Modular Multiprogramming*

Jayadev MISRA†
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
(512) 471-9547
email: misra@cs.utexas.edu
Home Page: http://www.cs.utexas.edu/users/misra

October 21, 1998

Abstract

Object-based sequential programming has had a major impact on software engineering. However, object-based concurrent programming remains elusive as an effective programming tool. The class of applications that will be implemented on future high-bandwidth networks of processors will be significantly more ambitious than the current applications (which are mostly involved with transmissions of digital data and images), and object-based concurrent programming has the potential to simplify designs of such applications. This paper shows that many of the programming concepts developed for databases, object-oriented programming and designs of reactive systems can be unified into a compact model of concurrent programs that can serve as the foundation for designing these future applications.

1 Introduction

Object-based sequential programming has had a major impact on software engineering. However, object-based concurrent programming re-

*This monograph is available at ftp://ftp.cs.utexas.edu/pub/psp/seuss/discipline.ps.Z

†This material is based in part upon work supported by the National Science Foundation Awards CCR–9803842, CCR–9707056 and CCR-9504190.

mains elusive as an effective programming tool. The class of applications that will be implemented on future high-bandwidth networks of processors will be significantly more ambitious than the current applications (which are mostly involved with transmissions of digital data and images), and object-based concurrent programming has the potential to simplify designs of such applications. Many of the programming concepts developed for databases, object-oriented programming and designs of reactive systems can be unified into a compact model of concurrent programs that can serve as the foundation for designing these future applications.

Research in multiprogramming has, traditionally, attempted to reconcile two apparently contradictory goals: (1) it should be possible to understand a module (e.g., a process or a data object) in isolation, without considerations of interference by the other modules, and (2) it should be possible to implement concurrent threads at a fine level of granularity so that no process is ever locked out of accessing common data for long periods of time. The goals are in conflict because fine granularity, in general, implies considerable interference. The earliest multiprograms (see, for instance, the solution to the mutual exclusion problem in Dijkstra [8]) were trivially small and impossibly difficult to understand, because the behaviors of the individual processes could not be understood in isolation, and all possible interactions among the processes had to be analyzed explicitly. Since then, much effort has gone into limiting or even eliminating interference among processes by employing a variety of synchronization mechanisms: locks or semaphores, critical regions, monitors and message communications.

Constraining the programming model to a specific protocol (binary semaphores or message communication over bounded channels, for instance) will prove to be short-sighted in designing complex applications. More general mechanisms for interactions among modules, that include these specific protocols, are required. Further, for the distributed applications of the future, it is essential to devise a model in which the distinction between computation and communication is removed; in particular, the methods for designing and reasoning about the interfaces should be no different from those employed for the computations at the nodes of the network.

Wide-Area Computing

Wide-Area computing is carried out over a set of asynchronously communicating machines. Machines interact by sending messages to each other. Often, the applications that run on this platform – interactive query over a replicated database, for instance – are coded using message communication as a basic primitive. We believe that applications should be coded at a higher level of abstraction. Remote Procedure Call (RPC) is one mechanism that goes beyond message communication. The concurrent-object model that we propose in this document is even more divorced from the implementation details. The query processing application, for instance, will be coded by invoking procedures that operate on segments of the database, and using processes that guarantee consistency among the replicas. Clearly, the procedure calls have to be implemented using message communications, but that

should not be a concern at the level of application programming.

There are several key questions in developing a model of concurrent-objects for wide-area computing. Foremost among the concerns is the difficulty associated with managing concurrent interactions, particularly when a multitude of machines interact. This is the main topic of this research. Additionally, issues of security and fault-tolerance are central to wide-area computing. How can a secure computation be carried out if the data and procedure reside at different machines? If a computation involves several thousand machines – as would be expected for a computation on the world-wide web – is it realistic to allow an unknown party to have exclusive access to a resource; that party may crash resulting in a major roll-back effort? Is it reasonable to queue a caller for access to a resource when the resource manager could possibly fail, causing the caller to block? In this case, it may be more effeicient to reject the call if the resource is not readily available. We seek a model that allows us to experiment with a variety of questions of this nature.

Overview of the Proposed Solution

We have developed a model of multiprogramming, called **Seuss**. One of the major goals of Seuss is to simplify multiprogramming; Seuss fosters a discipline of programming that makes it possible to understand a program execution as a single thread of control, yet it permits program implementation through multiple threads. As a consequence, it is possible to reason about the properties of a program from its single execution thread, whereas an implementation on a specific platform (e.g., shared memory or message communicating system) may exploit the inherent concurrency appropriately. A central theorem establishes that multiple execution threads implement single execution threads, i.e., for any interleaved execution of some actions there exists a non-interleaved execution of those actions that establishes an identical final state starting from the same initial state.

The programming system is built around a conceptually simple mathematical system. Accordingly, all well-known constructs of concurrent programming – process, message communication, synchronization, sharing and mutual exclusion – were eliminated. Traditional multiprogramming concepts such as, locking, rendezvous, waiting, interference and deadlock, do not appear as basic concepts in our model. No specific communication or synchronization mechanism, except procedure call, is built into this model. Yet, typical multiprograms employing message passing over bounded or unbounded channels can be encoded in Seuss by declaring the processes and channels as the components of a program; similarly, shared memory multiprograms can be encoded by having processes and memories as components. Seuss permits a mixture of either style of programming, and a variety of different interaction mechanisms – semaphore, critical region, 4-phase handshake, etc. – can be encoded as components.

Seuss proposes a complete disentanglement of the sequential and concurrent aspects of programming. We expect large sections of code to be written, understood and reasoned-about as sequential programs. *We view multiprogramming as a way to orchestrate the executions of*

these sequential programs, by specifying the conditions under which each program is to be executed. We propose an efficient implementation scheme that can, under user directives, interleave the individual sequential programs with fine granularity without causing any interference.

2 Seuss Syntax

In this section, we introduce a publication notation for writing programs. The notation is intended for implementation on top of a variety of host languages. Therefore, no commitment has been made to the syntax of any particular language (there are different implementations with C++ and Java as host languages) and syntactic aspects that are unrelated to the model are left unspecified in the publication notation.

The notation is described using BNF. All non-terminal identifiers are in Roman and all terminal identifiers are in boldface type. The traditional meta symbols of BNF – ::= { } [ ] ( ) – are used, along with $\vee$ to stand for alternation (the usual symbol for alternation, "|", is a terminal symbol in our notation). The special symbols used as terminals are, | // ; : :: in the syntax given below. A syntactic unit enclosed within "{" and "}" in a production may be instantiated zero or more times, and a unit within "[" and "]" may be instantiated zero or one time. In the right hand side of a production, $(p \vee q)$ denotes that a choice is to be made between the syntactic units p and q in instantiating this production. We omit the parentheses, "(" and ")", when no confusion can arise. Text enclosed within " { " and " } " in a program is to be treated as a comment.

2.1 Program

The Seuss programming model is sparse: a program consists of *cat*s (*cat* is short for *category*) and *box*es. A cat is similar to a process/class/monitor type; a box is an instance of a cat. A cat/box has a local state and it includes procedures by which its local state can be accessed and updated. Procedures in a box may call upon procedures of other boxes.

program ::= **program** program-name {cat $\vee$ box} **end**
cat ::= **cat** cat-name [parameters]: {variable} {procedure} **end**
box ::= **box** box-name [parameters]: cat-name

A program consists of a set of cats and boxes in any order. The declaration of a cat or box includes its name and, possibly, parameters. The names of programs, cats and boxes are identifiers. The parameters of a cat or box could be ordinary variables, cats or boxes. A cat consists of (zero or more) variable declarations followed by procedure declarations. A box is an instance of a cat. Variables are declared and initialized in a cat as in traditional programming languages.

2.2 procedure

A cat includes procedures that may be called from procedures of other cats. There are two kinds of procedures, *total* and *partial*. A total

procedure models computations that never wait (for an unbounded amount of time) to interact with their environments; a computation that may wait, possibly forever, for such an interaction is modeled by a partial procedure. Thus, a P operation on a semaphore (that may possibly never complete) is coded as a partial procedure. Our programming model does not include *waiting* as a fundamental concept; therefore, a partial procedure does not wait, but it *rejects* the call. We have a more detailed discussion of the distinction between total and partial procedures in section 4.1.

procedure ::= partial-procedure $\vee$ total-procedure
partial-procedure ::= **partial** partial-method $\vee$ partial-action
total-procedure ::= **total** total-method $\vee$ total-action
partial-method ::= **method** head :: partial-body
partial-action ::= **action** [head] :: partial-body
total-method ::= **method** head :: total-body
total-action ::= **action** [head] :: total-body

A procedure is either **partial** or **total**; also, a procedure is either a **method** or an **action**. Thus, there are four possible headings identifying each procedure. Each procedure has a head and a body; the head is optional for actions. The procedure head is similar to the form used in typical imperative languages; it has a procedure name followed by a list of formal parameters and their types. The procedure body has different forms for partial and total procedures. For this manuscript, we take a total-body to be any sequential program. The partial-body is defined by:

partial-body ::= alternative {(| alternative) $\vee$ ($\not|$ alternative)}
alternative ::= precondition [; preprocedure] $\rightarrow$ total-body
preprocedure ::= partial-method-call

The body of a partial procedure consists of one or more alternatives. Each alternative is *positive* or *negative*: the first alternative is positive; an alternative preceded by | is positive and one preceded by $\not|$ is negative. Each alternative has a precondition, an optional preprocedure and a total-body. A precondition is a predicate on the state of the box to which this procedure belongs (i.e., it is constrained to name only the variables of the cat in which the procedure appears). For each partial procedure at most one of its alternatives holds in any state, i.e., the preconditions in the alternatives of a partial procedure are pairwise disjoint. A preprocedure is a call upon a partial method (in some other box).

2.3 Example of Seuss Syntax

The following example illustrates Seuss syntax. A ubiquitous concept in multiprogramming is a semaphore. The program given below includes a definition of *Semaphore* as a cat and two instances of *Semaphore*, s and t. Cat *user* describes a group of users that execute their critical sections only if they hold both semaphores, s, t; there are three instances of *user*. Each user releases both semaphores upon

completion of its critical section. In *user*, boolean variables *hs* and *ht* are *true* only when the *user* holds the semaphores s and t, respectively. We explain the operational semantics of Seuss in section 3.

Notational Convention We write **box** s, t : *Semaphore* as a shorthand for **box** s : *Semaphore* and **box** t : *Semaphore*.

program *MutualExclusion*
cat *Semaphore*
  var n: nat **init** 1 {initially, the semaphore value is 1}
  partial method P:: $n > 0 \rightarrow n := n - 1$
  total method V:: $n := n + 1$
end {*Semaphore*}

box s, t : *Semaphore*

cat *user*
  var hs, ht: boolean **init** *false*
  partial action $s.acquire$:: $\neg hs;\ s.P\ \rightarrow hs := true$
  partial action $t.acquire$:: $\neg ht;\ t.P\ \rightarrow ht := true$
  partial action *execute*::
    $hs \wedge ht \rightarrow$ *critical section*; $s.V$; $t.V$; $hs := false$; $ht := false$
end {*user*}

box u, v, w : *user*
end {*MutualExclusion*}

The partial actions $s.acquire$ and $t.acquire$ in *user* include calls upon the partial methods $s.P$ and $t.P$ as preprocedures; they also include calls upon the total methods $s.V$ and $t.V$ in their bodies. The partial action P in *Semaphore* has no preprocedure. Each partial procedure in this example has exactly one (positive) alternative.

2.4 Constraints on Programs

Procedure Call A total-body can include a call only to a total method; a partial method cannot be called by a total body. A partial method can only be called as a preprocedure of an alternative of a partial procedure. The syntax specifies that an alternative can have at most one preprocedure. In the example of section 2.3, partial action $s.acquire$ calls $s.P$ as a preprocedure, and *execute* calls the total methods $s.V, t.V$ in its total body (i.e., the code following $\rightarrow$).

Partial Order on Boxes Every procedure p imposes a partial order $\geq_p$ over the boxes; during the execution of p a procedure of box b can call a procedure of box b' provided $b >_p b'$ (i.e., $b \geq_p b' \wedge b \neq b'$). Thus, calls are made from the procedures of a higher box to that of a lower box. In the example of section 2.3, *user* boxes (u, v, w) call upon *Semaphore* boxes (s, t), but not conversely.

In most cases, it is possible to have a single partial order over boxes that is obeyed by all procedure calls, as is the case in the example of section 2.3. There are important exceptions, however, for which we

allow different procedures to impose different partial orders over boxes. Consider a *sleeper* who sets an alarm clock (*AlarmClock*) to ring at a specified time, and the alarm clock notifies the *sleeper* by ringing at the specified time. Let the *sleeper* and the *AlarmClock* be coded as boxes; an action *set* in *sleeper* calls a procedure in *AlarmClock* (to set the clock), and an action *WakeUp* in *AlarmClock* calls a procedure in *sleeper* (to wake up the sleeper at the specified time). Then, there is no fixed order of these two boxes: *set* orders *sleeper* higher than *AlarmClock*, and *WakeUp* orders them in the opposite manner.

A consequence of the requirement of partial order is that no procedure from a box is called if some procedure of that box is executing; therefore, at most one procedure from any box is executing at any moment.

Termination Condition Execution of each total action must terminate; this is an obligation of the programmer. If this condition is met then it can be shown that execution of each action terminates (see the rules for procedure execution, below).

3 Seuss Semantics (Operational)

At run time, a program consists of a set of boxes; their states are initialized at the beginning of the run. There are two different execution styles for a program. In a *tight execution* one action is executed at a time. There is no notion of concurrent execution; each action completes before the next action is started. In a *loose execution* actions may be executed concurrently.

The programmer understands a program by reasoning about its tight executions only. We have developed a logic for this reasoning. An implementation may choose a loose execution for a program to maximize resource utilization.

3.1 Tight Execution

A tight execution consists of an infinite number of steps; in each step, an action of a box is chosen and executed, as described below, in section 3.2. The choice of action to execute in a step is arbitrary except for the following fairness constraint: each action of each box is chosen eventually.

Observe that methods are executed only when they are called from other methods or actions, though actions execute autonomously (and eventually).

3.2 Procedure Execution

A method is executed when it is called. To simplify description, we imagine that an action is called by a *scheduler*. Then the distinction between a method and an action vanishes; each procedure is executed when called.

A procedure *accepts* or *rejects* a call. A total procedure always *accepts* calls; its body is executed whenever it is called. Termination

condition (see section 2.4) ensures that execution of each total procedure terminates. A partial procedure may *accept* or *reject* a call. First, we describe the rules of execution of a partial procedure g that consists of a single (positive) alternative, of the following form:

partial method $g(x, y)$:: p; $h(u, v) \rightarrow S$

Execution of g can be described by the following rules.

if $\neg p$ **then** *reject*
else {p holds} call h with parameters (u, v):
  if h *rejects* **then** *reject*
  else {h *accepts*}
    execute S using parameters, if any, returned by h;
    return parameters, if any, to the caller of g and *accept*
  endif
endif

As stated earlier, the programmer must ensure that execution of each total procedure terminates. It can be then be shown that the execution of any partial procedure g terminates, by using induction on the partial order induced by $\geq_g$ (see Calls Condition of section 2.4).

The caller is oblivious of rejection, because its body is not executed and its state remains unchanged. If all alternatives in a program are positive, then the effect of execution of an action is either rejection – then the state does not change for any box – or acceptance – some box state may change then. This is because, if any procedure rejects during the execution of an action then the entire action rejects. If any procedure accepts – the lowest procedure, that has no preprocedure, accepts first, followed by acceptances by its callers in the reverse order of calls – then the entire action accepts. This execution strategy meets the *commit* requirement in databases where a transaction either executes to completion or does not execute at all.

We have described the execution of a partial procedure that has a single (positive) alternative. In case a procedure has several alternatives, positive and negative, the following execution strategy is adopted. Recall that preconditions of the alternatives are disjoint.

if preconditions of all alternatives are *false* **then** *reject*
else {precondition of exactly one alternative f holds}
  if f is a positive alternative then execute as described above
  else {f is a negative alternative}
    execute f as if it is a positive alternative except:
      reject the call and do not return parameter values
  endif
endif

The execution of a negative alternative always results in a rejection. The caller is still oblivious of rejection, because its body is not executed and its state remains unchanged. However, a called procedure may change the state of its own box while rejecting the call, if it executes a negative alternative.

3.3 Examples of Alternatives

Use of Positive Alternatives

The following cat includes a method *get* that alternately gets an item from box $in1$ and box $in2$, starting with $in1$. In the following, $c = 1/c = 2$ if the next item is to be retrieved from $in1/in2$.

cat *multiplexor*
  var $c : \{1, 2\}$ **init** 1
  partial method $get(x$: type$)$::
    $c = 1;\ in1.get(x)\ \rightarrow c := 2$
  | $c = 2;\ in2.get(x)\ \rightarrow c := 1$
end {*multiplexor*}

The solution shown below avoids the use of alternatives by retrieving one item in advance and buffering it in y; *get* simply returns the value in y if y holds any data. The program includes two actions to read the next item from each of $in1$, $in2$ into y. We let y assume a special value, $\perp$, when it holds no fresh data.

cat *multiplexor1*
  var $c : \{1, 2\}$ **init** 1,
  var y: type **init** $\perp$
  partial method $get(x$: type$)$:: $y \neq \perp \rightarrow x, y := y, \perp$
  partial action $get1$:: $c = 1\ \wedge\ y = \perp;\ in1.get(y)\ \rightarrow c := 2$
  partial action $get2$:: $c = 2\ \wedge\ y = \perp;\ in2.get(y)\ \rightarrow c := 1$
end {*multiplexor1*}

The execution of this program does not quite match that of the original program. In general, alternatives can not be eliminated.

Use of Negative Alternatives

Strong Semaphore The following program implements a strong semaphore. A group of users share a semaphore. Any user that calls P persistently is eventually granted the semaphore. Each caller passes its id as a parameter to P. A negative alternative is used to record the id's of the callers to P whose call was rejected, so that they could be granted the semaphore in the same sequence in which they called P. The program uses the following variables:

q: the sequence of users whose last call on P was rejected.
$avail \equiv$ the semaphore is available.

cat *StrongSemaphore*
  var q: seq of id **init** $\langle\rangle$,
    $avail$: boolean **init** *true*

  partial method $P(i$: id$)$ ::
      $avail \wedge i = q.head \rightarrow avail, q := false\ , tail.q$
    $\not|\ i \notin q \rightarrow q := q : i$ {i is appended at the end of q}

  total method V:: $avail := true$
end {*StrongSemaphore*}

The reader can argue operationally that (1) at most one caller is granted the semaphore at any time, and (2) the solution is starvation-free: each persistent caller is eventually granted the semaphore, provided each caller who had been granted the semaphore relinquishes it eventually. Several variations of semaphores are treated in section 12.

Mutual Execution For the example in section 2.3, each user acquired two semaphores, s, t, before it could execute its critical section. Our syntax prohibits a procedure from calling multiple partial procedures. Thus, it is illegal to write in the box *user*

partial action *execute*:: *true* ; $s.P$; $t.P \rightarrow$ *critical section*; $s.V$; $t.V$

This is clearly the intent, though, and we show how to simulate this using negative alternatives. We introduce a cat, *MultiSemaphore*, that acquires the semaphores, s, t. This cat has a partial procedure P that accepts only if it has acquired both s and t; it also includes a total procedure V that releases both s, t. In order to avoid deadlock, P in *MultiSemaphore* acquires s, t in this order. The user simply calls *MultiSemaphore*.P to enter its critical section and calls *MultiSemaphore*.V upon completion of the critical section.

The code for *MultiSemaphore*.P first attempts to acquire s, and the call is rejected – using a negative alternative – even if s is acquired. After acquisition of s, a call to P is accepted only if t can be acquired. There are three instances of *MultiSemaphore*, u', v', w', in this program, each of which acts as an agent for a specific user, u, v, or w. The cat *user* includes the agent as a parameter.

program *MutualExclusion1*
cat *Semaphore*
  var n: nat **init** 1 {initially, the semaphore value is 1}
  partial method P:: $n > 0 \rightarrow n := n - 1$
  total method V:: $n := n + 1$
end {*Semaphore*}

box s, t : *Semaphore*

cat *MultiSemaphore*
  var hs, ht: boolean **init** *false*
  partial method P::
    $hs \wedge \neg ht$; $t.P \rightarrow ht := true$
    $\not\vert\ \neg hs$; $s.P \rightarrow hs := true$
  total method V:: $s.V$; $t.V$; $hs := false$; $ht := false$
end {*MultiSemaphore*}

cat *user*(ms : *MultiSemaphore*)
  partial action *execute*::
    true ; $ms.P \rightarrow$ *critical section*; $ms.V$
end {*user*}

box u', v', w' : *MultiSemaphore*
box u : *user*(u'), v : *user*(v'), w : *user*(w')

end {*MutualExclusion1*}

4 Discussion

4.1 Total vrs Partial Procedures

It may seem that total and partial procedures are interchangeable. A total procedure f can be coded as a partial procedure, $true \rightarrow f$. Also, a partial procedure can be coded as a total procedure where the outcome of the call – acceptance or rejection – is coded explicitly as a parameter that can be tested by the caller.

The distinction between total and partial procedures is fundamental. Total procedures model terminating computations, i.e., *transformational*, aspects of programming, and partial procedures model potentially non-terminating computations, or *reactive*, aspects[15]. In this view, a P operation on a semaphore is modeled by a partial procedure – because it may never terminate – whereas a V operation is a total procedure. A total procedure can be assigned a semantic with pre- and post-conditions, i.e., based on its possible inputs and corresponding outputs *without* considerations of interference with its environment, where as for partial procedures interaction with the environment is of the essence.

The distinction between total and partial procedures is important for concurrent implementation. It can be shown that two threads can execute concurrently given that (1) total procedures in different threads commute, and (2) partial procedures in each thread *semi-commute* with total procedures in the other thread (there is no requirement on the partial procedures of different threads). This condition allows a richer set of concurrent computations, because not all procedures are required to commute.

Total procedure

A total-body is a wait-free program. A *total* procedure can be assigned a meaning based only on its inputs and outputs; if the procedure is started in a state that satisfies the input specification then it terminates eventually in a state that satisfies the output specification. Procedures to sort a list, find a minimum spanning tree in a graph or send a job to an unbounded print-queue, are examples of total procedures. A total procedure need not be deterministic; e.g., *any* minimum spanning tree could be returned by the procedure. Furthermore, a total procedure need not be implemented on a single processor, e.g., the list may be sorted by a sorting network[1], for instance. Data parallel programs and other synchronous computation schemes are usually total procedures. A total procedure may even be a multiprogram in our model admitting of asynchronous execution, provided it is guaranteed to terminate, and its effect can be understood only through its inputs and outputs; therefore, such a procedure never waits to receive input, for instance. An example of a total procedure that interacts with its environment is one that sends jobs to a print-queue (without waiting); the jobs may be processed by the environment while the

procedure continues its execution. Almost all total procedures shown in this manuscript are sequential programs.

A total procedure may call only total procedures. When a total procedure is called (with certain parameters, in a given state) it may (1) terminate normally, (2) execute forever, or (3) fail. Nontermination of a total procedure is the result of a programming error. We require (see Termination Condition of section 2.4) the programmer to establish that the procedure is invoked only in those states where its execution is finite.

A failure is also caused by a programming error; it occurs when the procedure is invoked in a state in which it should not be invoked, for instance, if the computation requires a number to be divided by 0 or a natural number to be reduced below 0. Failure is a general programming issue, not just an issue in Seuss or multiprogramming. We interpret failure to mean that the resulting state is arbitrary; any step taken in a failed state results in a failed state. Typically, a hardware or software trap terminates the program when a failure occurs.

Example Consider a V operation on a binary semaphore. If the semaphore value is 0 then the application of V changes it to 1. What happens when the semaphore value is 1 prior to the application of V? There are at least 4 possibilities: (1) the operation is interpreted as a *skip* (i.e., the semaphore value remains 1 and the operation terminates), (2) the operation fails, i.e., it changes the semaphore value arbitrarily, to either 0 or 1, (3) the operation waits for the semaphore value to become 0, and (4) the operation never terminates. If we adopt interpretations (1) or (2) then we may regard the V operation as a total procedure. With interpretation (3), the operation is viewed as a partial procedure. We insist that the V operation be so implemented that the possibility (4) does not arise. In this document, we will treat V as a total procedure with meaning (2).

Partial procedure

A partial procedure models potentially non-terminating computations. Each execution of a partial procedure is terminating though the procedure may be called over and over (possibly, infinitely often). For instance, in traditional programming the caller of a P waits as long as the semaphore value is 0. In our model, each call to P terminates – possibly, rejected –, but the caller continues calling as long as the condition for the attempt is met. Thus, in $c; P \rightarrow S$ the body S is executed only if P accepts the call; if P rejects the call, S is not executed and the state of the caller's box does not change (thus preserving c). To simulate waiting, the execution of this procedure is attempted repeatedly as long as P rejects.

We believe that a caller should be oblivious to rejection. A rejection represents a transient condition where as acceptance represents a stable condition: if a process polls its incoming channel and finds it empty, it can not assert that it is empty (and, hence, start any computation based on channel emptiness) because the condition may be falsified even before the start of the computation. Therefore, a partial procedure treats a rejected call by doing nothing.

4.2 Tight vrs Loose Execution

In a tight execution an action is completed before another action is started. This allows a program execution to be understood by a single thread of control, avoiding interleaved executions of the action-bodies. Each procedure, total or partial, may be understood from its text alone given the meanings of the procedures that it calls, without consideration of interference by other procedures. A simple temporal logic, such as UNITY logic, is suitable for deducing properties of a program in this execution model.

An implementation, however, need not be restricted to a single thread as long as it achieves the same effect as a single-thread execution. Implementations may exploit the structures of Seuss programs (and user supplied directives) to run concurrent threads of actions with a fine grain of interleaving; these loose executions preserve the semantics of tight execution.

A consequence of having a single thread in a tight execution is that the notion of waiting has to be abandoned, because a thread can afford to wait only if there is another thread whose execution can terminate its waiting; rendezvous-based interactions [12, 17] that require at least two threads of control to be meaningful, have to be abandoned in this model of execution. We have replaced waiting by the refusal of a procedure to execute; i.e., by rejection.

4.3 Programming Methodology in Seuss

In the Seuss model, we view a multiprogram as a set of *actions* where each action deals with one aspect of the system functionality, and *execution of an action is wait-free*. Additionally, we specify the conditions under which an action is to be executed. Typical actions in an operating system may include the ones for garbage collection, response to a device failure by posting appropriate warnings and initiation of communication after receiving a request, for instance. Process control systems, such as avionics and telephony, may contain actions for processing of received data, updates of internal data structures, and outputs for display and archival recordings.

Seuss divides the multiprogramming world into (1) programming of action-bodies whose executions are wait-free, and (2) specifying the conditions for orchestrating the executions of the action bodies. Different theories and programming methodologies are appropriate for these two tasks. In particular, if the action-bodies are sequential programs then traditional sequential programming methodologies may be adopted for their developments. The orchestration of the actions has to employ some multiprogramming theory, but it is largely independent of the action-bodies. Seuss addresses only the design aspects of multiprograms – i.e., how to combine actions – and not the designs of the action-bodies.

Seuss severely restricts the amount of control available to the programmer at the multiprogramming level. The component actions of a program can be executed through infinite repetitions only. In particular, sequencing of two actions has to be implemented explicitly. Such loss of flexibility is to be expected when controlling larger abstrac-

tions. For an analogy, observe that machine language offers complete control over all aspects of a machine operation: the instructions may be treated as data, data types may be ignored entirely, and control flow may be altered arbitrarily. Such flexibility is appropriate when a piece of code is very short; then the human eye can follow arbitrary jumps, and "mistreatment" of data can be explained away in a comment. Flow charts are particularly useful in unraveling intent in a short and tangled piece of code. At higher levels, control structures for sequential programs are typically limited to sequential composition, alternation, and repetition; arbitrary jumps have nearly vanished from all high-level programming. Flow charts are of limited value at this level of programming, because intricate manipulations are dangerous when attempted at a higher level, and prudent programmers limit themselves to appropriate programming methodologies in order to avoid such dangers. We expect that the rules of combination have to become even simpler at the multiprogramming level. That is why we propose that the component actions of a multiprogram be executed using a form of repeated non-deterministic selection only.

4.4 Modular Multiprogramming

Traditionally, multiprograms consist of processes that execute autonomously. A typical process receives requests from the other processes, and it may call upon other processes for data communication or synchronization. The interaction mechanism – shared memory, message passing, broadcast, etc. – defines the platform on which it is most suitable to implement a specific multiprogram.

Seuss can be used to describe a process-based view of a multiprogram. A cat is a mechanism for grouping related actions. It is not a process, though traditional processes may be encoded as cats (as we have done for the *multiplexor*). A cat can be used to encode protocols for communication, synchronization and mutual exclusion, and it can be used to encode objects as in object-oriented programming. The only method of communication among the cats is through procedure calls, much like the programming methodology based on remote procedure calls. The minimality of the model makes it possible to develop a simple theory of programming.

4.5 Partial Order on Boxes

There is a partial order on the boxes of a program imposed by each procedure. This is in contrast to the usual views of process networks in which the processes communicate by messages or through a shared store. Typically, such a network is not regarded as being partially ordered. For instance, suppose that process P sends messages over a channel cp to process Q and Q sends over cq to P. The processes are viewed as nodes in a cycle where the edges (channels), cp and cq, are directed between P, Q. Similar remarks apply to processes communicating through shared memory. We view communication media (message channels and memory) as boxes. Therefore, we would represent the system described above as a set of four boxes: P, Q, cp and cq with the procedures (*send* and *receive*) in cp, cq being called

from P and Q, but not vice versa. The direction of message flow is immaterial in this hierarchy. A partial order is extremely useful in deducing properties by induction on the "levels" of the procedures.

The restriction that procedure calls are made along a partial order implies that a partial procedure at a lowest level is of the form $p \rightarrow S$, where the preprocedure is absent and the body S contains no procedure calls. A total procedure at a lowest level contains no procedure calls.

5 Small Examples

A number of small examples are shown in the rest of the paper. The goal is to show that typical multiprogramming examples from the literature have succinct representations in Seuss; additionally, that the small number of features of Seuss is adequate for solving many well-known problems: communications over bounded and unbounded channels, mutual exclusions and synchronizations, resource allocations, etc. We show a number of variations of some of these examples, implementing various progress guarantees, for instance.

Notational Conventions

Single Instance of a Cat Whenever a cat, C, has a single instance, b, we declare b and append the body of C to it. This eliminates explicit introduction of cat C.

Quantification We use quantification in writing arithmetic and boolean expressions, in coding a (bounded) number of alternatives of a procedure and a (bounded) number of procedures. In all cases, the form of a quantification is as follows: (*op dummy*: *range*: *body*), where, *op* is an operator, described below, *dummy* is a variable (or a list of variables), *range* defines the constraints on *dummy*, and *body* is used to construct the expression, alternatives, or the procedures. The *op* may be an arithmetic/boolean operator (for arithmetic/boolean expressions), $|$ and $\not|$ to create a set of alternatives of a procedure, and $[\!]$ to create a set of procedures. We require that only actions, not methods, may be quantified.

$(+i : 0 \leq i \leq N : A[i])$
$(\forall i : 0 \leq i < N : A[i] \leq A[i+1])$
$(\forall i, j : 0 \leq i \leq N \wedge 0 \leq j \leq N \wedge i \neq j : M[i,j] = 0)$

partial action:: $([\!]\ i : 0 \leq i < N : sem[i].P \rightarrow$ total body$)$

partial action::
$(|\ i : 0 \leq i < N : x = i \rightarrow x := 0$
$\not|\ i : N \leq i < 2 \times N : x = i \rightarrow x := i - N$
$)$

The first expression is the sum of the values in array $A[0..N]$. The second expression is *true* if A is sorted in ascending order. The next expression has two dummies; it is a boolean expression that is *true* if

all off-diagonal elements of matrix $M[0..N, 0..N]$ are zero. The first **partial action** declaration creates N partial actions where the i^{th} action attempts the P operation on the i^{th} semaphore, $sem[i]$. The last **partial action** declaration creates N positive alternatives and N negative alternatives.

6 Channels

6.1 Unbounded Fifo Channel

An unbounded fifo channel is a cat that has two methods: *put* (i.e., send) is a total method that appends an element to the end of the message sequence and *get* (i.e., receive) is a partial method that removes and returns the head element of the message sequence, provided it is non-empty. We define a polymorphic version of the channel where the message type is left unspecified. In the method *put*, we use : in the assignment to denote concatenation; $\langle\rangle$ denotes an empty sequence.

cat *FifoChannel*(type)
  var r: seq of type **init** $\langle\rangle$ {r is initially empty}
  partial method $get(x$: type):: $r \neq \langle\rangle \rightarrow x, r := r.head, r.tail$
  total method $put(x$: type):: $r := r : x$
end {*FifoChannel* of type }

In the following cat partial action *transfer* copies the elements of *in* to *out*, both being boxes of *FifoChannel* of integer. Since *transfer* is executed repeatedly, every element of *in* is eventually transferred to *out*.

cat *copy*
  var x: integer
  partial action *transfer*:: *true*; $in.get(x) \rightarrow out.put(x)$
end {*copy*}

The following cat is similar to *copy* except that it includes two partial actions, to read from either channel $in1$ or $in2$ and output to *out*. Since the two actions are executed infinitely often, this cat implements a fair merge of $in1$ and $in2$.

cat *merge*
  var x: integer
  partial action *transfer1*:: *true*; $in1.get(x) \rightarrow out.put(x)$
  partial action *transfer2*:: *true*; $in2.get(x) \rightarrow out.put(x)$
end {*merge*}

6.2 Bounded Fifo Channel

We show a bounded channel of size N, $N > 0$, below. Here both *put* and *get* are partial. The messages are kept in a circular buffer, b. Let $\oplus$ denote addition mod N; f is the index of the oldest message, r is the index of the youngest message $\oplus 1$, and k is the total number of messages in the channel.

cat *bch* of type
  var $b[0..N-1]$: type, $f, r, k : 0..N$ **init** 0
    {initially the buffer is empty}
  partial method *put*(x: type)::
      $k < N \rightarrow r, b[r], k := r \oplus 1, x, k + 1$
  partial method *get*(x: type)::
      $k > 0 \rightarrow f, x, k := f \oplus 1, b[f], k - 1$
end {*bch*(N) of type}

Next, consider a channel that can hold at most one message. The sender writes into the channel only when the channel is empty, and writing makes the channel full. The receiver reads and removes from the channel only when it is full; therefore, reading makes the channel empty. Thus, the sender receives an acknowledgment (that its previous output has been received) when it is able to write into the channel. As before, both *get* and *put* are partial methods. The variable w holds the contents of the channel if any, and *full* is *true* if and only if there is some data in w.

cat *word*(type)
  var w: type, *full*: boolean **init** *false* {the buffer is empty}
  partial method *put*(x: type):: $\neg full \rightarrow w,\ full := x,\ true$
  partial method *get*(x: type):: $full \rightarrow x,\ full := w,\ false$
end {*word* of type}

As an application of bounded channels, we consider the following example due to Hoare[11]. A multiplexor process receives a stream of messages from 10 different consoles. It acknowledges each message it receives and sends the received message along an output channel. A console may terminate the stream by sending a special end-of-stream (*eos*) message.

The solution in [11] uses rendezvous-based communication that eliminates the need for acknowledging the receipt of messages. We achieve a similar effect by requiring that a console and the multiplexor communicate over a bounded channel of size 1, i.e., a *word*. Then, each console is assured that its last message has been received if it is able to send another message. This is slightly inferior to rendezvous-based communications where the buffer size is zero and each communication is instantly acknowledged.

The *multiplexor* and the i^{th} console communicate via $c[i]$, $0 \leq i \leq 9$, where $c[i]$ is an instance of *word*. The *multiplexor* sends its outputs along a *FifoChannel* called *out*. Variable $more[i]$ is *true* if the *multiplexor* has not received a *eos* message from channel i.

box *multiplexor*
  var m: message, $more[0..9]$: boolean **init** *true*
  partial action::
    ($[\!]i : 0 \leq i \leq 9$:
      $more[i];\ c[i].get(m) \rightarrow out.put(m);\ more[i] := (m \neq eos)$
)
end {*multiplexor*}

There is no restriction on the order in which the partial actions in the multiplexor are executed. The fairness constraint ensures that

any message sent by a console is eventually received and output by the multiplexor.

6.3 Unordered Channel

The fifo channel guarantees that the order of delivery of messages is the same as the order in which they were put into the channel. Next, we consider an unordered channel that returns any message from the channel in response to a call on *get* when the channel is non-empty. The channel is implemented as a bag and *get* is implemented as a non-deterministic operation. We write $x :\in b$ to denote that x is assigned any value from bag b (provided b is non-empty). The usual notation for set operations is used for bags in the following example.

cat *uch* of type
  var b: bag of type **init** $\{\}$ {initially b is empty}
  partial method $get(x$: type$)$:: $b \neq \{\} \rightarrow x :\in b;\ b := b - \{x\}$
  total method $put(x$: type$)$:: $b := b \cup \{x\}$
end {*uch* of type}

This implementation does not guarantee that every message will eventually be delivered, given that messages are removed from the bag an unbounded number of times. Such a guarantee is, of course, established by the fifo channel. We propose a solution below that implements this additional guarantee. In this solution there is an index – a natural number – for every message and there is a variable t that is less than or equal to the smallest index. A message is assigned an index strictly exceeding t whenever it is put in the channel. The indices need not be distinct. The *get* method removes any message with the smallest index and updates t.

cat *nch* of type
  var b: bag of ($index$: nat, msg: type) **init** $\{\}$ {b is empty},
    t: nat **init** 0, s: nat, m: type
  partial method $get(x$: type$)$:: $b \neq \{\} \rightarrow$
    let (s, m) have minimum index, s, in b;
    remove (s, m) from b;
    $t, x := s, m$
  total method $put(x$: type$)$::
    $b := b \cup \{(s, x)\}$,
      where s is some natural number exceeding t
end {*nch* of type}

We now show that every message is eventually removed given that there are an unbounded number of calls on *get*. For a message with index i we show that the pair $(i - t, p)$, where p is the number of messages with index t, decreases lexicographically with each execution of *get*, and it never increases. Hence, eventually, $i = t$ and $p = 0$ implying that this message has been removed. An execution of *put* does not affect i, t or p, because the added message receives an index higher than t; thus, $(i - t, p)$ does not change. A *get* either increases t, thus decreasing $i - t$, or it keeps t the same and decreases p, thus, decreasing $(i - t, p)$.

6.4 Task Dispatcher

We design a task dispatcher that is interposed between a set of clients and a set of servers. A client generates a sequence of tasks, where each task has a *priority* between 0 and N. A server requests to process a task whenever it is idle; however, a server can only process tasks of certain priorities; a server that can process tasks of priority i can also process tasks of priorities less than i. The task dispatcher responds to a request from a server by sending it a task that the server can process.

It is easy to see that a task dispatcher is nothing but a glorified channel; it has two methods, *put* and *get*. The client calls *put*, with a task and its priority as parameters, to deposit a task in the channel, and a server calls *get* with a parameter value p, and the dispatcher sends a task of priority at most p to the server, if such a task exists.

In the following solution, $r[i]$ is a queue of tasks of priority i that have been deposited by the clients and have not yet been processed by the servers. The dispatcher always sends the task of the highest priority that it can possibly send.

cat *dispatcher*
  var $r[0..N]$: seq of task **init** $\langle\rangle, i : 0..N$
  partial method $get(x$: task, $p : 0..N)$::
    {get a task of priority p or lower, as close to p as possible}
    $(\exists j :: 0 \leq j \leq p \wedge r[j] \neq \langle\rangle) \rightarrow$
      $i := p$;
      while $r[i] = \langle\rangle$ do $i := i - 1$ enddo;
      $x, r[i] := r[i].head; r[i].tail$
  total method $put(x$: task, $p : 0..N)$:: $r[p] := r[p] : x$
end $\{dispatcher\}$

There is no guarantee that every task will eventually be removed. This modification of the solution is left to the reader.

6.5 Faulty Channel

A cat, *FaultyChannel*, that simulates message loss, duplication and out-of-order delivery in a channel, is shown in this section. Such a channel has the usual methods, *put* and *get*, by which the senders and the receivers interact with it. Additionally, the channel may lose messages, it may duplicate any message an unbounded (though finite) number of times, and it may permute the order of messages.

We simulate the faulty channel using a bag b, as in *uch*, in order to simulate out-of-order delivery. To simulate message loss and duplication, we associate a count n with each element that is *put*; the count is an arbitrary natural number, denoting the maximum number of times that the element is to be delivered. If $n = 0$ for a message then the message is immediately discarded, and for n exceeding 0 the message is added n times to b.

The given faulty channel can provide no guarantee of any message transmission at all because all messages may be lost; clearly, no useful device can be built out of such a channel. The next requirement we add is that a message that is *put* repeatedly is eventually delivered,

provided that the receiver calls *get* over and over. We implement this requirement by insisting that n become non-zero periodically in the method *put*. In the following, $x :\in b$ means that x is to be assigned an arbitrary value from b.

cat *FaultyChannel* of type
  var b: bag of type **init** $\{\}$
    {initially b is empty}

  partial method *get*(x: type)::
    $b \neq \{\} \rightarrow x :\in b;\ b := b - \{x\}$

  total method *put*(x: type)::
    Let n be a fair natural number;
    while $n \neq 0$ do
      $b := b \cup \{x\};\ n := n - 1$
    od
end {*FaultyChannel*}

Now, we argue that if *put* is called repeatedly with a value m and, also, *get* is called repeatedly, then m will eventually be returned as a result of *get*. Message m is eventually added to b because eventually $n > 0$. Let c be the number of elements of b that differ from m. Whenever a message other then m is returned in a call to *get*, the value of c decreases. Also, no message other than m is added to b by a *put*; hence, c does not increase. Therefore, messages other than m can be delivered only a finite number of times before m is returned as the result of a call to *get*.

This fault model of a channel is assumed in the Alternating Bit Protocol[21]. Such a protocol can be studied (proved correct) by encoding the communication between the sender and the receiver using *FaultyChannel*.

It can be shown that the proposed solution is *maximal*; that it can display any possible behavior of the faulty channel. This is a necessity if the program is to be used as a simulator for a faulty channel.

7 A Simple Database

We use a simple database example to illustrate the use of some of the cats introduced so far.

A cat, *database*, has total methods *insert*, *delete* and *query* that act upon a stored database, D. Each of these procedures returns one of three results; *eff* (effective), *error* (error) or *ineff* (ineffective). An *insert* of x has the outcome *eff* if x is not in D (prior to the operation) and there is enough room to add x to D, and in this case, x is added to D; the outcome is *error* if x is not in D and there is not enough room to insert x; and the outcome is *ineff* if x is already in D. A *delete* of x has an outcome *eff* if x is in D prior to the operation, and then x is removed from D; the outcome is *ineff*, otherwise. A *query* for x has an outcome *eff* if x is in D, the outcome is *ineff*, otherwise.

type outcome = (*eff*, *error*, *ineff*)

```
cat database
  var D: set of element init {} {the database is initially empty}
  total method insert(x: element, r: outcome)::
    if x ∈ D then r := ineff
      elseif there is room to add x then r := eff; add x to D
      else r := error
    endif
  total method delete(x: element, r: outcome)::
    if x ∈ D then r := eff; remove x from D
      else r := ineff
    endif
  total method query(x: element, r: outcome)::
    if x ∈ D then r := eff
      else r := ineff
    endif
end {database}
```

We create one instance of this cat.

```
box store : database
```

Now, consider two users each of whom sends a stream of requests for operations on *store*. The requests are directed to a *multiplexor*. The *multiplexor* accepts the requests in arbitrary order and upon completion of each request returns the result of the operation to the corresponding user. An operation's result is a boolean; it is *true* if and only if the operation was effective. The *multiplexor* also outputs a log of the effective *insert* and *delete* operations, from which the database can be reconstructed.

First, we consider the communications between the users and the *multiplexor*. We implement the communication of requests by using *word*, described in section 6.2. We create two instances of *word* – *xreq*, *yreq* – for the two users to send requests to the *multiplexor*, by the following declaration.

```
type request = record
      op : (insert, delete, query), n: element
endrecord
box xreq, yreq  : word of request
```

The *multiplexor* creates a log of the effective *insert* and *delete* operations by sending the sequence of effective requests over an unbounded fifo channel, see section 6.1. We create one instance of this cat, *log*. Also, we create two instances of *FifoChannel* – *xrep*, *yrep* – for the *multiplexor* to reply to the two users with the results of their requests.

```
box log : FifoChannel of request
box xrep, yrep  : FifoChannel of boolean
```

The box *multiplexor* consists of two actions, to read from *xreq* and *yreq*.

```
box multiplexor
  var req: request, r: outcome
  partial action :: true; xreq.get(req) →
    if req.op = insert then store.insert(req.n, r);
      if r = eff then log.put(req) endif
    elseif req.op = delete then store.delete(req.n, r);
      if r = eff then log.put(req) endif
    else {req.op = query} then store.query(req.n, r)
    endif;
    xrep.put(r = eff)

  partial action :: true; yreq.get(req) →
    if req.op = insert then store.insert(req.n, r);
      if r = eff then log.put(req) endif
    elseif req.op = delete then store.delete(req.n, r);
      if r = eff then log.put(req) endif
    else {req.op = query} then store.query(req.n, r)
    endif;
    yrep.put(r = eff)
end {multiplexor}
```

The trail of the effective requests may, alternatively, be stored in a *database* instead of being sent on a fifo channel; then, declare **box** *log* : *database*, and let the *multiplexor* insert the effective requests into *log*. Our current implementation of *database* does not save the sequence in which data are inserted. Therefore, the *multiplexor* will have to add a sequence number explicitly to each effective request before inserting it into *log*.

8 Example of a Process Network

This example is attributed to Hamming in Dijkstra[10]. It is required to compute the sequence of integers of the form $2^i \times 3^j \times 5^k$ in increasing order, for all natural numbers i, j, k. Our solution follows the treatment in Section 8.2 (page 182) of Chandy and Misra[5], and it is sketched briefly below.

The computation strategy is as follows. Let H denote the sequence to be output. Then, $H = \langle 1 \rangle : merge(2 \times H,\ 3 \times H,\ 5 \times H)$, i.e.,

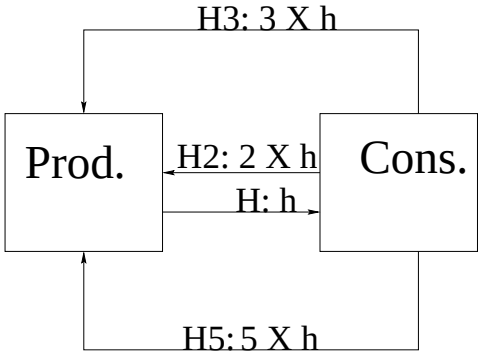

Figure 1: Network to Compute $2^i \times 3^j \times 5^k$

H is formed by starting with 1 and appending the *merge* of $2 \times H,\ 3 \times H,\ 5 \times H$. This equation has a unique solution in H where the function *merge* constructs an increasing sequence out of its argument sequences, each of which is also increasing (*merge* drops the duplicates from its arguments).

This computation strategy is realized by the network shown in Figure 1. The box *produce* receives $2 \times H,\ 3 \times H,\ 5 \times H$ along the *FifoChannel*s $H2, H3, H5$ respectively, and it produces the desired sequence along *FifoChannel* H by merging its inputs; initially, H has the integer 1 on it. The box *consume* removes items from H; for a removed item h it sends $2 \times H,\ 3 \times H,\ 5 \times H$ along the *FifoChannel*s $H2, H3, H5$, respectively.

The program *Hamming* is shown below. Box *produce* has three variables, $h2, h3, h5$, where $h2$ is the last number received along $H2$ that is yet to be sent along H; if all numbers received have already been output then $h2$ is 0; $h3, h5$ have similar meanings. Box *produce* has two kinds of partial actions, *read* and *write*. A *read* action, say *read*2, receives the next value from $H2$ provided $h2 = 0$. Procedure *write* outputs the smallest of $h2, h3, h5$, when they are all nonzero, along H and updates them if appropriate. The computation is started by having the value 1 in channel H, initially; this is accomplished by executing the method $put(1)$ in H. Box *consume* contains a single action that reads an input h from H and outputs $2 \times h,\ 3 \times h,\ 5 \times h$ along the *FifoChannel*s $H2, H3, H5$, respectively.

program *Hamming*
  box $H2, H3, H5$: *FifoChannel*(integer)
  box H: *FifoChannel*(integer) **init** $put(1)$

  box *produce*
    var $h2, h3, h5$: nat **init** 0, f: nat

    partial action *read2*:: $h2 = 0;\ H2.get(h2) \rightarrow\ skip$
    partial action *read3*:: $h3 = 0;\ H3.get(h3) \rightarrow\ skip$
    partial action *read5*:: $h5 = 0;\ H5.get(h5) \rightarrow\ skip$

    partial action *write*::
      $h2 \neq 0 \wedge\ h3 \neq 0 \wedge\ h5 \neq 0 \rightarrow$
        $f := \min(h2, h3, h5);\ H.put(f);$
        if $f = h2$ then $h2 := 0;$
        if $f = h3$ then $h3 := 0;$
        if $f = h5$ then $h5 := 0$
  end {*produce*}

  box *consume*
    var h: nat
    partial action::
    true ; $H.get(h) \rightarrow$
      $H2.put(2 \times h);$
      $H3.put(3 \times h);$
      $H5.put(5 \times h)$
  end {*consume*}

end {*Hamming*}

9 Broadcast

We show a cat that implements broadcast-style message communication. Processes, called *writers*, attempt to broadcast a sequence of values to a set of *readers*. A new value can be broadcast only if all previous values have been read by all readers. Cat *broadcast* synchronizes the reads and writes as follows.

The value to be broadcast is stored in variable v; and n counts the number of readers that have read v. Let N be the total number of readers. Both *read* and *write* are partial methods. The precondition for *write* is that the counter n equals N, i.e., all readers have read the current value. The precondition for *read* is that this particular reader has not read the current value of v.

To implement the precondition for reading, we associate a sequence number with the value stored in v. It is sufficient to have a 1-bit sequence number, a boolean variable t, as in the Alternating Bit Protocol for communication over a faulty channel [21]. A *read* operation has a boolean argument, s, that is the last sequence number read by this reader. If s and t match then the reader has already read this value and, hence, the call upon *read* is rejected. If s and t differ then the reader is allowed to read the value and both s and n are updated. The binary sequence number, t, is reversed whenever a new value is written to v. It is easy to show that n equals the number of readers whose s-value equals the cat's t-value. Initially, the local variable s for each reader is *true*.

cat *broadcast* of type
  var v: type, $n : 0..N$ **init** N, t: boolean **init** *true*
  partial method *read*(s: boolean, x: type)::
    $s \neq t \rightarrow s, x, n := t, v, n + 1$
  partial method *write*(x: type):: $n = N \rightarrow t, v, n := \neg t, x, 0$
end {*broadcast*}

10 Barrier Synchronization

Each process in a group of concurrently executing processes performs its computation in a sequence of phases. It is required that no process begin executing its $(p + 1)^{th}$ phase until all processes have completed their p^{th} phase, $p \geq 0$. Each process has a variable k, the highest phase that this process has completed. The cat *barrier* has a partial method, *sync*, that is called by each process with parameter k in order for it to advance to phase $k + 1$. Initially, $k = 0$ for all processes. The protocol for the process is shown below.

box *process*
  var k: nat **init** 0
  partial action ::
    true; *barrier.sync*(k) $\rightarrow$ do next phase
end {*process*}

The cat *barrier* has a phase number p, where p is the highest phase that all processes have completed. The following solution, due to Rajeev Joshi, is based on the observation that the call to *sync* should be accepted for a process with parameter k provided that $k = p$, i.e., all processes have completed phase k. Let n be the number of processes that have not yet started the $(p+1)^{th}$ phase. Then n decreases with every accepted call by *sync*, and if n becomes 0 then both p, n are updated. Let the number of processes be N.

box *barrier*
  $n : 0..N$ **init** N,
  p: nat **init** 0
  partial method *sync*(k: nat)::
    $k = p \rightarrow k, n := k+1, n-1;$
      if $n = 0$ then $p, n := p+1, N$
end {*barrier*}

It can be shown that $p \leq k \leq p+1$ is an invariant of this program, for all k of different processes. Therefore, $k = p$ may be evaluated by comparing the lowest bits of k, p, and incrementation of k, p can be implemented by inverting their lowest bits. Hence, we introduce booleans s, t that represent the lowest bits of k, p respectively. Such a solution is shown below.

box *process1*
  var s: boolean **init** *true*
  partial action ::
    true; *barrier.sync*(s) $\rightarrow$ do next phase
end {*process1*}

box *barrier1*
  $n : 0..N$ **init** N,
  t: boolean **init** *true*
  partial method *sync*(s: boolean)::
    $s = t \rightarrow s, n := \neg s, n-1;$
      if $n = 0$ then $t, n := \neg t, N$
end {*barrier1*}

11 Readers and Writers

We consider the classic Readers Writers Problem [6] in which a common resource – say, a file – is shared among a set of *reader* processes and *writer* processes. Any number of readers may have simultaneous access to the file where as a writer needs exclusive access. There are two partial methods, $StartRead$ and $StartWrite$, by which a reader and a writer gain access to the resource, respectively. Upon completion of its access, a reader releases the resource by calling the total method $EndRead$, and a writer by calling $EndWrite$. We assume throughout that read and write operations are finite, i.e., each accepted $StartRead$ is eventually followed by a $EndRead$ and a $StartWrite$ by $EndWrite$.

We employ a parameter N which is the maximum number of readers permitted to have simultaneous access to the resource; N may be

set arbitrarily high to permit simultaneous access by all readers. The following solution, based upon one in section 6.10 of [5], uses a pool of *tokens*. Initially, there are N tokens. A reader needs 1 token and a writer N tokens to proceed. It follows that many (up to N) readers could be active simultaneously where as at most one writer will have access to the resource at any time. Upon completion of their accesses, the readers and the writers return all tokens they hold, 1 for a reader and N for a writer, to the pool. In the following program n is the number of available tokens.

cat *ReaderWriter*
  var $n : 0..N$ **init** N
  partial method $StartRead :: n > 0 \rightarrow n := n - 1$
  partial method $StartWrite :: n = N \rightarrow n := 0$
  total method $EndRead :: n := n + 1$
  total method $EndWrite :: n := N$
end {*ReaderWriter*}

Guaranteed Progress for Writers

The solution given above can make no guarantee of progress for either the readers or the writers. Our next solution guarantees that readers will not permanently overtake writers: if there is a waiting writer then some writer gains access to the resource eventually. The strategy is to deny access to the readers if there is some writer attempting to execute $StartWrite$. A boolean variable, $WriteTry$, is *true* if a call upon $StartWrite$ has been rejected since completion of the last write operation, i.e., $EndWrite$. We don't argue the correctness of this solution because a more general case is treated next. We note, however, that writers may permanently overtake the readers in this solution.

cat *ReaderWriter1*
  var $n : 0..N$ **init** $N, WriteTry$: boolean **init** *false*
  partial method $StartRead ::$
    $n > 0 \wedge \neg WriteTry \rightarrow n := n - 1$
  partial method $StartWrite ::$
    $n = N \rightarrow n := 0$
    $\not| \; n \neq N \rightarrow WriteTry := true$
  total method $EndRead :: n := n + 1$
  total method $EndWrite :: n := N;\ WriteTry := false$
end {*ReaderWriter1*}

Guaranteed Progress for Readers and Writers

The next solution guarantees progress for both readers and writers; it is similar to the previous solution – we introduce a boolean variable, $ReadTry$, analogous to $WriteTry$. However, the analysis is considerably more complicated in this case. We outline an operational argument for the progress guarantees; a formal argument will appear elsewhere.

cat *ReaderWriter2*
  var $n : 0..N$ **init** N,
    $WriteTry, ReadTry$: boolean **init** *false*
  partial method *StartRead* ::
      $n > 0 \wedge \neg WriteTry \rightarrow n := n - 1$
    $\not\downarrow$ $n = 0 \rightarrow ReadTry := true$
  partial method *StartWrite* ::
      $n = N \wedge \neg ReadTry \rightarrow n := 0$
    $\not\downarrow$ $n \neq N \rightarrow WriteTry := true$
  total method *EndRead* :: $n := n + 1$; $ReadTry := false$
  total method *EndWrite* :: $n := N$; $WriteTry := false$
end {*ReaderWriter2*}

We show that if $WriteTry$ is ever *true* it will eventually be falsified, asserting that a write operation will complete eventually, i.e., $EndWrite$ will be called. Similarly, if $ReadTry$ is ever *true* it will eventually be falsified. To prove the first result, consider the state in which $WriteTry$ is set *true* (note that initially $WriteTry$ is *false*). Since $n \neq N$ is a precondition for such an assignment, either a read or a write operation is underway. If it is the latter case, then the write will eventually be completed by calling $EndWrite$, thus setting $WriteTry$ to *false* . If $WriteTry$ is set when a read is underway then no further call on *StartRead* will be accepted and successive calls on *EndRead* will eventually establish $n = N \wedge \neg ReadTry$, i.e., $n = N \wedge \neg ReadTry \wedge WriteTry$ will hold. No method other than $StartWrite$ will execute in this state: none of the alternatives of *StartRead* will accept; no call upon *EndRead* or $EndWrite$ will be made because no read or write operation is underway, from $n = N$. Therefore, a call upon $StartWrite$ will be accepted, which will be later followed by a call upon $EndWrite$.

The argument for eventual falsification of $ReadTry$ is similar. The precondition of the assignment $ReadTry := true$ is $n = 0$ implying that either N readers are reading or a write operation is underway. In the former case, no more readers will be allowed to join, and upon completion of reading (by any reader) $ReadTry$ will be set *false*. In the latter case, upon completion of writing $EndWrite$ will be called and its execution will establish $n = N \wedge ReadTry \wedge \neg WriteTry$. No method other than *StartRead* will execute in this state, and any reader that succeeds in executing *StartRead* will eventually execute *EndRead*, thus falsifying $ReadTry$.

Starvation-Freedom for Writers

Our final variation guarantees absence of starvation for the writers, but no progress guarantees for the readers. We identify a writer by including its process id as a parameter in the call to $StartWrite$. A queue of writer ids is maintained and $StartWrite$ accepts a call only if $n = N$ and the caller is at the head of the queue. The test on variable $WriteTry$ is replaced by a test on the queue length. In the following, the type pid stands for process id.

cat *ReaderWriter3*
  var $n : 0..N$ **init** N, wq: seq of pid **init** $\langle\rangle$

partial method $StartRead$ ::
$n > 0 \wedge wq = \langle\rangle \rightarrow n := n - 1$

partial method $StartWrite(i\text{: pid})$ ::
$n = N \wedge i = wq.head \rightarrow n := 0;\ wq := tail.wq$
$\not\downarrow\ i \notin wq \rightarrow wq := wq : i$

total method $EndRead$ :: $n := n + 1$

total method $EndWrite$ :: $n := N$
end $\{ReaderWriter3\}$

A solution that guarantees absence of starvation for both readers and writers is slightly more involved. One strategy is to create a single queue in which the list of reader and writer ids are kept for the calls that have been rejected; subsequent calls are accepted in order of appearance in this queue. Consecutive readers in the queue are permitted to have simultaneous access to the resource.

12 Semaphore

A binary semaphore, often called a *lock*, is typically associated with a resource, such as a file, device or communication channel[9]. A process has exclusive access to a resource only when it has *acquired*, i.e., it holds the corresponding semaphore. A process acquires a semaphore by completing a P operation and it releases the semaphore by executing a V. We regard P as a partial method and V as a total method.

Traditionally, a semaphore is *weak* or *strong* depending on the guarantees made about the eventual success (i.e., acceptance) of the individual calls on P. For a weak semaphore no guarantee can be made about the success of a particular process no matter how many times it attempts a P, though it can be asserted that a call on P by some process is accepted if the semaphore is available. Thus, a specific process may be *starved*: it is never granted the semaphore even though another process may hold it arbitrarily many times. A strong semaphore avoids individual (process) starvation: if the semaphore is available infinitely often then it is eventually acquired by each process attempting a P operation. We discuss both types of semaphores and show several subtle variations in their implementations.

In section 12.3 we introduce a new kind of semaphore, called the *snoopy semaphore*. Unlike a typical semaphore that is first acquired and then released after its associated resource has been used, the holder of a snoopy semaphore, s, releases s only if there are other processes requesting it. This is a useful strategy if there is low contention for the resource (and, hence, for its associated semaphore), because a process may use the resource as long as it is not required by the other processes.

We restrict ourselves to binary semaphores in all cases; extensions to general semaphores are straightforward; see section 2.3 for an example of a general weak semaphore.

12.1 Weak Semaphore

The following cat describes a weak binary semaphore.

cat *semaphore*
  var *avail*: boolean **init** *true* {the semaphore is available}
  partial method P:: $avail \rightarrow avail := false$
  total method V:: $avail := true$
end {*semaphore*}

A typical calling pattern on such a semaphore is shown below.

box s : *semaphore*

box *user*
  partial action :: c; $s.P \rightarrow$ use the resource for s; $s.V$
  { other actions of the box}
end {*user*}

Note that the *user* can release the semaphore (i.e., execute $s.V$) in an action different from the one in which the semaphore was acquired. Usually, once the precondition c becomes *true* then it remains *true* until the process acquires the semaphore. There is no requirement in Seuss, however, that c will remain *true* as described. This feature can be used to acquire either of the two semaphores, $s1$ and $s2$, as shown below.

box *s1*, *s2* : *semaphore*

box *user*
  partial action :: c; $s1.P \rightarrow$ use $s1$'s resource; $s1.V; c := false$
  partial action :: c; $s2.P \rightarrow$ use $s2$'s resource; $s2.V; c := false$
  { other actions of the cat}
end {*user*}

The implementation of the weak semaphore does not impose any restriction on its callers; for instance, a process that does not hold the semaphore may release it by executing a V operation, thereby causing the semaphore to be acquired by a process while another process is still holding the semaphore. The proposed implementation is appropriate when the semaphore is shared by processes that are designed by trustworthy programmers. A more elaborate implementation is shown below that restricts the V operation to be applied only by the semaphore holder. If a non-holder attempts a V then there is no effect (another possibility would be to make the program fail in this case). In the following, the type pid stands for process id; each caller on P or V supplies its id as a parameter.

cat *semaphore1*
  var *holder*: pid **init** *nil* {initially the semaphore is not held}
  partial method $P(i$: pid) :: $holder = nil \rightarrow holder := i$
  total method $V(i$: pid) ::
      if $holder = i$ then $holder := nil$ endif
end {*semaphore1*}

One of the drawbacks of this solution is that a process can not transfer a semaphore it holds to another process, the latter releasing the semaphore subsequently. Our next solution treats each semaphore as a *ticket* that may freely be passed around among the processes. The process id is then irrelevant; a P operation returns a ticket (an arbitrary positive integer) and a V operation has effect only if attempted by the appropriate ticket holder. Let AN be a cat that returnds a natural number j in $AN.anat(j)$. In the following solution we assign positive integers to tickets.

cat *semaphore2*
  var *holder*: nat **init** 0 {initially the semaphore is not held},
      j: nat
  partial method $P(i$: nat) ::
      $holder = 0 \rightarrow AN.anat(j);\ i, holder := j+1, j+1$
  total method $V(i$: nat) ::
      if $holder = i$ then $holder := 0$ endif
end {*semaphore2*}

This solution still does not guarantee that the holder will have exclusive access to the resource. A determined intruder could attempt to guess the holder's ticket value in a series of attempts. For use in an environment where malice rules, tickets should carry digital signatures [7, 20]. The current implementation guards against errors caused by poor programming or hardware malfunction.

None of the implementations shown in this section guarantees absence of individual starvation: a cat may have a partial action of the form $c; s.P \rightarrow ...$ where s is an instance of a weak semaphore and the precondition, c, remains *true* as long as $s.P$ is not accepted. It cannot be guaranteed that the call on $s.P$ will ever be accepted. We can, however, assert a simple form of progress: if each accepted P is subsequently followed by a call upon V then eventually some process's call upon P is accepted.

12.2 Strong Semaphore

A strong semaphore guarantees absence of individual starvation; in Seuss terminology, if a cat contains a partial action of the form, $c;\ s.P \rightarrow ...$, where the precondition c remains *true* as long as $s.P$ is not accepted and s is a strong semaphore, then $s.P$ will eventually be accepted.

The following cat, shown earlier in section 3.3, implements a strong semaphore. The call upon P includes the process id as a parameter. Procedure P adds the caller id to a queue, q, if the the id is not in q, and it grants the semaphore to a caller provided the semaphore is available and the caller id is at the head of the queue.

cat *StrongSemaphore*
  var q: seq of id **init** $\langle\rangle$,
    avail: boolean **init** *true*

  partial method $P(i$: id) ::
      $avail \wedge i = q.head \rightarrow avail, q := false\ , tail.q$

$\not| \ i \notin q \rightarrow q := q : i$ {i is appended at the end of q}

total method V:: $avail := true$

end {*StrongSemaphore*}

The sequence q may be replaced by a fair bag, as was done for the unordered channel, *nch* (see section 6.3). Note that a call upon P is rejected even when the queue is empty and the semaphore is available. It is straightforward to add an alternative to grant the semaphore in this case.

The variations shown for the weak semaphore may also be applied to the strong semaphore. We consider a variation that employs a ticket instead of the process id as a parameter.

A process calls P with a parameter t that is initially 0. Procedure P rejects the call; however, it sets t to the position of this process in the queue. If a call upon P has a parameter value equaling the head position of the queue and the semaphore is available then the call is accepted. A V operation makes the semaphore available provided the caller shows the appropriate ticket, otherwise the call has no effect. Since the tickets have consecutive values as integers, we need two variables, f and r, to keep the front and rear positions in the queue, i.e.,

f= the value of the lowest issued ticket that is still outstanding if $f < r$

r= the value to be assigned to the next ticket

We have the invariant $f \leq r$. Note that $f = r$ implies that there is no outstanding ticket.

cat *StrongSemaphore1*
  var f, r: nat **init** 1, $avail$: boolean **init** *true*
  {initially the semaphore is available}
  partial method $P(t$: nat) ::
    $avail \wedge f = t \wedge f < r \rightarrow avail, f := false\ , f + 1$
    $\not| \ t = 0 \rightarrow t, r := r, r + 1$
  total method $V(t$: nat)::
    if $t = f - 1$ then $avail := true$ endif
end {*StrongSemaphore1*}

This solution can be made slightly more secure by assigning random integers as ticket values, as was done for the weak semaphore. Then, the order in which the tickets are issued has to be stored in a queue to guarantee absence of starvation. Such a solution is shown below.

cat *StrongSemaphore2*
  var q: seq of nat **init** $\langle\rangle$, *holder*: nat **init** 0
  {initially the semaphore is available}
  partial method $P(t$: nat) ::
    $holder = 0 \wedge t = q.head \rightarrow holder, q := t, tail.q$
  $\not| \ t = 0 \rightarrow t :=$ a positive integer not in q; $q := q : t$
  total method $V(t$: nat)::
    if $holder = t$ then $avail := true$ endif
end {*StrongSemaphore2*}

A process requesting a semaphore is a *persistent* caller if it calls the P operation infinitely often as long as it has not acquired the semaphore, otherwise it is a *transient* caller. Our solutions for the strong semaphore work only if all callers are persistent. If there is a transient caller, it will block all other callers from acquiring the semaphore. Unfortunately, there exists no solution for this case: there can be no guarantee that every persistent caller will eventually acquire the semaphore (given that every holder of the semaphore eventually releases it) in the presence of transient callers. A reasonable compromise is to add a new total method to the strong semaphore which a transient caller may call to remove its process id from the queue of callers.

12.3 Snoopy Semaphore

Traditionally, a semaphore associated with a resource is first acquired by a process executing a P, the resource is used and then the semaphore is released by executing a V. We consider a variation of this traditional model in which the resource is not released unless there are outstanding requests for the resource by the other processes. This is an appropriate strategy if there is low contention for the resource, because a process may use the resource as long as it is not required by the others. We describe a new kind of semaphore, called a *SnoopySemaphore*, and show how it can be used to solve this problem. In a later section, we employ the snoopy semaphore to solve a multiple resource allocation problem in a starvation-free fashion.

We adopt the strategy that a process that has used a resource *snoops* to see if there is demand for it, from time to time. If there is demand, then it releases the semaphore; otherwise, it may continue to access the resource.

A weak snoopy semaphore is shown below. We add a new method, S (for *snoop*), to *semaphore*. Thus, a *SnoopySemaphore* has three methods: P, V, and S. Methods P and V have the same meaning as for traditional semaphores: a process attempts to acquire the semaphore by calling the partial method P, and releases it by calling V. The partial method S accepts if the last call upon P by some process has been rejected. A process typically calls S after using the resource at least once, and it releases the semaphore if S accepts. In the following solution, boolean variable b is set *false* whenever a call on P is accepted, and set *true* whenever a call on P is rejected. Thus, b is *false* when a process acquires the semaphore and if it subsequently detects that b is *true* then the semaphore is in demand.

cat *SnoopySemaphore*
  var b: boolean **init** *false* , $avail$: boolean **init** *true*
    {initially the semaphore is available}
  partial method P ::
      $avail \rightarrow avail, b := false\ , false$
    $\not|\ \neg avail \rightarrow b := true$
  total method V:: $avail := true$
  partial method S:: $b \rightarrow skip$
end {*SnoopySemaphore*}

The proposed solution implements a weak snoopy semaphore; there is no guarantee that a specific process will ever acquire the semaphore. Our next solution is similar to *StrongSemaphore*. Since that solution already maintains a queue of process ids (whose calls on P were rejected), we can implement S very simply.

cat *StrongSnoopySemaphore*
  var q: seq of pid **init** $\langle\rangle$, *avail*: boolean **init** *true*
    {initially the semaphore is available}
  partial method $P(i$: pid) ::
      $avail \wedge i = q.head \rightarrow avail, q := false\ , tail.q$
    $\not\downarrow\ i \notin q \rightarrow q := q : i$
  total method V:: $avail := true$
  partial method S:: $q \neq \langle\rangle \rightarrow skip$
end {*StrongSnoopySemaphore*}

The variations employing tickets can be also be used with snoopy semaphores. Note that the two methods S and V may be combined into a single method if every process calls V only after a call upon S is accepted. We have retained them as two separate methods to allow a process to release the semaphore unconditionally.

13 Multiple Resource Allocation

A typical problem in resource allocation has a set of resources and a set of processes, where each process is in one of three states: *thinking*, *hungry*, and *eating*. A thinking process has no need for any resource. A thinking process may become hungry for exclusive use of a subset of the resources; the specific subset may differ with each thinking to hungry transition for a process. A hungry process remains hungry until it acquires all the resources it needs, then it transits to the eating state. We assume through out that every eating process eventually transits to the thinking state.

A solution for this problem specifies the steps to be taken by a hungry process to acquire all the resources it needs and the protocol for releasing the resources. A solution is *starvation-free* if each process transits from hungry to eating; a solution is *deadlock-free* if some hungry process transits to eating. We will associate a semaphore with each resource, and henceforth, we shall not distinguish between the semaphore and the resource it represents. Different kinds of semaphores will be used to guarantee different properties of the solutions.

The problem stated above is quite general. If there is a single resource then the problem is equivalent to the mutual exclusion problem where the *critical section* corresponds to the eating state: two processes can not be in the eating state simultaneously because both would then have exclusive access to the resource. The problem also subsumes the classical *dining philosophers* problem and its variations [9, 3]: there are equal number of processes and resources, numbered 0 through N, and the i^{th} process needs resources i and $i \oplus 1$, where $\oplus$ is addition modulo N, when it is hungry.

We show the code for a generic process. The action for transition from thinking to hungry is not shown; it is accompanied by setting the array *needs*, where $needs[i]$ is *true* if the process needs resource i. Also, the transition from eating to thinking is not shown. Throughout this description, $N \geq 0$, and variable r is an array $[0..N]$ of semaphores, one for each resource. Different solutions will employ different kinds of semaphores. The state of a process – thinking, hungry or eating – is in variable *state*. We write *thinking* as an abbreviation for $state = thinking$; similarly *hungry* and *eating*.

13.1 A Deadlock-Free Solution

Our first solution is well known in the operating systems literature: a hungry process acquires the resources it needs in increasing order of resource index. In the following solution we have the invariant that all needed resources up to d have been acquired by the process; i.e.,

$$
\begin{array}{l}
(\forall i : 0 \leq i < N : \\
\quad \text{process holds the } i^{th} \text{ semaphore} \equiv hungry \wedge needs[i] \wedge i < d \\
)
\end{array}
$$

For our first solution, the semaphores are all weak semaphores, i.e., of type *semaphore* from section 12.1. The process releases all the semaphores it holds when it is in the thinking state; the order of release is immaterial.

box $r[0..N]$: *semaphore*

box *user*1
  var $needs[0..N]$: boolean **init** *false* , d: $0..(N+1)$ **init** 0,
    $state$:$(thinking, hungry, eating)$ **init** *thinking*

  partial action *acquire* ::
      $hungry \wedge d \leq N \wedge \neg needs[d] \rightarrow d := d + 1$
    | $hungry \wedge d \leq N \wedge needs[d]; r[d].P \rightarrow d := d + 1$

  partial action *eat* ::
      $hungry \wedge d > N \rightarrow state := eating$; use resources

  partial action *release* ::
      $thinking \wedge d > N \rightarrow$
        while $d \neq 0$ do
          $d := d - 1$; if $needs[d]$ then $r[d].V$ endif
        enddo
end $\{user1\}$

We argue that this solution is deadlock-free. Assume to the contrary that at some point in the computation, a set of processes are hungry, but no process eats. We may assume that there is no thinking to hungry transition beyond this point, because this transition can happen only a finite number of times before all processes become hungry. Since no process eats no semaphore is released. Consider a hungry process j whose d value is d_j. Since it is permanently blocked,

its calls upon $r[d_j].P$ are permanently rejected. Therefore, semaphore d_j is held by a process, k. From the invariant, k is hungry, $d_j < d_k$, and k is blocked for d_k. Proceeding in this fashion, we can show an infinite sequence of increasing d values, an impossibility.

13.2 A Starvation-Free Solution

Our next solution is starvation-free. We employ

box $r[0..N]$: *StrongSemaphore*

and everything else remains the same. The proof is along the same lines as for the weak semaphore case. Suppose there is a process, j, that remains permanently blocked for semaphore d_j. Then semaphore d_j is never released beyond some point in the computation; if it is infinitely often released then j will acquire it, according to the properties of strong semaphore. Let the permanent holder of d_j be process k, and according to the invariant k is hungry, $d_j < d_k$, and k is blocked for d_k. We derive an impossibility as before.

13.3 A Deadlock-Free Solution with Snoopy Semaphores

The next solution employs an array of snoopy semaphores. A semaphore is not released until there is demand for it. We introduce a new array for each process: $holds[i]$ is *true* if this process holds semaphore i. Assume that d is set 0 along with the transition from eating to thinking.

box $r[0..N]$: *SnoopySemaphore*

box *user*2
  var $needs[0..N]$, $holds[0..N]$: boolean **init** *false* ,
    d: $0..(N+1)$ **init** 0,
    state:$(thinking, hungry, eating)$ **init** *thinking*

  partial action *acquire* ::
    $hungry \wedge d \leq N \wedge (\neg needs[d] \vee holds[d]) \rightarrow d := d + 1$
  | $hungry \wedge d \leq N \wedge needs[d] \wedge \neg holds[d]; r[d].P \rightarrow$
      $holds[d] := true;\ d := d + 1$

  partial action *eat* ::
    $hungry \wedge d > N \rightarrow$
      $state := eating$; use resources

  partial action *release* ::
    ($[\!]j : 0 \leq j \leq N$:
      $holds[j] \wedge (\neg needs[j] \vee j \geq d); r[j].S \rightarrow$
        $r[j].V;\ holds[j] := false$
)
end $\{user2\}$

14 Related Work

Our work incorporates ideas of serializability from databases[2], objects and inheritance[16], Communicating Sequential Processes[12], i/o automata[14], and Temporal Logic of Actions[13]. A partial procedure is similar to a database (nested) transaction that may commit or abort; the procedure commits (to execute) if its precondition holds and its preprocedure commits, and it aborts otherwise. A typical abort of a database transaction requires a rollback to a valid state. In Seuss, if there are no negative alternatives, a partial procedure does not change the program state until it commits, and therefore, there is no need for a rollback. The form of a partial procedure is inspired by Communicating Sequential Processes[12]. Our model may be viewed as a special case of CSP because we disallow nested partial procedures.

Seuss is an outgrowth of our earlier work on UNITY [5]. A UNITY program consists of statements each of which may change the program state. A program execution starts in a specified initial state. Statements of the program are chosen for execution in a non-deterministic fashion, subject only to the (fairness) rule that each statement be chosen eventually. The UNITY statements were particularly simple – assignments to program variables – and the model allowed few programming abstractions besides asynchronous compositions of programs.

Seuss is an attempt to build a compositional model of multiprogramming, retaining some of the advantages of UNITY. An action is similar to a statement, though we expect actions to be much larger in size. We have added more structure to UNITY, by distinguishing between total and partial procedures, grouping the actions in cats, and allowing procedure calls among cats. Executing actions as indivisible units would extract a heavy penalty in performance; therefore, we have developed a theory that permits interleaved executions of the actions. Programs in UNITY interact by operating on a shared data space; Seuss cats, however, have no shared data and they interact through procedure calls only. As in UNITY, the issues of deadlock, starvation, progress (liveness), etc., can be treated by making assertions about the sequence of states in every execution. Also, as in UNITY, program termination is not a basic concept. A program has reached a *fixed point* when preconditions of all actions are *false* ; further execution of the program does not change its state then, and an implementation may terminate a program execution that reaches a fixed point. We have developed a simple logic for UNITY (for some recent developments, see [19], [18], [4]) that is applicable to Seuss as well.

A practical programming language needs (1) scope rules for names, (2) ways to combine programs written by different people, in particular, features for importing and exporting cats and boxes, and (3) features for a program to interact with programs written in other notations, particularly for input and output. These issues have been addressed in two implementations of Seuss on C++ and Java. The implementations support the building of libraries as cats.

Acknowledgment I am indebted to Rajeev Joshi and Will Adams for significant interactions in the development of Seuss.

References

[1] K. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, Reston, VA, 1968. AFIPS Press.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Mass., 1987.

[3] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, 1984.

[4] K. M. Chandy and B. A. Sanders. Towards Compositional Specifications for Parallel Programs. In *DIMACS Workshop on Specifications of Parallel Algorithms*, Princeton, NJ, May 9-11 1994.

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

[6] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, October 1967.

[7] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, 22(6):644–654, 1976.

[8] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[9] E. W. Dijkstra. Cooperating sequential processes. In F.Genuys, editor, *Programming Languages*, pages 43–112. Academic press, 1968.

[10] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[11] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug 1978.

[12] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, London, 1984.

[13] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[14] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, Sept. 1989.

[15] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag, 1991.

[16] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall PTR, Upper Saddle River, NJ, 1997.

[17] R. Milner. *Communication and Concurrency.* International Series in Computer Science, C. A. R. Hoare, Series Editor. Prentice-Hall International, London, 1989.

[18] Jayadev Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.

[19] Jayadev Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.

[20] R.L. Rivest, A.Shamir, and L. Adelman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[21] R. A. Scantlebury, K. A. Bartlett, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, May 1969.