once again the specification of the JOIN

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

Because c is common to both "weavands", the specification prescribes that output c may be produced only after *both* inputs a and b have arrived. We can consider the JOIN as a committee scheduler for a committee consisting of two professors: $C = \{p_a, p_b\}$: Inputs a and b are ready messages from professors p_a and p_b respectively, and output c is the signal that both members of a committee are available and that a meeting can begin. If we generalize the JOIN to the n-input JOIN , we get a committee scheduler for the barrier-synchronization problem [1, 13]: When a process reaches the barrier it may not continue until the barrier is reached by all processes. Crossing the barrier can be seen as a meeting of a committee that has all processes on its roster.

In the next special case there are n committees, each with two members: $C_i = \{p_i, cs\}$, where $0 \leq i < n$. Since professor cs is a member of all committees (probably a department chair), only one meeting can be held at any time. Let us examine the communication between professor p_i and the scheduler. Professor p_i signals her availability by sending message r_i to the scheduler. The scheduler may later send message m_i to professor p_i to announce that committee C_i will meet. From the scheduler's point of view this communication can be specified as

$$*[r_i?;\ m_i!] \qquad 0 \leq i < n$$

The communication between professor cs and the scheduler is as follows. After the scheduler receives ready message r_{cs} it may send professor cs to a meeting of one of the committees C_0, C_1, ..., C_{n-1}. This behavior can be specified as

$$*[r_{cs}?;\ (m_0! \mid m_1! \mid \ldots \mid m_{n-1}!)\,]$$

The complete behavior of the committee-meeting scheduler is the conjunction of communications between the scheduler and the professors, therefore it can be specified as the weave

$$\begin{array}{lll} \mathsf{mutex} & = & \quad *\,[r_0?;\ m_0!] \\ & & \|\ *\,[r_1?;\ m_1!] \\ & & \vdots \\ & & \|\ *\,[r_{n-1}?;\ m_{n-1}!] \\ & & \|\ *\,[r_{cs}?;\ (m_0! \mid m_1! \mid \ldots \mid m_{n-1}!)\,] \end{array}$$

Note that, because of the synchronization on the common symbols, output m_i can be produced only if both input r_i and input r_{cs} have been received. Note also that between any two consecutive outputs m_i and m_j input r_{cs} must be received.

We can interpret this special case of the committee problem as the classic mutual-exclusion problem [15]. Professor p_i can be seen as a process requesting entry to the critical section, which is represented by professor cs. Input r_{cs} signals that the critical section is available, process i requests an access to the critical section by sending r_i, and output m_i is a signal that process i may enter the critical section. We remark that the committee scheduler for the mutual-exclusion problem can be implemented by an n-input SEQUENCER.

The dining-philosophers problem is another special case of the committee problem. Suppose there are n philosophers p_0, ..., p_{n-1}, and n forks f_0, ..., f_{n-1}, such that philosopher p_i can use only forks f_i and $f_{(i+1) \bmod n}$. In this case we can see the philosophers and the forks as members of committees, where each committee consists of one philosopher and the two forks that the philosopher can use. The specification of the committee-meeting scheduler for the dining-philosophers problem is left as an exercise to the reader.

3.1 The general case

Let $P = \{p_0, \ldots, p_{n-1}\}$ be a set of n professors, and let $S = \{C_0, \ldots, C_{k-1}\}$ be a set of k committees. 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 in isolation and then combine all these subbehaviors into a specification. When professor p_i is ready to attend a meeting, she sends ready message r_i to the scheduler. After the scheduler has determined that a committee C_j can meet, it sends a message m_j to the members. Thus, after receiving r_i, the scheduler may send to the professor one of the messages m_j, where $p_i \in C_j$. We specify such a partial behavior as

$$*[r_i?;\ (\,|j\,:\,p_i \in C_j\,:\,m_j!\,)]$$

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

$$\begin{array}{lll} \mathsf{scheduler} & = & \quad *\,[r_0?;\ (\,|j\,:\,p_0 \in C_j\,:\,m_j!\,)] \\ & & \|\ *\,[r_1?;\ (\,|j\,:\,p_1 \in C_j\,:\,m_j!\,)] \\ & & \vdots \\ & & \|\ *\,[r_{n-1}?;\ (\,|j\,:\,p_{n-1} \in C_j\,:\,m_j!\,)] \end{array}$$

Synchronization on common symbols is the key issue in this specification. When committee C_j meets, all weavands corresponding to the members of the committee synchronize on signal m_j.

Example 1: Let $S = \{A, B, C\}$ be the set of committees, and let $P = \{a, b, c\}$ be the set of professors. Furthermore, let the committee rosters be

$$A = \{a, b\} \quad B = \{b, c\} \quad C = \{a, c\}$$

This configuration gives us the following specification (a, b, and c denote ready signals, and A, B, and C denote calls for meetings for the corresponding committees):

$$\begin{array}{lcl} \mathsf{scheduler} & = & \quad *[\,a?;\ (A! \mid C!)\,] \\ & & \| \ *[\,b?;\ (A! \mid B!)\,] \\ & & \| \ *[\,c?;\ (B! \mid C!)\,] \end{array}$$

□

Let us verify whether our specification indeed meets the two restrictions on committee meetings. Because of the synchronization on a common symbol each committee can meet only if all its members are ready. For example, committee A can meet only after *both* professors a and b have become available. Furthermore, no two committees can meet simultaneously if they have a common member. For example, if all three professors are available and committee A meets, professors a and b cannot attend another meeting before the scheduler receives their new ready signals.

4 Implementations

In this section we propose three implementations of a general committee scheduler. We obtain all solutions using the same strategy: We first write a program that corresponds to a solution, and then we implement the program with a network of DI components. Derivations of efficient implementations of committee schedulers for a number of special cases can be found in [3].

4.1 Straightforward solution

Assume the committee scheduler maintains Boolean variables $ready[i]$, where $0 \leq i < n$, such that $ready[i]$ is equal to true if and only if professor i is available for a meeting. All elements of the array must be initialized to false. After the scheduler has received a ready message from professor i, $ready[i]$ is set to true , and it remains true until professor i attends a meeting. When input r_i has been recorded in $ready[i]$, the scheduler can inspect the array *ready* to determine whether there is a committee for which all members are available for a meeting. If such a committee is found, all elements of *ready* corresponding to the members of that committee are reset to false , and an output that initiates a meeting of that committee is produced.

We can design the scheduler as a collection of n processes, each processing ready messages from one professor. After receiving a ready message, a process inspects array *ready* to determine whether any committee can meet, updates the array accordingly, and, if a committee can meet, sends a message to initiate the meeting. Note that the scheduler can process only one ready message at a time. If two messages were processed at the same time, each of the messages could enable a meeting of a different committee and two committees with the same member would be allowed to meet concurrently. The processes use a binary semaphore to ensure mutually exclusive access to shared array *ready*. Under these assumptions, the process that processes ready messages from professor p_i can be described as follows:

```
process Professor(i : integer) =
shared ready: array [0..#Profs) of boolean;
        s: semaphore;
|[ var j: integer;
   *[ r_i?; P(s);      { get ready message, lock ready }
      ready[i], j := true , 0;          { update ready }
      do (j < #Committees) then
        if (⋀ l : p_l ∈ C_j : ready[l]) then
          { committee j can meet }
          (forall l : p_l ∈ C_j : ready[l]:= false ); m_j!
        fi ;
        j := j + 1
      od ;
      V(s) ]
]|
```

In the program above we extended our basic language with the standard guarded-commands control structures. We also used $|[\ldots]|$ to denote the scope of variables and the **forall** construct to denote that the order in which the statements in its body are executed does not matter.

The condition in the **if** statement guarantees that a committee meets only when all its members are available. Since all committees are inspected sequentially, and since the usage of the semaphore guarantees that only one process at a time can access *ready*, it is guaranteed that a professor attends at most one meeting at a time. It is also guaranteed that if all members of a committee are ready to meet, one will attend some meeting: Let p be the the last member of committee A that is ready for a meeting. When the ready message from p is processed, either A is allowed to meet or p attends a meeting of another committee. The usage of semaphore s guarantees that access to the whole array *ready* is mutually exclusive and also guarantees that deadlock cannot occur.

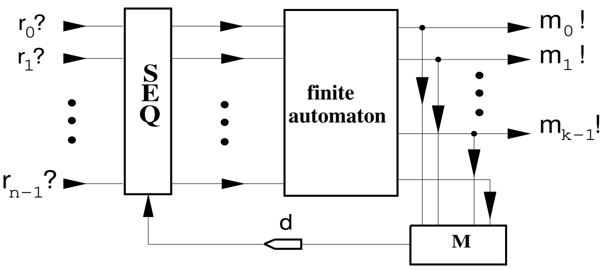

Figure 1: FA-based implementation of a committee scheduler

An outline of a delay-insensitive implementation of the scheduler described above is shown in Figure 1. The ready signals from professors are connected to "request" ports of a SEQUENCER , which is used to implement the binary semaphore. A P operation succeeds when the arbiter grants the request, and a V operation is performed when the SEQUENCER receives input d. The remaining part of the processes can be implemented as a sequential finite automaton that stores the current state of the committee scheduler and, upon receiving a ready signal, outputs a signal to initiate a committee meeting, if all members of that committee are available. After the ready signal has been recorded and processed, the finite automaton enables processing of the next ready message by producing output d. Detailed implementation of the finite automaton is beyond the scope of this paper. We mention that the number of states required may grow exponentially with the number of professors. Therefore, a non-trivial implementation may be required [3, 8].

4.2 Token-ring based solution

In the implementation of the committee scheduler described above, every ready message is processed separately, because we want to ensure that two committees with the same member do not meet at the same time. One consequence of this scheme is that two ready messages cannot be recorded at the same time, even if the corresponding professors are members of different committees. In our next implementation, queries whether committees can meet are performed only one at a time, but ready messages can be recorded concurrently.

Imagine the scheduler being composed of two kinds of processes. For each professor we have one process that receives her ready message and records it in array *ready*; for each committee there is one process called "committee controller", whose job is to determine whether the committee is ready to meet. If the committee can meet, the controller updates array *ready*, and sends a signal to initiate a meeting. Note that $ready[i]$ is a variable shared among committee controllers for all committees that have professor i on their roster and the process that receives ready messages from professor i. In order to ensure mutually-exclusive access to each $ready[i]$ individually, we replace semaphore s used in the previous solution with semaphores $s[i]$, one for each element of array *ready*. The process for receiving ready messages is then:

process Prof-controller$_i$ =
shared $ready[i]$: boolean;
        $s[i]$: semaphore;
|[
  $*$[r_i?; P($s[i]$); $ready[i]$:= true ; V($s[i]$)]
]|

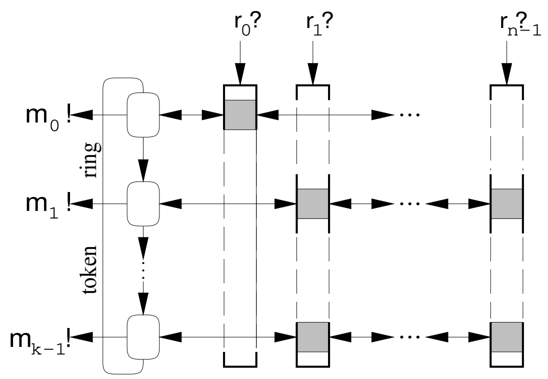

Figure 2: An outline of the token-ring based solution

Since only one committee controller can be active at a time, we connect the controllers into a token-ring as shown in Figure 2. Only the controller that has a token can query whether its committee is ready to meet. We also assign a distinct number to each professor and introduce two functions: $first(A)$ returns the number of the member of committee A whose availability is checked first; $next(p, A)$ returns the number of the member of A that is queried after professor p. If $next(p, A)$ is called with p being the last member of A to check, the function returns the number that is larger than the number of professors. We remark that the order in which members of a committee are queried is arbitrary and can be different for different committees. In fact, members of each committee could be queried in parallel. With assumptions above the controller for committee A can be described as

```
process Committee-controller(A: committee) =
shared ready: array [0..#Profs) of boolean;
       s: array [0..#Profs) of semaphore;
|[ *[ get_token?
      if Query(A,first(A)) then m_A! || send_token!
      else send_token!
      fi
    ]
]|

procedure Query(A: committee; j: integer) =
shared ready: array [0..#Profs) of boolean;
       s: array [0..#Profs) of semaphore;
|[    if j > #Profs then
         return true { all professors are available }
      else P(s_j);
           if ready[j] then
              { query the rest of the committee }
              if Query(A, next(j,A)) then
                 ready[j] := false ; V(s_j) || return true
              else V(s_j) || return false
              fi
           else V(s_j) || return false
           fi
      fi
]|
```

From properties of function $next$ it follows that a committee meets only if all its members are ready for a meeting. Only one committee controller at a time holds the token, and, in case of a meeting, the elements of $ready$ are set to false before all semaphores are released. This means that a professor can attend only one meeting at a time; she has to declare her availability again before she can attend another meeting. If all members of committee A are available for a meeting then either the controller for committee A will get the token and initiate a meeting of A, or some member of A will go to a meeting of another committee that gets the token before A, that is, some member of A will attend a meeting.

How can the algorithm above be implemented using delay-insensitive circuits? We first consider the token-ring part of the committee controller. When the token arrives, the controller initiates a query for a committee meeting by producing output $query$. If the committee cannot meet, the query returns $false$ and the controller just has to pass the token along the ring. If the committee can meet, input $true$ arrives. The controller produces output m to initiate a meeting of the committee and passes the token along the ring. An implementation of this part of the committee controller is shown in Figure 3.

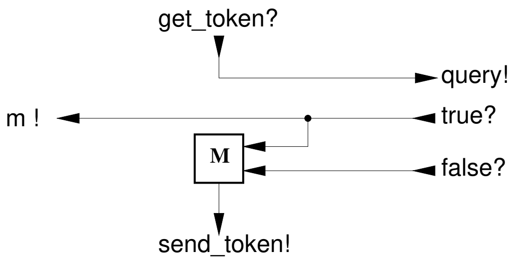

Figure 3: Token-ring part of a committee controller

In order to store the availability status of professor i, recorded in $ready[i]$, we use a one-bit memory cell with dual-rail data encoding, shown in Figure 4. The

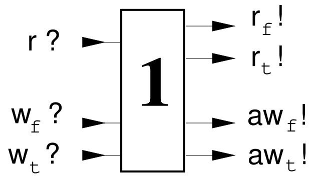

Figure 4: One-bit memory cell

cell responds to a read request r with either output r_f or r_t, depending on its contents. Upon receiving a write request w_f (w_t) the cell updates its contents and responds with aw_f (aw_t). The environment of the cell is not allowed to produce the next input to the cell until the cell has responded to the previous input.

The part of the committee scheduler that deals with storing the state of a professor and that implements queries for a professor's readiness is shown in Figure 5. Access to $ready[i]$ is protected by semaphore $s[i]$,

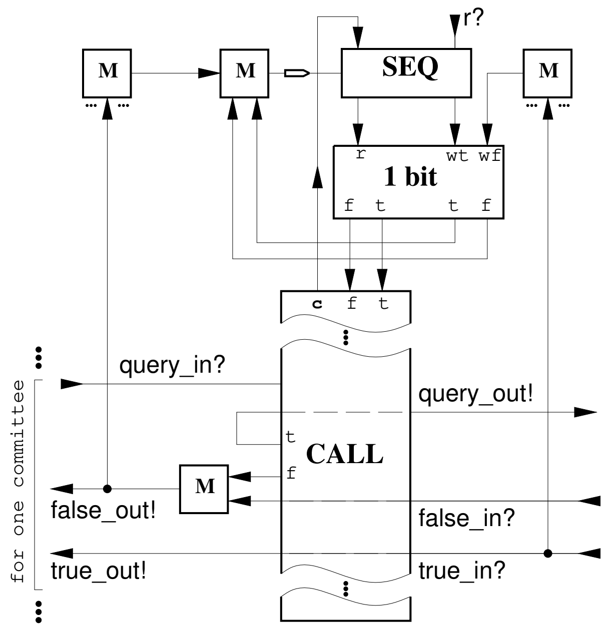

Figure 5: An implementation of Query and Receive

which can be implemented by a SEQUENCER in a similar way as in the previous solution. Suppose professor p is a member of m committees, A being one of them. Request *query_in* can be seen as a call to Query(A, p). The CALL component converts this request into a request for reading the content of the 1-bit memory cell. If the professor is not available, the negative answer *false_out* is generated immediately and the semaphore is released. If the professor is available, the *query_out* signal is generated. This signal is used to initiate a call to Query$(A, next(p, A))$, i.e., to inspect the status of the next member of committee A. Updating the state of professor p can be done for two reasons. When a call to Query$(A, next(p, A))$ returns true , i.e., input *true_in* arrives, we know that the committee can meet, so we use this input to update the contents of the memory cell and to generate the output *true_out*, which can be seen as the value Query(A, p) returns. Updating also occurs when a professor becomes available for a meeting and sends a ready message. The ready message (r) is routed through the SEQUENCER (semaphore), then used to update the content of the cell, and, finally, an acknowledgment of the update is used to produce the "done" signal for the SEQUENCER .

The complete scheduler consists of a token-ring part for each committee and of a query part for each professor. Token-ring parts and query parts are connected according to the structure of committee rosters.

4.3 Parallel queries

In the token-ring solution only one committee can be queried at a time to ensure that two committees with a common member do not meet concurrently. In order to parallelize committee queries we look at professors as resources, and at committee controllers as processes competing for the resources. When a controller acquires all members of its committee, a meeting can be called. This analogy indicates that the committee problem can be solved by a well-known resource-allocation scheme [16] that avoids circular waiting and avoids deadlock: To prevent deadlock we order the professors, and all controllers acquire professors one by one with respect to the ordering. The resource-allocation algorithm assumes that a resource is available when it is not captured by any process. This may not be true if a professor plays the role of a resource: A professor may decide to do some research first, so she may become ready for a meeting after an arbitrary long time. This calls for a modification of the resource-allocation scheme. If a controller learns that a professor is currently not available for a meeting, two possible approaches can be taken. In the first approach the controller simply locks professors that have already been acquired and waits until the professor becomes available for a meeting. This approach works well if it is guaranteed that each professor eventually becomes available for a meeting. However, if there is a professor that may never become available for a meeting, some committee controller could keep a number of available professors locked for an indefinite time, thus possibly preventing some other committee to convene a meeting. Even if all professors are guaranteed to eventually become available for a meeting, this approach could cause long delays before committees can meet, because some professor could submit a ready message after a long period of doing research.

In the second approach a controller, when it learns that a professor is currently not available, releases all captured professors and begins a new query, thus avoiding blocking. A description of such a controller is very similar to the description of a token-ring committee controller. One difference is that now all professors have to be queried with respect to the same ordering. There is also no token the controller would have to wait for, so the controller process becomes

process Controller-parallel(A: committee) =
shared *ready*: **array** [0..#*Profs*) **of** boolean;
$\quad$ *s*: **array** [0..#*Profs*) **of** semaphore;
|[
$\quad *$[**if** Query(A,first(A)) **then** m_A! **fi**]
]|

The function Query is exactly the same as it was for the token-ring solution. The only difference is that functions first and next now have to respect the same global ordering for all committees, whereas in the previous solution the ordering for each committee was arbitrary.

The outline of an implementation of such a committee scheduler for the committee configuration from Example 1 is shown in Figure 6. For every committee first a query is initiated. When an answer is received, a meeting starts if a committee can meet, and a new query is initiated. The part of the scheduler that im-

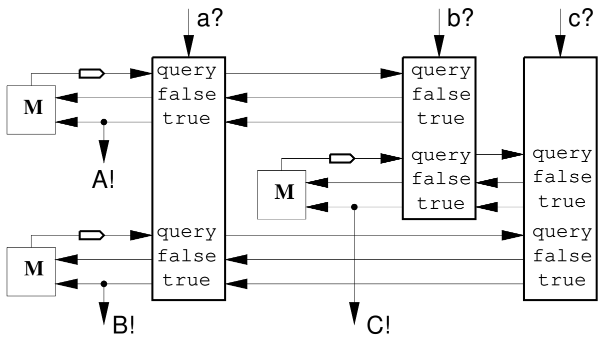

Figure 6: Polling committees in parallel

plements calls to function Query and updates the state of a professor is shown in Figure 7. The role of this part of the scheduler is the same as the role of the corresponding part in the token-ring based scheduler, so we reuse the implementation shown in Figure 5, marked by `Mem` in Figure 7. The difference in usage of this circuit is that now several committee controllers can simultaneously inspect the state of a professor. Since concurrent queries would violate the behavioral constraints for the CALL, we use a SEQUENCER to serialize the queries. After a query has been processed, a "done" signal for the SEQUENCER is generated to enable processing the next query.

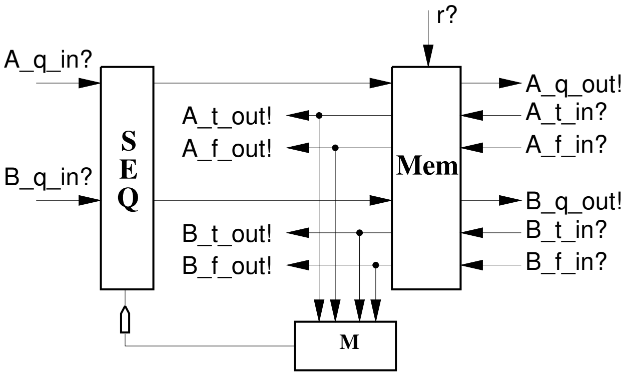

Figure 7: Circuit that performs queries and updates the state of a professor (q = query, t = true , f = false)

5 Concluding Remarks

We have found the committee problem a challenging exercise from which we learned a few lessons.

First, the weave turned out to be a very useful primitive for parallel composition in specifying a component. The weave allowed us to first look at partial behaviors in isolation (sometimes also called "snippets" [18]) and, second, to take the conjunction of these partial behaviors to obtain the complete specification. Because of this strategy, the specification of the general-case committee scheduler was a simple exercise.

Describing and studying an implementation first in terms of a parallel program before mapping it into a delay-insensitive circuit was extremely helpful. For each implementation we first wrote a simple guarded-command program with P and V operations. Once we were convinced the program satisfied the specification, the program was mapped onto a delay-insensitive circuit, where P and V operations where conveniently implemented by means of a SEQUENCER. We believe that such a step was of significant help in reasoning about the correctness of our designs and contributed to the simplicity of our designs.

Modularity in design is one of the main advantages of delay-insensitive circuit design. Although we did not explicitly state this property, we used it several times. For example, in the solution where all committees are queried in parallel, we separately designed the circuit that initiates a query, and the circuit that performs queries and updates the state of a professor. The design of the latter circuit again was broken down into a number of steps: designing a memory cell, designing the circuit that updates the cell, and enforcing serialization of requests in order to satisfy behavioral restrictions for some components. Such an approach considerably reduces the level of complexity of a problem and contributes to a faster design process with fewer errors.

The progress properties of our designs have been discussed in an informal way. For example, we did not give a formal definition of progress or of deadlock, but rather we argued that each solution in terms of a parallel program had absence of deadlock. We also did not consider livelock, where an unbounded number of internal communications can take place before an external communication occurs. Reasoning about progress properties is still an open problem. For promising approaches to solving this problem we refer to [11, 19]. Solutions to the committee problem will probably provide interesting test cases for any tentative definition of progress.

Acknowledgment: The authors would like to thank Theo Norvell for careful reading of an eralier draft of this paper and for many constructive suggestions.

References

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

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

[3] I. Benko. The Committee Problem and Delay-Insensitive Circuits. Master's thesis, University of Waterloo, Department of Computer Science, 1993.

[4] M. K. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Massachusetts, 1988.

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

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

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

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

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

[10] N. Francez, B. Hailpern, and G. Taulbert. Script: a Communication Abstraction Mechanism and its Verification. *Science of Computer Programming*, 6(1):35–88, 1986.

[11] M. Josephs and J. T. Udding. An Overview of DI Algebra. In *Proceedings of the 26th Hawaii International Conference on System Sciences, Vol. 1*, pages 329–338, 1986.

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

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

[14] S. Ramesh. A New and Efficient Implementation of Multiprocess Synchronization. In *PARLE Parallel Architectures and Languages Europe*, pages 387–401, june 1987.

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

[16] A. Silberschatz, J. Peterson, and P. Galvin. *Operating Systems Concepts.* Addison-Wesley, 1991.

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

[18] I. E. Sutherland. The Future Ain't What It Used To Be, The 1993 Organick Memorial Lecture, 1993.

[19] T. Verhoeff. *A Theory of Delay-Insensitive Systems.* PhD thesis, Eindhoven University of Technology, 1994.