Master's Essay
Solutions to the Readers-Writers Problem Using Delay-Insensitive Circuits

Jimmy Ning

Computer Science Department
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

Abstract

The "readers and writers" selective mutual-exclusion problem is a very important synchronization problem. It is often used to compare and contrast synchronization mechanisms. There are three variations of the problem: readers preference, writers preference, and equal preference. In this paper, a specification in a CSP-like notation will be derived for the problem. Schedulers for the readers and writers preference variations of the problem will be designed using asynchronous circuit components.

Acknowledgement

I am very thankful to my supervisor Jo Ebergen for answering my questions and for suggesting improvements while I was in the process of writing this paper.

I thank John Segers for suggesting me to try a tree-structure solution. I would also like to thank Igor Benko for answering my questions and for maintaining a smooth computing environment for me to do my research.

NSERC, ITRC and the Computer Science Department of the University of Waterloo are acknowledged for their financial support.

Contents

1 Introduction

In 1971, Courtois et al[6] proposed the "readers and writers" synchronization problem. The problem consists of two groups of processes, readers and writers, and a shared "file". The readers only examine the content of the "file", and any number of readers may concurrently examine the "file". The writers both examine and alter the content of the "file", and each writer must have exclusive access to the "file". The "file" in the problem is not necessarily a file on a physical disk. It can be a memory location or a record in a file.

The "readers and writers" problem, like the dining philosophers and producer-consumer problems, is often used to compare and contrast synchronization primitives such as semaphores and conditional critical regions[1, 10]. It is important not only as a test for synchronization mechanisms; it is also a very important practical problem. For example, in a banking system, several people may want to examine the balance of an account at the same time while several other people may want to update the balance of that account. Each update of the account (write) has to be done separately from other operations, but inspections of the account (read) can all be done at the same time.

Many software solutions for the "readers and writers" problem have been proposed. The software solutions can be categorized by the kinds of synchronization mechanisms they employ. Some of the synchronization mechanisms used are busy waiting, conditional critical regions, monitors and semaphores. Busy waiting requires the constant checking of a condition until it becomes true. Busy waiting solutions have two approaches to the problem: the "preventive" approach and the "cure" approach. In the "preventive" approach, conflicts between processes are prevented to happen. In the "cure" approach, processes are allowed to execute until a conflict is detected and some processes are required to backtrack. Busy waiting requires the use of shared variables and generally is very complex for large numbers of processes. In a multiprogramming environment, it is generally inefficient, because CPU cycles are wasted in the checking of conditions when they can be better utilized for executing active processes. Solutions to the "readers and writers" problem that use busy waiting will not be discussed in this paper. Please consult Axford[2] and Raynal[8] for solutions that use busy waiting. Solutions that require language constructs such as monitors and conditional critical regions will also not be discussed in this paper, because their asynchronous circuit implementations are not trivial. Please consult Andrews[1] and Silberschatz[10] for solutions that require special language constructs. The Courtois et al's readers-preference solution[1, 6, 10] that utilizes operating primitives such as semaphores will be discussed in this paper, because these primitives are available in most multiprogramming environments such as UNIX and are easily implemented using asynchronous circuits. An asynchronous circuit solution to the problem will also be modelled after this software solution.

There are three variations of the "readers and writers" problem: readers preference, writers preference, and equal preference. In addition to the given mutual exclusion requirement of the problem, in the readers preference solution, new writers are delayed if a reader is waiting and only if no reader is waiting is a delayed writer awakened. In the writers preference solution, new readers are delayed if a writer is waiting and only if no writer is waiting is a delayed reader awakened. In the equal-preference solution, readers and writers take turns alternately if they both are waiting. When a writer finishes writing, all waiting readers are allowed access to the "file", and when the readers finish reading, one waiting writer is allowed to modify the "file".

In the first part of this paper, the scheduler for the readers and writers problem will be specified using the CSP-like notation described in section 3 and Ebergen[4]. In the second part of this paper, schedulers for different variations of the problem will be designed. A software solution using semaphores for the readers-preference problem will be discussed and a scheduler will be designed in correspondence with the software solution. Following it, a readers-preference scheduler based on a tree-structure will be designed. A writers-preference scheduler will also be discussed. Due to the length of this essay, equal-preference schedulers will not be discussed.

2 Asynchronous Circuits

Conventional circuits are synchronous circuits that perform computations in lockstep with the successive pulses of a clock. The major advantage of synchronous circuits is that they are easy to design. However, due to the past fifty years of rapid development in the field, synchronous circuit technology is approaching its limit. One of the problems with synchronous circuits is that their correctness is dependent on the bounds of delays in elements and wires. Problems can crop up in the design of synchronous circuits, because the satisfaction of the delay requirements cannot be guaranteed under all situations. To avoid these problems, research in asynchronous circuits, circuits without a clock, is becoming more active in recent years. In a recent article in Byte[7], a group at Manchester University is described to be working on a fully asynchronous implementation of a "real-life" commercial architecture. In this paper, we will only be concerned with one type of asynchronous circuit, a delay-insensitive circuit. Informally, the correctness of delay-insensitive circuits is insensitive to delays in both the elements and the wires of the circuits.

The major disadvantage of asynchronous circuits is that they are very difficult to design in comparison to synchronous circuits. Even though they are difficult to design, they have many advantages over synchronous circuits. The following is a short list of advantages. For more information, please consult Ebergen[3] and Pountain[7].

- Asynchronous circuits tend to reflect average-case behavior and synchronous circuits tend to reflect worst-case behavior.
- Asynchronous circuits generally require less energy consumption than their synchronous counterparts.
- Asynchronous circuits exhibit modularity whereas synchronous circuits do not. In an asynchronous circuit, components can simply be replaced by equivalent components that behave identically at the interface but have different designs.
- In the design of asynchronous circuits, there is a clear separation between the physical correctness concerns and the functional correctness concerns.

3 The Command Language

A delay-insensitive circuit can be interpreted as a network of basic elements implementing a specification. The correctness of the implementation is insensitive to the delays in both elements and wires. In the design of delay-insensitive circuits, the functional correctness is separated from the physical correctness. The correctness of basic elements is proved by means of physical principles only. It is only in the design of basic elements that delays in wires are a concern. The correctness of a delay-insensitive circuit (i.e., a network of basic elements) is proved by mathematical principles only. Proving that a delay-insensitive circuit decomposes into a given set of basic elements is often laborious. An automatic verifier can be used to relieve the circuit designer of such burden. To assist the specification of delay-insensitive circuits, the decompositions of these circuits, and the verification of the decompositions, a CSP-like notation is used. Programs written in this language for the specification of delay-insensitive circuits are called *commands*. The CSP-like notation is described in detail in [3] and [4]. The automatic verifier for network decompositions in this language is called VERDECT and is described in [5].

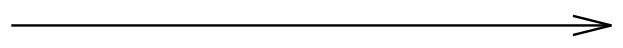
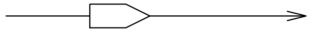
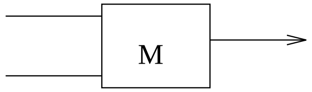
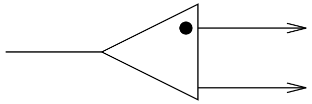
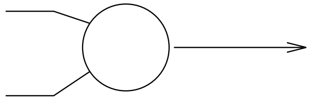
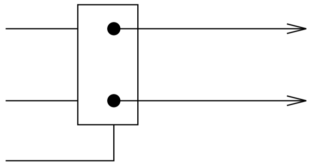
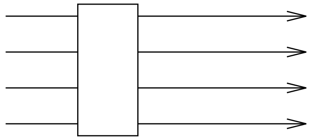
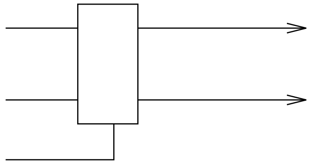
Section 3.1 will introduce the CSP-like notation by specifying some of the basic elements. The communication behaviours of the basic elements will be discussed operationally. For a theoretical approach to the subject, consult the above mentioned articles. Section 3.2 will introduce the up/down counter which was discussed in [9].

3.1 Basic Components

The specification of a component is a contract between the component and its environment (i.e., all the other components it is connected to). The specification describes the communication behaviours between the component and its environment. A component communicates with its environment through its terminals. Each terminal can only be either an input terminal or an output terminal. The communication action at an input terminal is caused by the environment, and the communication action at an output terminal is caused by the component. In order to express that a communication action occurs at an input terminal, the name of the input terminal is postfixed with "?". To express that a communication action occurs at an output terminal, the name of the output terminal is postfixed with "!".

For a WIRE with input terminal a and output terminal b, the initial state is ϵ which means no communication action has occurred. The semicolon, ";", denotes concatenation and is used to specify one communication action occurs after another. If the environment produces a communication action $a?$, the sequence of communication actions at the component is $\epsilon;\ a?$. If the WIRE then produces $b!$, the sequence of communication actions is $\epsilon;\ a?;\ b!$. The communication behaviour $a?;\ b!$ can be repeated any number of times. The set of all finite repetitions of the behaviour $a?;\ b!$ is denoted by $*[a?;\ b!]$. The prefix operator, **pref**, is used to express any prefix of a behaviour. For example, **pref** $(a?;\ b!)$ expresses ϵ, $\epsilon;\ a?$ and $\epsilon;\ a?;\ b!$. To completely express all possible communication behaviours that are possible of a WIRE, the expression **pref** $*[a?;\ b!]$ is used.

The following is a table of some of the basic components:

Name	Specification	Schematic
WIRE	*pref *[a?; b!]*	
IWIRE	*pref *[b!; a?]*	
MERGE	*pref *[(a? | b?); c!]*	
TOGGLE	*pref *[a?; b!; a?; c!]*	
2-input JOIN	*pref *[(a? || b?); c!]*	
2-by-1 JOIN	*pref *[a0? || b?; c0!* *| a1? || b?; c1!* *]*	
RGDA arbiter	*pref *[r0?; g0!; d0?; a0!]* *|| pref *[r1?; g1!; d1?; a1!]* *|| pref *[g0!; d0? | g1!; d1?]*	
SEQUENCER	*pref *[r0?; g0!]* *|| pref *[r1?; g1!]* *|| pref *[d?; (g0! | g1!)]*	

The IWIRE is similar to the WIRE except that the component generates an output *b!* before accepting an input *a?*. The TOGGLE can be used to distinguish between odd and even occurrences of input *a?*. The component generates output *b!* for odd occurrences of the input and *c!* for even occurrences of the input. The or operator, "|", is used in the specification of the MERGE component. If one of the two inputs, *a?* or *b?*, is generated by the environment, then MERGE may generate the output *c!*.

The 2-input JOIN generates an output after receiving its two inputs in parallel. The weave operator, "||", expresses the interleaving of two communication actions with synchronization on common communication actions. The communication behaviour *a?* || *b?* is actually a shorthand for *(a?; b?)* | *(b?; a?)*. Another way of writing the specification for JOIN is

$$\begin{array}{ll} & \mathbf{pref}\ *[a?; c!] \\ \| & \mathbf{pref}\ *[b?; c!] \end{array}$$

This specifies the interleaving of *a?* and *b?* with synchronization on *c!*. An m-input JOIN behaves in a similar way as the 2-input JOIN except that all m inputs have to be received before generating the output. When one of *a0?* and *a1?* is generated by the environment along with *b?*, the corresponding output *c0!* or *c1!* may be generated by a 2-by-1 JOIN. For an m-by-n JOIN (i.e., one set of m inputs and one set of n inputs), there are $m \times n$ outputs where each output corresponds to one combination of one input from each input set.

The RGDA arbiter is used for realizing the mutual exclusive access of processes to their critical sections. RGDA stands for request, grant, done and acknowledgement. When two processes request to enter their critical sections, the RGDA arbiter may grant at most one of them. When a process wants to leave its critical section and sends a done to the arbiter, the arbiter acknowledges it and may grant the entering of another process to its critical section. The SEQUENCER is another component used for realizing mutual exclusion of processes or for sequencing requests from concurrent processes. If one of the requests, *r0?* and *r1?*, and done *d?* are generated by the environment, the corresponding grant *g0!* or *g1!* may be generated by the SEQUENCER. If both requests occur with a done, the SEQUENCER would grant at most one of the requests. The other request would be granted after another done occurs.

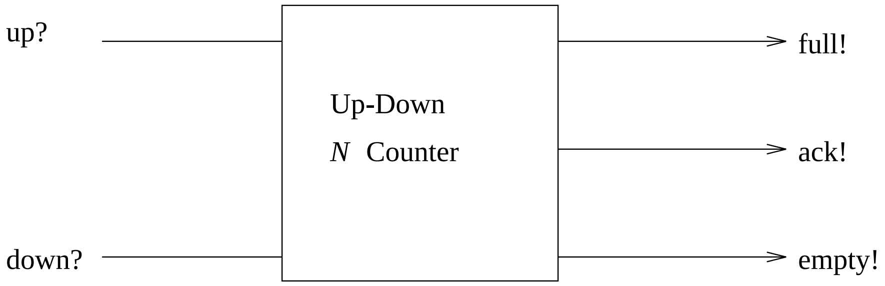
A useful construct in the CSP-like language is the hiding of internal symbols. Internal symbols are introduced at appropriate places in several specifications and are synchronized when these specifications are weaved together. Because internal symbols do not correspond to input or output terminals, they have to be hidden. The format for hiding internal symbols is as follow:

```
|[  'internal symbols' ::
    'subbehaviours 0'
||  'subbehaviours 1'
 .  .
 .  .
 .  .
||  'subbehaviours k'
]|
```

Please note that the weave operator, "||", has the highest precedence followed by concatenation, ";", and selection, "|".

3.2 Up/Down Counter

The asynchronous up/down N counter is described in detail in [9] along with several implementations. In this section, we will briefly look at the up/down counter with its specification. The following is the schematics of the up/down counter:



Initially, the internal count of the counter is 0. After an *up?* action, the internal count increases by 1 and decreases by 1 after a *down?* action. The internal count equals the total number of *up?* actions minus the total number of *down?* actions. After each *up?* or *down?* action, the up/down counter may respond with *full!* if the internal count is N, *empty!* if the internal count is 0 or *ack!* if the internal count is neither N nor 0. The counter assumes that no *up?* action is performed by the environment when the counter is full and no *down?* action when the counter is empty. The specification of the up/down counter is as follows:

$$S.0 = \ \textbf{pref}\ (up?; ack!; S.1)$$

$$S.i = \begin{cases} \textbf{pref}\ (up?; ack!; S.(i+1) \mid down?; empty!; S.(i-1)) & \text{for } i = 1 \text{ and } 2 < N \\ \textbf{pref}\ (up?; ack!; S.(i+1) \mid down?; ack!; S.(i-1)) & \text{for } 1 < i < N-1 \\ \textbf{pref}\ (up?; full!; S.(i+1) \mid down?; ack!; S.(i-1)) & \text{for } 1 < i \text{ and } i = N-1 \\ \textbf{pref}\ (up?; full!; S.(i+1) \mid down?; empty!; S.(i-1)) & \text{for } i = 1 \text{ and } N = 2 \end{cases}$$

$$S.N = \ \textbf{pref}\ (down?; ack!; S.(N-1))$$

The above specification of the up/down N counter has $N+1$ states. State $S.0$ is the initial state. If the counter is at state i, then after receiving and responding to the communication actions for that state, the counter changes its state to either state $i+1$ or $i-1$. For example, if the current state is $S.0$, after receiving *up?* and responding with *ack!*, the counter changes its state to $S.1$. There are several implementations for the Up/Down Counter. The simplest is the unary implementation. The response time of the implementation is determined by the current count of the counter. If the current count is i, for some i larger than one, then the number of internal transitions between the next input (i.e., *up?* or *down?*) and corresponding output is $2(i-1)$. An advanced implementation has a constant response time. For other implementations of the up/down counter, please consult [9].

4 Specification for the Readers and Writers Scheduler

In order to design a scheduler for the "m-readers and n-writers" problem using asynchronous circuit components, the scheduler has to be first specified in a CSP-like language[3, 4]. Requests from readers and writers may come in parallel. The scheduler grants requests from readers if no writer is in its critical section and grants requests from writers if no writer and no reader is in its critical section.

To derive the specification of the scheduler, we first specified the protocol for a reader entering and exiting its critical section. Let the symbol *CS* denotes the critical section. The symbols *rr*, *rg*, *rd* and *ra* stand for the reader's request, grant, done and acknowledgement. The communication behavior, as seen from the reader, can be specified as follow.

$$|[\ CS :: \mathbf{pref}\ *[rr!; rg?; CS; rd!; ra?]\ ]|$$

$$=$$

$$\mathbf{pref}\ *[rr!; rg?; rd!; ra?]$$

The scheduler communicates with m-readers and n-writers. For reader x, we denote the terminals by the symbols *rrx*, *rgx*, *rdx* and *rax*. With respect to these symbols and the scheduler, the behavior can be expressed as follow:

$$\mathbf{pref}\ *[rrx?; rgx!; rdx?; rax!]$$

A similar reasoning hold for writers. The symbols *wry*, *wgy*, *wdy* and *way* stand for writer y's request, grant, done and acknowledgement. The communication behavior with respect to the scheduler can be specified as follow.

$$\mathbf{pref}\ *[wry?; wgy!; wdy?; way!]$$

Writer processes exclude each other; this means that after issuing a grant to a writer and before receiving a done, the scheduler cannot issue a grant to another writer. This mutual exclusion condition can be expressed by the following for writer y and writer z.

$$\mathbf{pref}\ *[wgy!; wdy? \mid wgz!; wdz?]$$

Readers as a class exclude writers. The mutual exclusion condition between a reader and a writer can be expressed by the following for reader x and writer y.

$$\mathbf{pref}\ *[rgx!; rdx? \mid wgy!; wdy?]$$

Since each reader excludes all writers and writer processes exclude each other, the above two protocols can be combined to express this mutual exclusion for reader x and writers 1 to n.

$$\mathbf{pref}\ *[rgx!; rdx? \mid wg1!; wd1? \mid ... \mid wgn!; wdn?]$$

The following is the specification of the scheduler for m-readers and n-writers. It is derived by weaving together the above protocols.

$$
\begin{array}{ll}
 & \mathbf{pref} \ *[rr1?; rg1!; rd1?; ra1!] \\
|| & ... \\
|| & \mathbf{pref} \ *[rrm?; rgm!; rdm?; ram!] \\
|| & \mathbf{pref} \ *[wr1?; wg1!; wd1?; wa1!] \\
|| & ... \\
|| & \mathbf{pref} \ *[wrn?; wgn!; wdn?; wan!] \\
|| & \mathbf{pref} \ *[rg1!; rd1? \mid wg1!; wd1? \mid ... \mid wgn!; wdn?] \\
|| & ... \\
|| & \mathbf{pref} \ *[rgm!; rdm? \mid wg1!; wd1? \mid ... \mid wgn!; wdn?]
\end{array}
$$

5 Solutions to the Readers-Preference Version

5.1 Courtois' Solution

In 1971, when Courtois, Heymans and Parnas[6] proposed the "readers and writers" problem, they also provided a readers-preference solution and a writers-preference solution using semaphores. Only the readers-preference software solution will be discussed in this paper; the writers-preference software solution will not be discussed due to its complexity.

The "readers and writers" problem is a selective mutual exclusion problem. The segments of instructions in readers and writers that access the "file" are the critical sections of the problem. A writer must have exclusive access to the critical section but any number of readers may execute concurrently. In the readers-preference version of the problem, no reader has to wait unless a writer has already obtained access to its critical section.

In the following readers-preference software solution, two semaphores and a counter are required. The semaphore w is used as a mutual exclusion semaphore for the writer but it is only used by the first reader to enter the critical section and by the last reader to leave the critical section. The counter *readcount* is used to determine which process is the first reader to enter its critical section and which reader is the last reader to leave the critical section. The semaphore *mutex* ensures only one reader will enter or leave its critical section at a time thereby avoid conflict in accessing *readcount*.

```
int readcount = 0;
semaphore mutex = 1, w = 1;


reader:

  P(mutex);
  readcount += 1;
  if (readercount == 1)
     P(w);
  V(mutex);

  /* read from the ''file'' */

  P(mutex);
  readcount -= 1;
  if (readcount == 0)
     V(w);
  V(mutex);


writer:

  P(w);

  /* writing to the ''file'' */

  V(w);
```

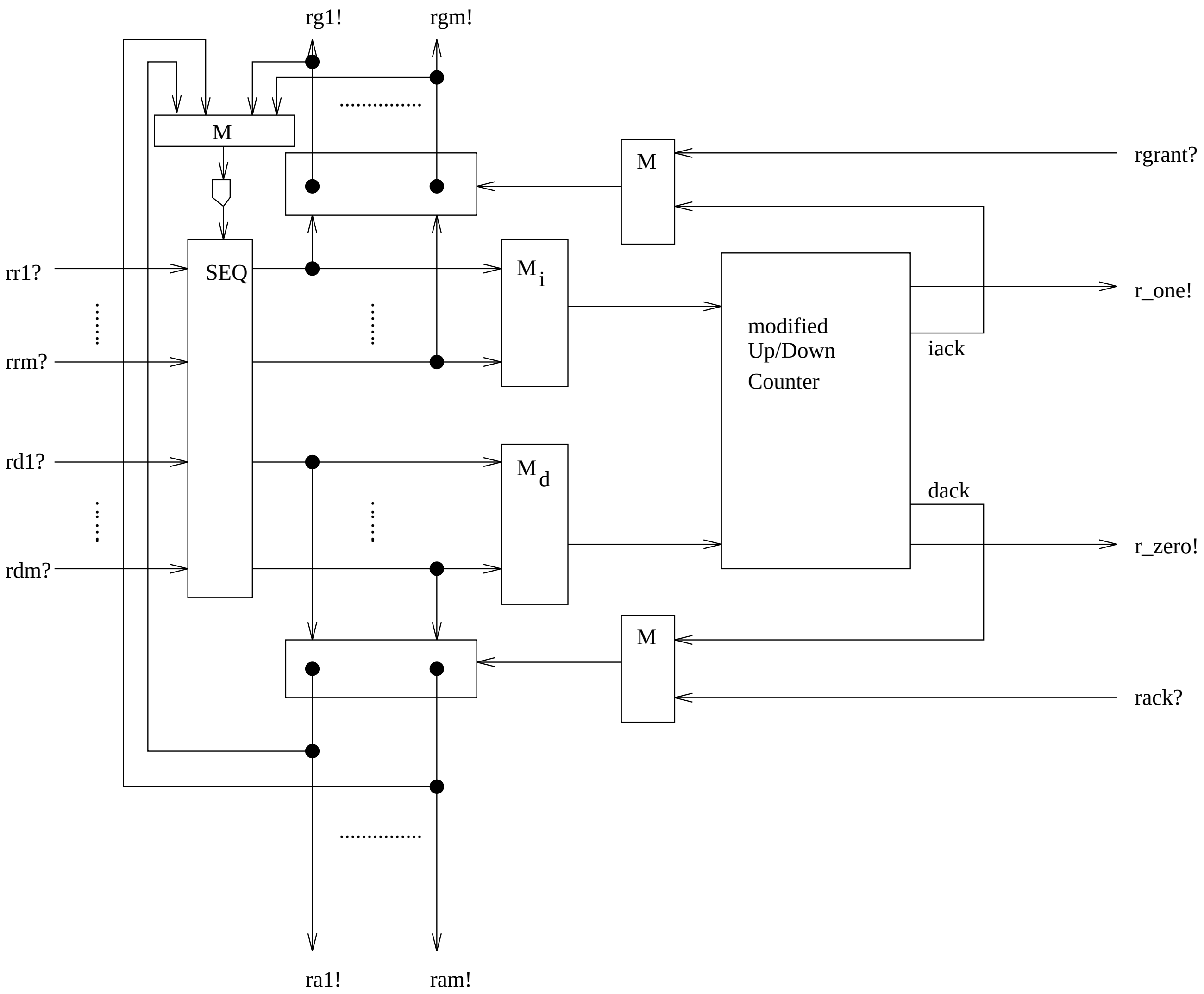
Due to the simplicity of this algorithm, many textbooks use it to illustrate the use of semaphores[1, 8, 10]. However, there is one major problem with the algorithm. It does not suggest solutions to the writers-preference and equal-preference versions of the problem. In Courtois et al[6], a software solution to the writers-preference version of the "readers and writers" problem had also been proposed, but it was a lot more complex than the readers-preference version. The "passing the baton" technique, also uses semaphores, can be adapted easily to various variations of the problem. However, due to the complexity of the technique, it is difficult to translate it to delay-insensitive circuits and will not be discussed in the paper. In the next section, a translation of Courtois et al's readers-preference solution to delay-insensitive circuits will be discussed.

5.2 Delay-Insensitive Solution Based on Courtois' Solution

The Courtois et al's readers preference software solution can be easily translated to a delay-insensitive implementation of the readers and writers scheduler. To implement the software solution in a delay-insensitive circuit, a modified version of the up/down counter is needed. The initial count of the counter is 0. When the first *up?* occurs, the action *one!* is output. If the counter is not 0 when an *up?* occurs, the counter outputs *iack!*. For a *down?* action, the counter outputs *dack!* when the counter is not 0. If after a *down?* action the counter becomes 0, the *zero!* action is output.

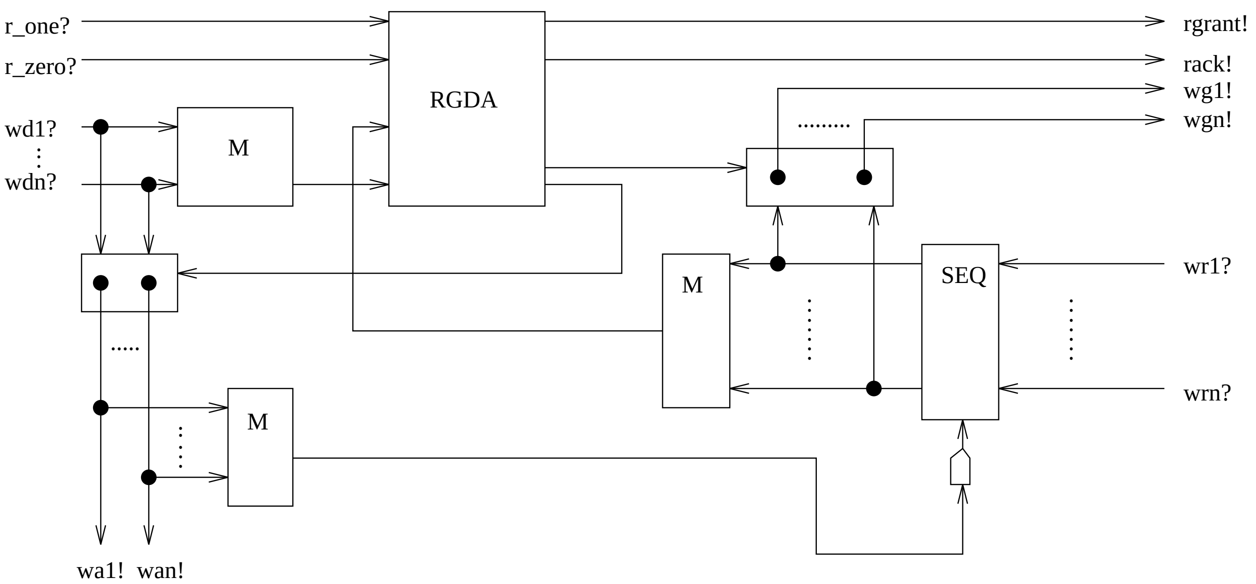
There are two parts to the delay-insensitive circuit that implements the software solution. The first part of the circuit simulates the *mutex* semaphore and the increment of the *readercount* counter in reader processes. The second part simulates the *w* semaphore which is to mutually exclude readers as a class and individual writers.

The following circuit diagram is the first part of the implementation of the readers and writers scheduler.



In this implementation, the effect of the *mutex* semaphore is simulated using a SEQUENCER. It is used to prevent multiple readers from incrementing and decrementing the modified Up/Down Counter, which simulates the *readercount* counter. There is a symmetry between what happens to request actions and done actions leaving the SEQUENCER. A request action after leaving the SEQUENCER would go to an m-by-1 JOIN awaiting for an event to trigger it to grant the request. Similarly, a done action would go to another JOIN awaiting for action to trigger it to acknowledge exiting of a critical section. Since at most one request action would go through the SEQUENCER, all the request actions are passed to a MERGE, M_i. All the done actions go into a different MERGE, M_d. The action from M_i increments the modified up/down counter, and the action from M_d decrements the counter. If the counter is 0 when it is incremented, it sends out an *r_one!* event to the second part of the scheduler. This is used to simulate the calling of *P(w)* by the first reader that wants to enter its critical section. When the requesting reader is allowed to enter its critical section, *r_grant?* would be received from the second part. The action goes to an m-by-1 JOIN and triggers the granting of the appropriate reader. After a grant has been sent out, an action triggers the SEQUENCER to allow another reader request or done to be processed. If the counter is not 0 when an *up?* occurs, the counter outputs *iack* to an m-by-1 JOIN and triggers the JOIN to grant the corresponding request. The action is not propagated to the second part. If the counter becomes 0 after a decrement, the counter would send *r_zero!* to the second part. This is used to simulate the calling of *V(w)* by the last reader to exit its critical section. The second part of the scheduler then responds with *rack?* which goes into the m-by-1 JOIN for acknowledging the appropriate reader. After an acknowledgement is sent, an action triggers the SEQUENCER to allow another reader request or done to be processed. If the counter is not 0 after a decrement, the counter sends out *dack* to the m-by-1 JOIN for acknowledging the done immediately without sending action to the second part of the scheduler.

The second part of the implementation of the readers and writers scheduler is as follows:

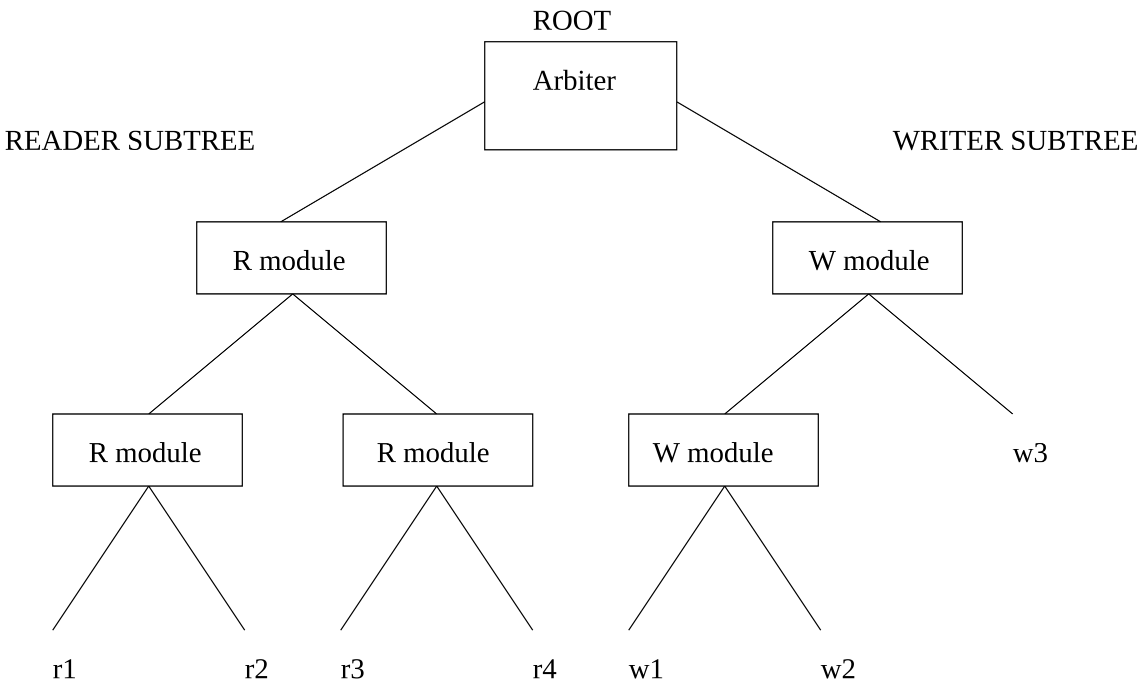


The SEQUENCER is used to realize the mutual exclusion of individual writers from each other. It allows at most one writer request to enter the scheduler. Requests leaving the SEQUENCER would enter an n-by-1 JOIN awaiting to be triggered to send grants to appropriate writers. Unlike reader grants, writer grants do not enter the SEQUENCER because at most one writer can enter its critical section, and no other writer can enter its critical section until the active writer sends a done. Because at most one writer request can leave the SEQUENCER, writer requests are merged into one which is then sent to the RGDA arbiter. If the requesting writer is allowed to enter its critical section, the RGDA arbiter sends an action to trigger the n-by-1 JOIN to send a grant to the appropriate writer. Because at most one writer done can occur at a time, all writer done signals go into an n-by-1 JOIN to be triggered to acknowledge the appropriate writer. Done actions also go to a MERGE which sends an action to the RGDA arbiter to signal the exit of a writer. The RGDA arbiter responds by sending an action to trigger an n-by-1 JOIN to acknowledge the exiting writer. After sending an acknowledgement, an action is sent to the SEQUENCER to trigger the processing of the next writer request. Readers as a class make a request for the entering of their critical sections by sending *r_one?* from part one of the scheduler to the RGDA arbiter. The arbiter responds by sending *rgrant!* back to part one if it is able to grant the request. When the last reader leaves its critical section, part one sends *r_zero?* to the RGDA arbiter, and the arbiter responds by sending *rack!*.

The above delay-insensitive circuit to the readers-preference solution is a direct translation of Courtois et al's software solution. It solves the problem and is very simple. However, its worst disadvantage is that its design is not modular. Given an m-readers and n-writers scheduler, there is no obvious way for making an $m+1$-readers $n+1$-writers scheduler from it. In the next section, a tree-structure scheduler will be examined. The major advantage of the tree-structure solution is that it is modular and is more expandable.

5.3 Tree-Structure Solution

The tree-structure solution to the readers preference readers and writers problem has the advantage of not using an up/down counter and of being modular. Given an m-readers and n-writers scheduler, it is very easy to construct an $m+1$-readers and $n+1$-writers scheduler. The solution has a tree-structure. The following is an example of the structure of a 4 readers and 3 writers scheduler.



There are three parts to the tree-structure solution: the root, the reader subtree and the writer subtree. The root is used to realize the mutual exclusion of writer processes and reader processes. When no process is accessing the "file", the root decides whether a reader or a writer can enter the critical section. The reader subtree is responsible for communicating with reader processes and keeping track of readers that are in their critical sections. Reader processes communicate with the scheduler through the leaves of the reader subtree. The writer subtree is responsible for communicating with writer processes and making sure no two writer processes access their critical sections at the same time. Writer processes communicate with the scheduler through the leaves of the writer subtree.

All the nodes in the reader subtree are identical. Each node is a reader module and can communicate with its parent and at most two children (can be either reader modules or reader processes). Each child can send at most one request to a reader module for accessing the common "file". If no child is in its critical section when a request arrives, the reader module propagates the request to its parent. The module remains inactive until its parent signals the granting of the request. The module then passes the grant to the appropriate child. If one child is in its critical section when a request arrives from the other child, the reader module returns a grant to the requesting child without passing the information to its parent. If the only active child (i.e., one that is in its critical section) requests for exiting from its critical section, the reader module passes the information to its parent and remains inactive until its parent acknowledges. The reader module then passes the acknowledgement to the appropriate child. If a child requests for exiting when the other child is active, the reader module passes an acknowledgment to the child without propagating the information to its parent.

Like the reader subtree, all the nodes in the writer subtree are identical. Each node is a writer module and can communicate with its parent and at most two children (can be either writer modules or writer processes). Each node expects at most one request from each child for entering its critical section. If no child is in its critical section when a request arrives, the writer module passes the request to its parent. The module remains inactive until a grant arrives and the signal is passed to the appropriate child. If a child is active when the another child requests for entering its critical section, the module does not process the request until the active child has exited from its critical section. When a writer module receives a request for exiting from an active child, the module propagates the request to its parent and waits for an acknowledgement. When the module receives the acknowledgement, the acknowledgement is passed to the exiting child. The module can then process another request for accessing the common "file".

Assuming no reader process is accessing the common "file" when a reader process requests for entering its critical section, the request is propagated from a leaf to the root of the reader subtree then is passed to the root of the readers and writers tree. Future reader requests will not be propagated passed the root of the reader subtree. Only when the last reader to request for exiting of its critical section is the request passed to the root of the readers and writers tree. This is similar to the delay-insensitive implementation of the software solution section 5.2. Reader processes make requests for accessing and terminating access to the "file" as a class. All writer requests propagate to the root of the readers and writers tree. Requests for accessing of the "file" would reach the root only if no writer is currently accessing the "file". In other words, a writer request to the arbiter for accessing of the "file" can only be followed by a request for terminating the access of the "file". The arbiter is responsible for realizing the mutual exclusion of the readers as a class and individual writers. The arbiter is simply an RGDA arbiter with the following specification.

$$\begin{array}{ll} & \mathbf{pref} * [sr?; sg!; sd?; sa!] \\ \| & \mathbf{pref} * [tr?; tg!; td?; ta!] \\ \| & \mathbf{pref} * [sg!; sd? \mid tg!; td?] \end{array}$$

The events *sr?*, *sg!*, *sd?* and *sa!* are reader request for entering of a critical section, grant for the request, reader done for exiting of a critical section and acknowledgement for the exit. The events *tr?*, *tg!*, *td?* and *ta!* are writer request for entering of a critical section, grant for the request, writer done for exiting of a critical section and acknowledgement for the exit.

A reader module has three sets of terminals: one for communicating with its parent and one for communicating with each child. Each reader module has at most two children, child 1 and child 2. With respect to the set of terminals for communicating with child 1, the communication protocol is

$$\textbf{pref } *[rr1?; rg1!; rd1?; ra1!]$$

The child may request to enter its critical section. The module may then respond with a grant. The child then sends a done when it wants to exit its critical section. The module can then acknowledge the exit. When the module receives a request for accessing the "file", it changes its internal state and/or propagates the request to its parent. The internal symbols *x1* and *y1* are placed between the reader request and the reader grant in the protocol so that processing can be done after a request has been received and before sending a grant. The internal symbols *z1* and *u1* are placed between the reader done and the reader acknowledgement in the protocol so that processing can be performed after a done has been received and before replying with an acknowledgement. The resulting communication protocol is

$$\textbf{pref } *[rr1?; x1; y1; rg1!; rd1?; z1; u1; ra1!]$$

Similar reasoning holds for the communication protocol with child 2, and the protocol is

$$\textbf{pref } *[rr2?; x2; y2; rg2!; rd2?; z2; u2; ra2!]$$

A reader module has three internal states: states *S0*, *S1* and *S2*. A reader module is initially at state *S0* when none of its children is active (i.e., accessing the critical sections). It is at state *S1* when only one child is active, and it is at state *S2* when both children are active. The following communication protocol shows the internal processing of a reader module at the three states.

$$
\begin{array}{lllll}
1: & |[& \text{states} & S0, S1, S2 :: \\
2: & & \text{initially} & S0 : \\
3: & & S0 = \textbf{pref} & ((x1|x2); sr!; sg?; (y1|y2); S1) \\
4: & & S1 = \textbf{pref} & ((x1|x2); (y1|y2); S2 \\
5: & & & |(z1|z2); sd!; sa?; (u1|u2); S0 \\
6: & & &) \\
7: & & S2 = \textbf{pref} & (s2; (z1|z2); (u1|u2); S1) \\
8: &]| & &
\end{array}
$$

Please note that the numbers on the left of the protocol are not part of the specification. They are line numbers used to assist in the explanation. The above protocol specifies that a reader module begins at state *S0* initially and implements the proper state transitions. For example, assume that a reader module is at state *S0* when the specified events for that state have occurred, the module changes its state to *S1* as specified by the last symbol on line 3.

When a reader module is at state *S0*, it may receive a request for accessing the common "file". The module propagates the event to its parent by sending *sr!* and waits for *sg?*. After receiving the grant from its parent, the module allows the sending of a grant to the appropriate child (i.e., initiated by *y1* or *y2*), and the module changes its state to *S1*. This is shown on line 3 of the protocol.

When a reader module is at state *S1* and receives a request from a child for entering its critical section, the module outputs a grant to the child without any further processing. The module then changes its state to *S2*. This is specified on line 4 of the protocol. If a reader module receives a done from an active child, the module passes the done event *sd!* to its parent and waits for the acknowledgement *sa?* from its parent. The module then sends an acknowledgement to the exiting child, and the module changes its state back to *S0*. This is specified on line 5 of the protocol.

When a reader module is at state *S2*, it can only receive done signals from its children. When it receives a done signal, it does not do any processing and allows the signal of an acknowledgement to the appropriate child. It then changes state to *S1*. This is specified in line 7.

Like the reader module, a writer module also has three sets of terminals: one for communicating with its parent and one for communicating with each child. The communication protocols between a writer module and its children are the same as the communication protocols between a reader module and its children. When a child requests to enter its critical section, the writer module performs some processing before sending a grant. When a child wants to leave its critical section, the writer module performs some processing before sending an acknowledgement. The communication protocols between a writer module and its two children are

$$\mathbf{pref}\ *[wr1?; x1; y1; wg1!; wd1?; z1; u1; wa1!]$$

and

$$\mathbf{pref}\ *[wr2?; x2; y2; wg2!; wd2?; z2; u2; wa2!]$$

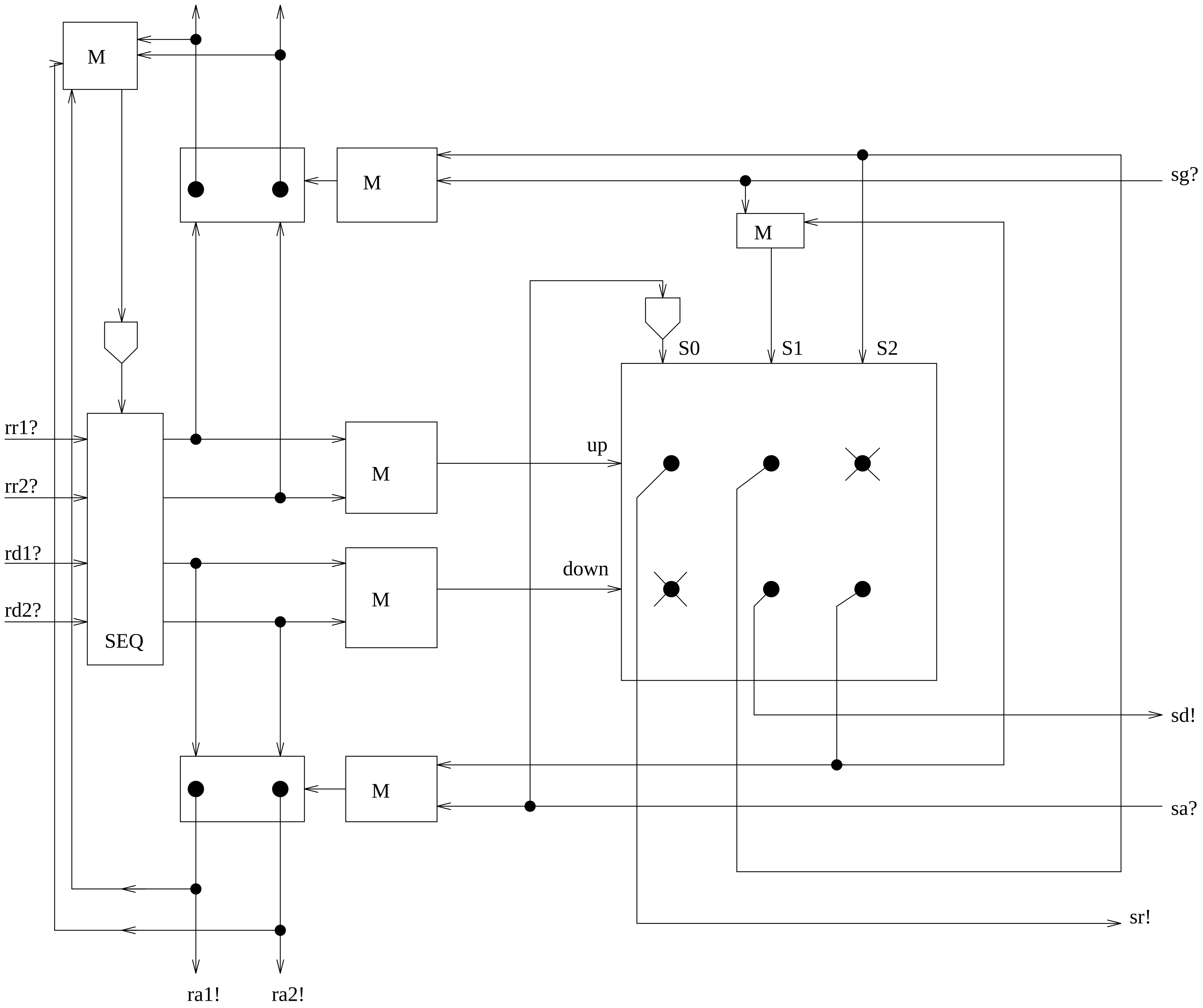
A writer module is simpler than a reader module in that it does not have to keep track of the number of active children it has because at most one child is active. When a writer module receives a request for accessing the common "file", it passes the request to its parent by sending *tr!*. When the module receives a *tg?* from its parent, it then sends a grant to the appropriate child. When the active child requests to exit its critical section, the writer module sends *td!* to its parent and waits until a *ta?* is received before allowing an acknowledgment to be sent to the appropriate child. The communication protocol between a writer module and its parent is

$$\mathbf{pref}\ *[(x1|x2); tr!; tg?; (y1|y2); (z1|z2); td!; ta?; (u1|u2)]$$

The specification of the writer module is just the weave of the above three protocols.

$$\begin{array}{ll}
|[& x1, x2, y1, y2, z1, z2, u1, u2 :: \\
 & \mathbf{pref}\ *[wr1?; x1; y1; wg1!; wd1?; z1; u1; wa1!] \\
|| & \mathbf{pref}\ *[wr2?; x2; y2; wg2!; wd2?; z2; u2; wa2!] \\
|| & \mathbf{pref}\ *[(x1|x2); tr!; tg?; (y1|y2); (z1|z2); td!; ta?; (u1|u2)] \\
]| &
\end{array}$$

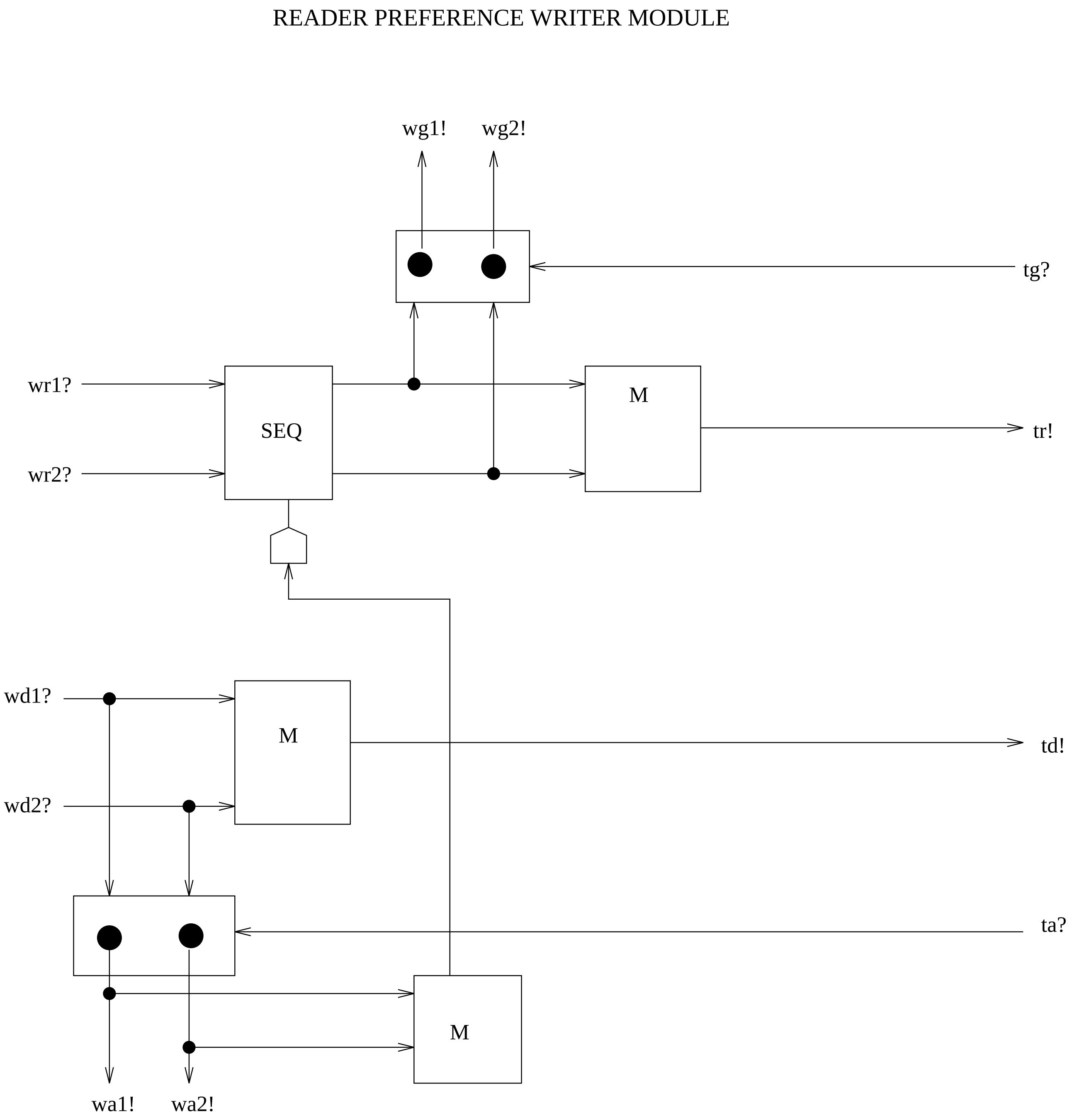
READER PREFERENCE READER MODULE



The above diagram is the decomposition diagram of a readers-preference reader module.

Reader requests and dones are required to pass through a SEQUENCER in order for the module to process requests for entering and exiting of critical section one at a time. After passing through the SEQUENCER, each reader request forks into two actions. One goes into a 2×1 JOIN, and another goes into a MERGE. The JOIN is used to send a grant to the appropriate child when triggered. Similarly, each reader done also forks into two actions. One goes into a 2×1 JOIN, and another goes into a MERGE. The JOIN is used to send an acknowledgement to the appropriate child when triggered. When replying with a grant or an acknowledgement, the action is forked into two. One goes to the appropriate child, and another goes through a MERGE which eventually reaches the SEQUENCER; thereby, triggering the SEQUENCER to process the next event. All the reader requests eventually merge into one action, *up*. All the reader dones merge into another action, *down*. A reader module has three states: *S0*, *S1* and *S2*. A 2×3 JOIN is used to implement the state machine. When the state machine is at a certain state, certain events are expected . For example, the module is initially at state *S0*, as indicated by the IWIRE, event *S0* occurs initially. If the module is at state *S0* when an *up* occurs (i.e., a reader request occurs), *sr!* then occurs which is passed to the module's parent. The module then waits until its parent replies with an *sg?* which causes a grant to send to the appropriate child, and the state machine changes its state to *S1*. If an *up* occurs when the module is at state *S1*, then *S2* occurs and a grant is sent to the appropriate child. If a *down* event occurs when the module is at state *S1*, *sd!* is sent to the module's parent. The module then waits for its parent to reply with an *sa?* which causes the module to change its state back to *S0* and an acknowledgement to be sent to the appropriate child. If a *down* occurs when the module is at state *S2*, the module changes its state to *S1*, and an acknowledgement is sent to the appropriate child.

The following is a decomposition diagram of a readers-preference writer module.



Because writer requests can come in parallel, a SEQUENCER is used to allow only one request to be processed at a time. Like the reader module, each request is forked into two events. One goes into a 2×1 JOIN, and another goes into a MERGE. The JOIN is used to send a grant to the requesting child when triggered. The merged event is passed to the module's parent as a *tr!* event. The module then waits until its parent replies with a *tg?* which causes a grant to be sent from the JOIN. When the active child wants to leave its critical section, it would send a done to the module. Because only one done event can occur, no arbitration is necessary. Dones are forked into two with one goes to the module's parent as a *td!* event, and another goes into a 2×1 JOIN which sends an acknowledgement to the exiting child when the parent sends a *ta?*.

6 Solutions to the Writers-Preference Version

6.1 Tree-Structure Solution

Unlike the tree-structure readers-preference solution, the tree structure writers-preference solution requires the use of an up/down counter. A scheduler for an m-readers and n-writers problem requires an up/down k counter where $k \geq m$. The writer preference solution is not as modular and expandable as the reader preference solution. It is divided into two parts: the arbiter and the two subtrees. The arbiter contains the up/down counter. Given an m-readers and n-writers scheduler, a $m+1$-readers and $n+1$-writers scheduler can be constructed by adding a reader module to the reader subtree, a writer module to the writer subtree and replacing the arbiter with another arbiter that uses an up/down k counter where $k \geq m+1$. If the up/down counter in the original arbiter is at least $m+1$, then it is not necessary to replace the arbiter. Like the tree-structure readers-preference solution, all the modules in the reader subtree are identical, and all the modules in the writer subtree are also identical. Reader processes communicate with the scheduler through the leaves in the reader subtree, and writer processes communicate with the scheduler through the leaves in the writer subtree. All reader requests for accessing the common "file" are propagated up from the leaves of the reader subtree to the root of the readers and writers tree. The arbiter keeps track of the number of readers in their critical sections. After receiving each reader request, the arbiter increments its internal count by 1. All the reader dones are also propagated up to the arbiter. After receiving each reader done, the arbiter decrements its internal counter by 1. The arbiter is responsible for deciding which process is allowed to access the common "file". The arbiter only allows a writer to modify the "file" when no reader is accessing it. Readers are only allowed to examine the "file" when no writer is waiting to access the "file". When a reader module receives a request or a done from its child, it passes the event to its parent. After passing an event to its parent, the reader module does not have to wait for grant or acknowledgement from its parent before passing additional request or done to its parent. When the module receives a grant or an acknowledgement from its parent, the module passes the event to the appropriate child. A writer module has three states: *S0* when no child is active, *S1* when one child is active and *S2* when one child is active and one is waiting to access the "file". If a writer module receives a request when it is in state *S0*, the request is propagated up to its parent. The module waits until a grant is received which is then passed to the requesting child. If a request is received when the module is in state *S1*, the module changes its state to *S2* and does not passed the request to its parent. The module waits until the active child sends a done to it. The module acknowledges the done action right away and then sends a grant to the waiting child. The module then changes its state back to *S1*. If a done is received when the module is in state *S1*, the module propagates the event to its parent and changes its state to *S0* when an acknowledgement is received from its parent. The module also passes the acknowledgement to the exiting child.

The following is the specification of the writers-preference reader module.

1 : $|[\quad x0, x1, y0, y1, z0, z1, u0, u1 ::$
2 : $\quad\quad \textbf{pref} \; *[rr1?; x0; y0; rg1!]$
3 : $\| \quad \textbf{pref} \; *[rd1?; z0; u0; ra1!]$
4 : $\| \quad \textbf{pref} \; *[rr2?; x1; y1; rg2!]$
5 : $\| \quad \textbf{pref} \; *[rd2?; z1; u1; ra2!]$
6 : $\| \quad \textbf{pref} \; *[(x0|x1); sr!; sg?; (y0|y1)]$
7 : $\| \quad \textbf{pref} \; *[(z0|z1); sd!; sa?; (u0|u1)]$
8 : $]|$

A reader module has three sets of terminals: one for communicating with each child and one for communicating with its parent. With respect to child 1, the communication protocol is

$$\textbf{pref} \; *[rr1?; rg1!; rd1?; ra1!]$$

After child 1 sends a request to the module, the child expects a grant to be sent from the module. It can then enter its critical section. When the child wants to leave its critical section, it sends a done to the module and waits for an acknowledgement. After a writer module receives a request, it performs some processing before sending a grant to the requesting child. Internal symbols are used to ensure that processing begins after getting a request and ends before sending a grant. Internal symbols are placed between a done event and an acknowledgement event for the same reason. After receiving a request, the module passes the event to its parent. The module sends a grant to the waiting child only if an *sg?* occurs (line 6). After a done is received, the module passes the event to its parent. An acknowledgement is sent to the waiting child only if the module receives an *sa?* from its parent (line 7). The protocols in line 6 and line 7 are weaved together instead of concatenated together because requests and dones from both children are processes independently of each other.

The following is the specification of the writers-preference writer module.

```
 1:  |[  u0, u1, x0, x1, y0, y1, z0, z1 ::
 2:      pref *[wr1?; x0; y0; wg1!; wd1?; z0; u0; wa1!]
 3:  ||  pref *[wr2?; x1; y1; wg2!; wd2?; z1; u1; wa2!]
 4:  ||
 5:        |[      states   S0, S1, S2 ::
 6:              initially  S0 :
 7:            S0 = pref    ((x0|x1); tr!; tg?; (y0|y1); S1)
 8:            S1 = pref    ((x0|x1); S2
 9:                         |(z0|z1); td!; ta?; (u0|u1); S0
10:                         )
11:            S2 = pref    ((z0|z1); (u0|u1); (y0|y1); S1)
12:        ]|
13:  ]|
```

Like the reader module, the writer module also uses RGDA protocol with synchronizing internal symbols to communicate with its children. A writer module has three states: *S0* when no child is active, *S1* when one child is active and *S2* when one child is active and one is waiting to be activated. If a request is received when the module is in state *S0*, *tr!* is passed to its parent. The module waits until a *tg?* is received before sending a grant to the waiting child and changing its internal state to *S1*. If a request is received when the module is in state *S1*, the module changes its state to *S2*. If the active child sends a done to the module when its state is in *S1*, the module passes *td!* to its parent and waits for a *ta?* before sending an acknowledgement to the exiting child and changing its internal state back to *S0*. If a done is received when the module is in state *S2*, an acknowledgement is sent back to the exiting child without passing the done action back to the module's parent, and a grant is sent to the waiting child. Lines 5 to 12 implements the state machine.

The following is the specification of the root of the readers and writers tree (arbiter).

```
1 :  |[  x0, x1, x2, y0, z ::
2 :      pref ∗[tr?; z; x0; tg!; td?; y0; ta!]
3 :  ||  pref ∗[sr?; x1; sg!]
4 :  ||  pref ∗[sd?; x2; sa!]
5 :  ||
6 :        |[  var         i : int ::
7 :            initially   i := 0
8 :            if          i = 0 then (x1; i := i + 1 | z; x0; y0)
9 :            |           i > 0 then
10 :                        ( x1; i := i + 1
11 :                        | x2; i := i − 1
12 :                        | z;  do i > 0 then x2; i := i − 1 od ; x0; y0
13 :                        )
14 :           fi
15 :       ]|
16 :  ]|
```

The arbiter has two sets of terminals: one for communicating with the reader subtree and another one for communicating with the writer subtree. The communication protocol between the arbiter and the writer subtree is

$$\textbf{pref} \; *[tr?; z; x0; tg!; td?; y0; ta!]$$

Synchronizing internal symbols (i.e., *x0* and *y0*) are placed between writer-subtree request and grant and between writer subtree done and acknowledgement. These internal symbols are used to ensure no grant is sent to the reader subtree while a writer is active. The synchronizing internal symbol, z, is used to ensure no grant is sent to the reader subtree after a request is received from the writer subtree. The communication protocol with the reader subtree is

$$\begin{array}{ll} & \textbf{pref} \; *[sr?; x1; sg!] \\ || & \textbf{pref} \; *[sd?; x2; sa!] \end{array}$$

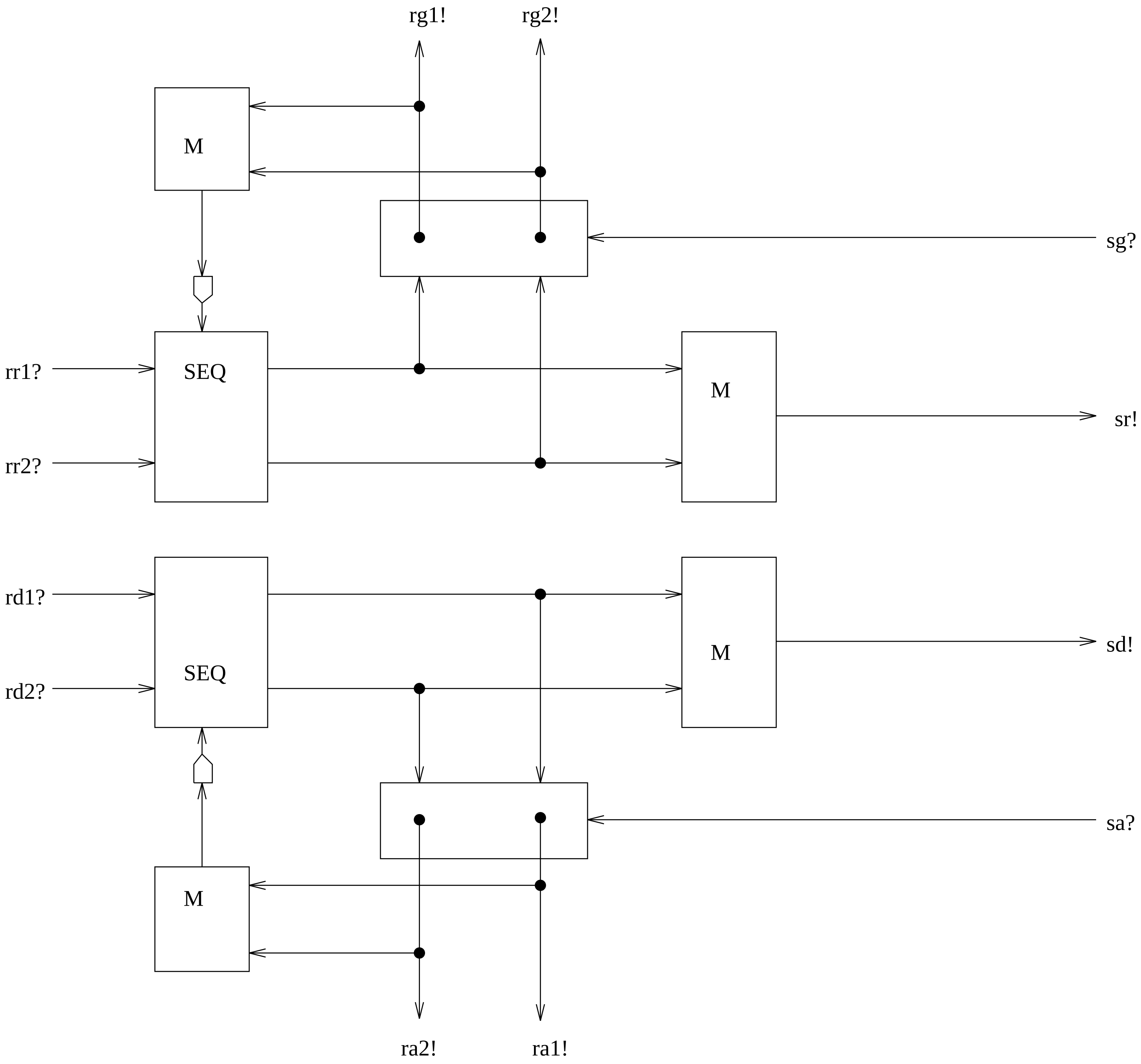
The two subprotocols are weaved together, instead of concatenated together, because requests and dones can come in parallel from the reader subtree. The arbiter contains an up/down k counter where k is at least equal to the total number of reader processes. When a request is received from the reader subtree, the counter is incremented by 1. The internal symbol *x1* is used to ensure the counter is incremented. When a done is received from the reader subtree, the counter is decremented by 1. The internal symbol *x2* is used to ensure the counter is decremented.

Lines 6 to 15 specifies the up/down k counter. The variable i contains the value of the internal count. If $i \geq 0$ and a reader subtree request is received (i.e., synchronization with *x1* occurs), the variable is incremented by 1. If $i > 0$ and a reader subtree done is received (i.e., synchronization with *x2* occurs), the variable is decremented by 1. If $i > 0$ and a writer-subtree request is received (i.e., synchronization with z occurs), no further reader-subtree grants can be given, and the variable

i can only be decremented. Once the variable i becomes 0 (i.e., no reader is active), synchronization with $x0$ and $y0$ can occur which allows a writer to become active.

The diagram on the next page is the decomposition diagram of the writers-preference reader module.

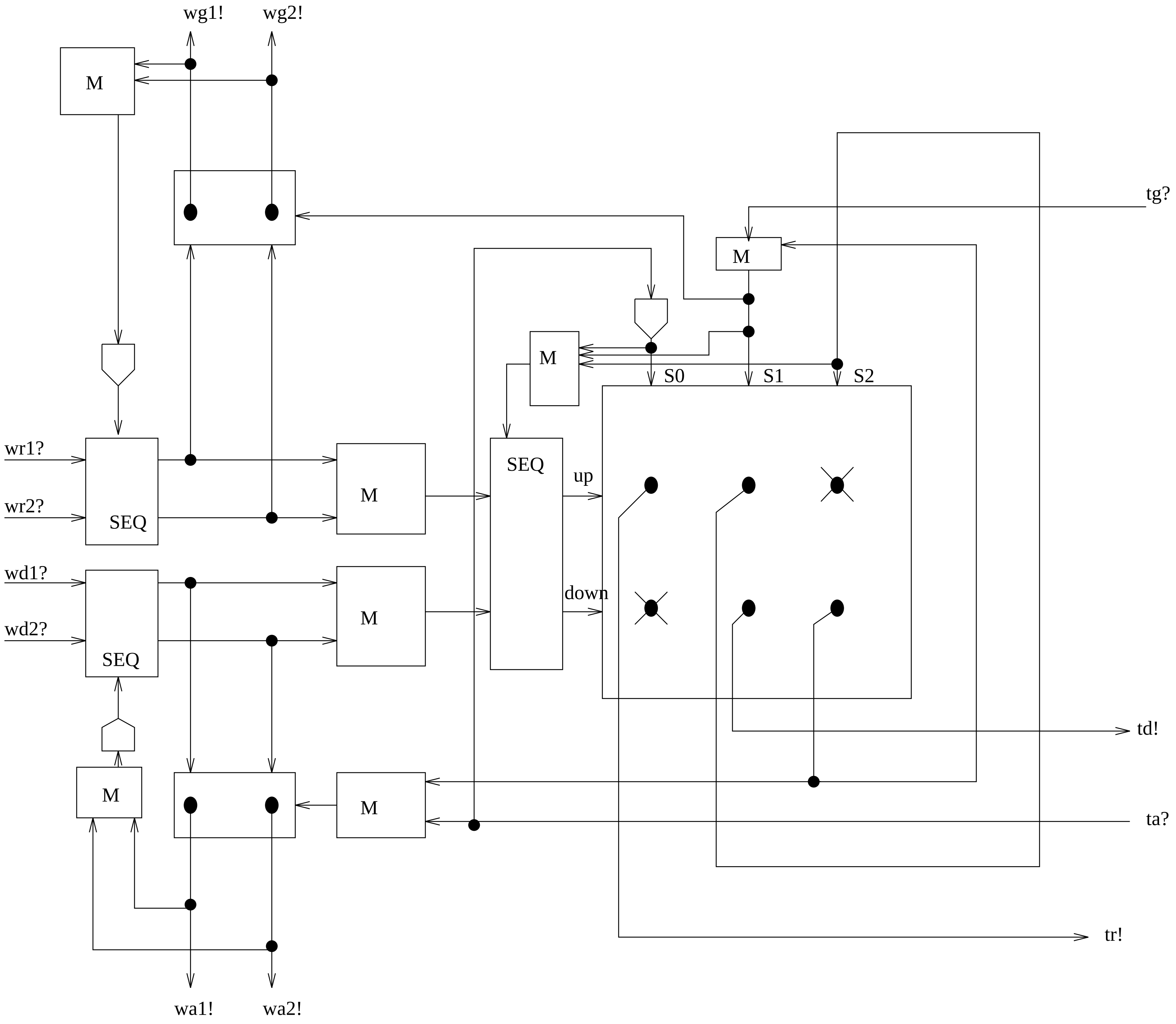
WRITER PREFERENCE READER MODULE



A reader module is divided into two independent halves with identical structure: one for handling requests and one for handling dones. Because reader requests can come in parallel, a SEQUENCER is used. After passing through the SEQUENCER, a request is forked into two events, one goes to the parent of the module as *sr!*, and the other one goes into a 2×1 JOIN to be triggered when granting the request. When the parent of the module grants the request, an *sg?* is passed back. This event triggers a grant to be sent to the requesting child. The action also triggers the SEQUENCER to allow the next request to be processed. The structure for processing dones and acknowledgements is the same as processing for requests and grants. When a done is received, *sd!* is sent to the module's parent, and when the parent returns an *sa?*, an acknowledgement is sent to the exiting child.

The diagram on the next page is the decomposition diagram of the writers-preference writer module.

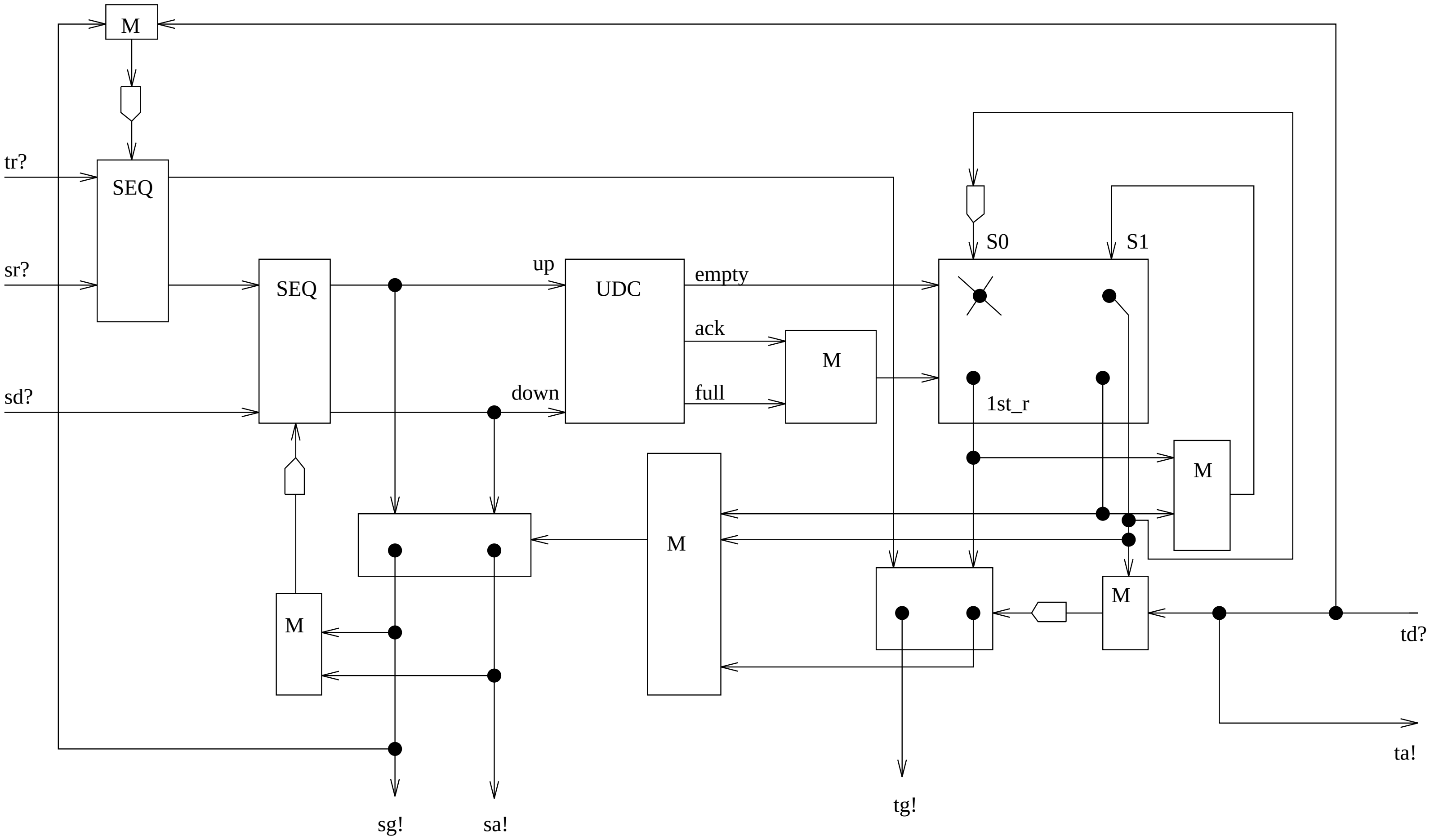
WRITER PREFERENCE WRITER MODULE



A SEQUENCER is used to mutually exclude writer requests, because writer requests can arrive in parallel. Once passed the SEQUENCER , a request is forked into two events. One goes into a 2×1 JOIN to be used to send a grant to the requesting writer when triggered, and the other one goes into another SEQUENCER which is used to mutually exclude a request and a done. Similarly, done signals are mutually excluded by a SEQUENCER. A done is then forked into two: one goes into a JOIN to be used to send an acknowledgement to the exiting child when triggered and another one goes into the SEQUENCER used to mutually exclude a request and a done. Once a request or a done passed through the SEQUENCER, the event goes into a 2×3 JOIN. A request that has passed through the SEQUENCER is labelled as *up*, and a done is labelled as *down*. The 2×3 JOIN is used to implement a state machine with 3 states. The state machine is initially at state *S0*. If an *up* is received, the state is changed to *S1*, and a grant is sent to the requesting child. If the state is at *S1*, which means one child is active, when an *up* is received, the state is changed to *S2*, and the module waits for a done to be received. If a *down* is received, which means the active child is exiting, when the module is at state *S2*, the state is changed to *S1*, a grant is sent to the waiting child, and an acknowledgement is sent to the exiting child. All the SEQUENCERs are then triggered to allow the processing of the next input. If the state is at *S1* when a *down* is received, *td!* is sent to the parent of the writer module. The module waits until a *ta?* is received from its parent. The module changes its state back to *S0*, and an acknowledgement is sent to the exiting child. The SEQUENCER that is responsible for mutually excluding done events is allowed to process the next done.

The diagram on the next page is the decomposition diagram of the arbiter (i.e., the root of the readers and writers tree).

WRITER PREFERENCE ARBITER



Reader-subtree request and writer-subtree request can arrive at the arbiter in parallel. A SEQUENCER is used to mutually exclude the two requests. A reader-subtree done can also come in parallel with a reader request. A different SEQUENCER is used to mutually exclude the two events. Two SEQUENCERs are used instead of one, because a done event can still be processed even when the SEQUENCER used to mutually exclude reader and writer-subtree requests is not able to allow events to pass through. An arbiter for the m-readers and n-writers problem contains an up/down k counter where $k \geq m$. After a reader-subtree request passed through the two SEQUENCERs, it increments the counter. After a reader-subtree done passed through the SEQUENCER, it decrements the counter. Part of the implementation contains two states: *S0* when the counter is 0 or *S1* when the counter is not 0. When the counter is 0, it means no reader is in its critical section. When the counter is not 0, there is at least one reader in its critical section. If the counter is incremented (i.e., either *ack* or *full* is outputted from the counter) when the arbiter is in state *S0*, the state machine changes its state to *S1*, and a grant is sent to the reader subtree. If the counter is incremented when the state machine is in state *S1*, the state remains in *S1*, and a grant is sent to the reader subtree. If the counter is decremented and an *ack* is sent, the state machine remains in state *S1*, and an acknowledgement is sent to the reader subtree. If the counter is decremented and an *empty* is sent, the state machine changes state from *S1* to *S0*. Besides changing the state, if the counter is incremented when the arbiter is in state *S0*, an action is sent from the terminal *1st_r*. The action passes through a 2×1 JOIN and eventually becomes an *sg!*. No signal can be sent out from the JOIN until the up/down counter is 0. When a writer-subtree request arrives at the arbiter, it passes through the SEQUENCER which mutually exclude reader-subtree request and writer subtree request. The event then reaches the 2×1 JOIN. When the up/down counter becomes 0, a writer subtree grant is sent from the JOIN. No further request can be processed by the arbiter until a done is sent from the writer subtree.

7 Further Research

In this report, a tree-structure readers-preference scheduler and a tree-structure writers-preference scheduler for solving the readers and writers problem have been provided. However, there are still many improvements and areas for exploration. A more modular tree-structure writers-preference solution is needed. A tree-structure equal-preference solution would be useful, because no starvation of readers or writers can occur. It would also be useful to grant requests and acknowledge dones in FIFO order, because no starvation of individual processes would occur.

8 Conclusion

In this essay, a specification of an m-readers and n-writers scheduler in a CSP-like notation has been provided. A tree-structure solution to the readers-preference version of the problem using delay-insensitive circuits has been designed. Its major advantage is that it is modular and expandable. A tree-structure solution to the writers-preference version of the problem using delay-insensitive circuits have also been provided. It is not as modular as the readers-preference version. The tree-structure writers-preference scheduler can be as modular as the tree-structure readers-preference scheduler if the total number of readers is not as greater than the maximum count of the up/down counter.

References

[1] Gregory R. Andrews. *Concurrent Programming: principles and practice.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[2] Tom Axford. *Concurrent Programming: fundamental techniques for real-time and parallel Software Design.* John Wiley & Sons Ltd., 1989.

[3] Jo C. Ebergen. Cs798a lecture notes: Parallel program and asynchronous circuit design. Winter of '93 at the University of Waterloo.

[4] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 3(5):107–119, 1991.

[5] Jo C. Ebergen and Sylvain Gingras. A verifier for network decompositions of command-based specifications. In *Proc. 1993 Hawaii International Conference on Systems Sciences*, 1993.

[6] F. Heymans P.J. Courtois and D.L. Parnas. Concurrent control with 'readers' and 'writers'. *Communications of the ACM*, 14(10):667–668, 1971.

[7] Dick Pountain. Computing without clocks. *Byte*, pages 145–150, 1 1993.

[8] M. Raynal. *Algorithms for Mutual Exclusion.* The MIT Press, 1986.

[9] John P. Segers. The design and analysis of asynchronous up-down counters. Master's thesis, Eindhoven University of Technology, 1993.

[10] A. Silberschatz, J. Peterson, and P. Galvin. *Operating System Concepts.* Addison-Wesley Publishing Company, 1991.