Formal Methods in System Design, 5, 275 (1997)

A Simple, Object-based View of Multiprogramming

JAYADEV MISRA*
misra@cs.utexas.edu
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Received November 25, 1995; Revised March 30, 1996

Editor: Dominique Mery and Beverly Sanders

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. 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. We propose a model of multiprograms and a discipline of programming that addresses the issues of reasoning (e.g., understanding) and efficient implementation. The major point of departure is the disentanglement of sequential and multiprogramming features. We propose a sparse model of multiprograms that distinguishes these two forms of computations and allows their disciplined interactions.

Keywords: Multiprogramming, Concurrency, Object-based programming, Distributed Implementation, Semaphore, Communicating Processes.

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

Contributing Authors

Jayadev Misra Jayadev Misra is a professor and holder of the Regents' chair in the Department of Computer Sciences at the Univ. of Texas at Austin. He earned his Ph.D. in 1972 from the Johns Hopkins University. He has been a faculty member at the University of Texas at Austin since 1974, except for a sabbatical during 1983-1984 spent at Stanford University.

His research interests are in the area of concurrent programming, with emphasis on rigorous methods to improve the programming process. He has been the past editor of several journals in this area, including: Computing Surveys, Journal of the ACM, Information Processing Letters and the Formal Aspects of Computing. He has published extensively, and he is the co-author, with Mani Chandy, of a book, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

Misra is a fellow of ACM and IEEE; he held the Guggenheim fellowship during 1988-1989.

Formal Methods in System Design, 5, 277–275 (1997)

¡

Table of Contents: Volume 5 (1997)

Number 1

Numbers 2/3 (Special Issue on Computational Learning Theory)

Number 4

Formal Methods in System Design

An International Journal

Volume 5, No. 3, July 1997

1. Introduction

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. 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.

1.1. Motivation

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.

1.2. Seuss

We have developed a model of multiprogramming, called **Seuss**. 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.

A major point of departure in Seuss is that there is no built-in concurrency and no commitment to either shared memory or message-passing style of implementation. No specific communication or synchronization mechanism, except the procedure call, is built into the model. In particular, the notions of input/output and their complementary nature in rendezvous-based communication [9, 17] is outside this model. There is no distinction between computation and communication; process specifications and interface specifications are not distinguished. Consequently, we do not have many of the traditional multiprogramming concepts such as, processes, locking, rendezvous, waiting, interference and deadlock, as basic concepts in our 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.* Typically, several sequential programs will execute simultaneously; yet, we can guarantee that their executions would be non-interfering, and hence, each program may be regarded as atomic. 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 Programming Model

The Seuss programming model is sparse: a program is built out of *cat*s (*cat* is short for *category*) and *box*es, and a cat is built out of *procedure*s. A cat is similar in many ways to a process/class/monitor type; a cat denotes a type and a box is an instance of a cat. A 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. Cats are used to encode processes as well as the communication protocols for process interactions; therefore, it is necessary only to develop the methodology for programming and understanding cats and their component procedures.

We propose two distinct kinds of procedures, to model terminating and potentially non-terminating computations – representing computations of wait-free programs and multiprograms, respectively. The former 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. Multiprograms, however, cannot be given a pre- and post-condition semantic because on-going interaction with the environment is of the essence. We distinguish between these two types of computations by using two different kinds of procedures: a *total* procedure never waits (for an unbounded amount of time) to interact with its environment whereas a *partial* procedure may wait, possibly forever, for such interactions. In this view, a P operation on a semaphore is a partial procedure – because it may never terminate – whereas a V operation is a total procedure. A total procedure models wait-free, or *transformational*, aspects of programming and a partial procedure models concurrent, or *reactive*, aspects of programming[15]. Our programming model does not include *waiting* as a fundamental concept; therefore, a (partial) procedure does not wait, but it *rejects* the call, thus preserving the program state. We next elaborate on the main concepts: total and partial procedure, cat and program.

2.1. Total procedure

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) and 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) fail, or (3) execute forever. A failure is 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.

Non-termination of a total procedure is also the result of a programming error. We expect the programmer to establish that the procedure is invoked only in those states where its execution is finite.

2.2. Partial procedure

We first consider a simple form for a *partial* procedure, g:

$$g \ :: \ p; h \ \rightarrow \ S$$

where p is the *precondition*, h is the *preprocedure* and S is the *body* of the procedure. The precondition is a predicate on the state of the box to which g belongs. The preprocedure is the name of a partial procedure in another box; the preprocedure is optional. The body, S, consists of local computations affecting the state of g's box and calls on total procedures of other boxes. A partial procedure can call another partial procedure only as a preprocedure.

A partial procedure *accepts* or *rejects* each call made upon it. A partial procedure of the form $p \ \rightarrow \ S$, where the preprocedure is absent, accepts a call whenever p holds. A partial procedure, g, of the form $p; h \ \rightarrow \ S$ *accepts* a call if its precondition, p, holds and its preprocedure, h, accepts the call made by g (we will impose additional restrictions on the program structure so that this definition is well-founded). When a call is accepted, the body of the procedure, S, is executed, potentially changing the state of the box and returning computed values in the parameters. When a call is rejected, the procedure body is not executed and the state does not change. The caller is made aware of the outcome of the call in both cases; if the call made by g to h is rejected then the caller, g, also rejects the call made upon it, and if the call to h is accepted then g accepts the call and executes its body, S. In this sense, partial procedures differ fundamentally from the total ones because all calls are accepted in the latter case. Examples of partial procedures are P operation on a semaphore and a *get* operation on a print-queue performed by a printer; the call upon P is accepted only if the semaphore value is non-zero, and for the *get* if the print-queue is non-empty. Observe that whenever a call is rejected, the caller's state does not change, and whenever a call is accepted by a procedure that has the form $p; h \ \rightarrow \ S$, the body of the preprocedure, h, is executed before the execution of the procedure body, S.

We require that the execution of the body, S, terminate whenever the partial procedure, g, accepts a call.

Alternative Now, we introduce a generalization: the body of a partial procedure consists of one or more *alternatives* where each alternative is of the form described previously for partial procedures. Each alternative is *positive* or *negative*: the first alternative is positive; an alternative preceded by $|$ is positive and one preceded by $\not|$ is negative. The precondition of at most one alternative of a partial procedure holds in any state, i.e., the preconditions are pairwise disjoint.

The rule for execution of a partial method with alternatives is as follows. A partial-method *accepts* or *rejects* each call; it accepts a call if and only if one of its positive alternatives accepts the call, and it rejects the call otherwise. An alternative, positive or negative, accepts a call in a given state as follows. An alternative of the form $p \;\rightarrow\; S$ accepts the call if p holds; then its body, S, is executed and control is returned to its caller. An alternative of the form $p;\ h \;\rightarrow\; S$ accepts a call provided p holds and h accepts the call made by this procedure (using the same rules, since h is also a partial procedure); upon completion of the execution of h the body S is executed, and control is returned to the caller. Thus, an alternative rejects a call if the precondition does not hold, or if the preprocedure, provided it is present, rejects the call. Note that, since the precondition of at most one alternative of a partial procedure holds in a given state, at most one alternative will accept a call (if no alternative accepts the call, the call is rejected). It follows that the state of the caller's box is unchanged whenever a call is rejected, though the state of the called box may be changed because a negative alternative may have accepted the call.

Alternatives are essential for programming concurrent systems; negative alternatives are especially useful in coding strong semaphores, for instance.

2.3. method and action

A procedure is either a **method** or an **action**. An **action** is executed autonomously an infinite number of times during a (tight) program execution; see section 2.4.1. A **method** is not executed autonomously but only by being called from another procedure. Declaration of a procedure indicates if it is partial or total, and if it is an action or a method.

Example (Semaphore) A ubiquitous concept in multiprogramming is a semaphore.

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*}

A binary semaphore may be encoded similarly except that the V fails if $n \neq 0$ prior to its execution. Next, we show a small example employing semaphores. Let s, t be two instances of *semaphore*, declared by

box s, t : *semaphore*

Cat *user*, shown below, executes its critical section only if it holds both s and t, and it releases both semaphores upon completion of its critical section. The code for *user* dealing with accesses to s and t is shown below. Boolean variables hs and ht are *true* only when the *user* holds the semaphores s and t, respectively.

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 $\{user\}$

This solution permits acquiring s and t in arbitrary order. If it is necessary to acquire them in a specific order, say, first s and then t, the precondition of action $t.acquire$ should be changed to $hs \wedge \neg ht$.

2.4. Program

A program consists of a finite set of boxes (cat instances). We restrict the manner in which a procedure calls other procedures: all the procedures executing at any time belong to different boxes. We impose a condition, *Partial Order on Boxes*, below that ensures this restriction.

Definition: For procedures p, q, we write p *calls* q to mean that in some execution of p a call is made to q. Let $calls^+$ be the transitive closure of $calls$, and $calls^*$ the reflexive transitive closure of $calls$. Define a relation $calls_p$ over procedures where,
$(x\ calls_p\ y) \equiv (p\ calls^*\ x) \wedge (x\ calls\ y)$.

In operational terms, $x\ calls_p\ y$ means procedure x calls procedure y in some execution of procedure p. Each program is required to satisfy the following condition.

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.
Note: Observe that $\geq_p$ is reflexive and $>_p$ is irreflexive.

Observation 1:

$$p\ calls^*\ x \Rightarrow p.box \geq_p x.box$$
$$p\ calls^+\ x \Rightarrow p.box >_p x.box$$

It follows from Observation 1 that all the procedures that are part of a call-chain belong to different boxes.

Observation 2: $calls^+$ is an acyclic (i.e., irreflexive, asymmetric and transitive) relation over the procedures.

The definition of a program is in contrast to the usual views of process networks in which the processes communicate by messages or by sharing a common memory. Typically, such a network is not regarded as being partially ordered. For instance, suppose that process P sends messages over a channel chp to process Q

and Q sends over *chq* to P. The processes are viewed as nodes in a cycle where the edges (channels), *chp* and *chq*, are directed from P to Q and from Q to P, respectively, representing the direction of message flow. 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, *chp* and *chq* with the procedures in *chp*, *chq* being called from P and Q, respectively. The direction of message flow is immaterial in this hierarchy; what matters is that P, Q call upon *chp* and *chq* (though *chp* and *chq* do not call upon P, Q). 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 consists of one or more alternatives 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.

2.4.1. Program Execution We prescribe an execution style for programs, called *tight execution*. A tight execution consists of an infinite number of steps; in each step, an action of a box is chosen and executed. If that action calls upon a preprocedure that accepts the call, then the preprocedure is first executed followed by the execution of the action body. If the action calls upon a preprocedure that rejects the call then the state of the caller does not change. The choice of the action to execute in a step is arbitrary except for the following fairness constraint: each action of each box is chosen eventually.

A tight execution is easy to understand because execution of an action is completed before another action is started. 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 [19, 18], is suitable for deducing properties of a program in this execution model. Later, we show how a program may be implemented on multiple asynchronous processors with a fine grain of interleaving of actions that preserves the semantics of tight execution.

3. Small Examples

A number of small examples are treated in this section. 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. We show a number of variations of some of these examples, implementing various progress guarantees, for instance. For operational arguments about program behavior, we use tight executions of programs as defined in section 2.4.1.

3.1. Channels

Unbounded Channels 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 polymorphic version of the channel where the message type is left arbitrary. In the method *put*, we use : in the assignment to denote concatenation.

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

An instance of this cat may be interposed between a set of senders and a set of receivers.

Unordered Channels The fifo channel guarantees that the order of delivery of messages is the same as the order in which they arrived. 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 are used for bags in the following example.

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

This channel 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 every message is assigned an index, a natural number, and the variable t 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** $\{\}$ {initially b is empty},
> t: nat **init** 0, s: nat, m: type
>   **partial method** *get*(x: type)::
> $b \neq \{\} \;\rightarrow\;$ remove any pair (s, m) with minimum index, s, from b;
> $t, x := s, m$
>   **total method** *put*(x: type)::
> $b := b \cup \{(s, x)\}$, where s is a natural number strictly 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 $(i - t, p)$.

3.2. 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 N processes, called *readers*. We introduce a cat, *broadcast*, into which a writer writes the next value and from which a reader reads. The structure of the cat is as follows.

Internally, the value to be broadcast is stored in variable v; and n counts the number of readers that have read v. 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*. In the following definition, N is a parameter of the cat.

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

3.3. Barrier Synchronization

The problem and solution in this section are due to Rajeev Joshi[10]. In *barrier synchronization*, each process in a group of concurrently executing processes performs its computation in a sequence of stages. It is required that no process begin computing its $(k+1)^{th}$ stage until all processes have completed their k^{th} stage, $k \geq 0$. We propose a cat that includes a partial method, *sync*, that is to be called by each process in order to start computation of its next stage; the call is accepted only if all processes have completed the stage that this process has completed, and then the caller may advance to the next stage.

From the problem description, we see that at any point during the execution, all users have completed execution upto stage k and some users may be executing (or may have completed) stage $k+1$, for some k, $k \geq 0$. Initially, $k = 0$. As in the problem of *Broadcast*, each user has a boolean s, and *barrier* has a boolean t. We maintain the invariant that for any user $s = t$ means that the user has not yet entered stage $k+1$, and n is the number of ticket holders with $s = t$.

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

box *barrier*
  $n : 0..N$ **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 {*barrier*}

3.4. Readers and Writers

We consider the classic Readers Writers Problem [7] 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. The following solution includes two partial methods, *StartRead* and $StartWrite$, by which a reader and a writer gain access to the resource, respectively. Upon completion of their accesses, a reader releases the lock 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 in our solution that indicates the maximum number of readers permitted to have simultaneous access to the resource; N may be set arbitrarily high to permit simultaneous access for 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*}

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 as follows: A boolean variable, $WriteAttempt$, is set *true* , using a negative alternative, if a call upon $StartWrite$ is rejected. Once $WriteAttempt$ holds calls on $StartRead$ are rejected; thus no new readers are allowed to start reading. All readers will eventually stop reading – $n = N$ then – and the next call on $StartWrite$ will succeed.

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

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

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

We show that if $WriteAttempt$ is ever *true* it will eventually be falsified, asserting that a write operation will complete eventually, i.e., $EndWrite$ will be called. Similarly, if $ReadAttempt$ is ever *true* it will eventually be falsified. To prove the first result, consider the state in which $WriteAttempt$ is set *true* (note that initially $WriteAttempt$ 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 $WriteAttempt$ to *false* . If $WriteAttempt$ is set when a read is underway then no further call on $StartRead$ will be accepted and successive calls on $EndRead$ will eventually establish $n = N \land \neg ReadAttempt$, i.e., $n = N \land \neg ReadAttempt \land WriteAttempt$ 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 $ReadAttempt$ is similar. The precondition of the assignment $ReadAttempt := true$ 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) $ReadAttempt$ will be set *false.* In the latter case, upon completion of writing $EndWrite$ will be called and its execution will establish $n = N \land ReadAttempt \land \neg WriteAttempt$. No method other than $StartRead$ will execute in this state, and any reader that succeeds in executing $StartRead$ will eventually execute $EndRead$, thus falsifying $ReadAttempt$.

3.5. Semaphore

A binary semaphore, often called a *lock*, is typically associated with a resource. A process has exclusive access to a resource only when 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 some call on P 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 some variations.

We restrict ourselves to binary semaphores in all cases; extensions to general semaphores are straightforward.

3.5.1. Weak Semaphore The following cat describes a weak binary semaphore.

> **cat** *semaphore*
>   **var** *avail*: boolean **init** *true* {initially 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 associated with s; $s.V$
>   { other actions of the box}
> **end** {*user*}

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.

3.5.2. 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 implements a strong semaphore. The call upon P includes the process id as a parameter (pid is the type of process id). 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 pid **init** $\langle\rangle$, *avail*: boolean **init** *true*
>   {initially the semaphore is available}
>   **partial method** $P(i$: pid$)$ ::

$$avail \wedge i = q.head \;\rightarrow\; avail, q := false\ , tail.q$$
$$\not| \; i \notin q \;\rightarrow\; q := q : i$$
total method V:: $avail := true$
end $\{StrongSemaphore\}$

Observe the way a negative alternative is employed to record a caller's id while rejecting the call. The sequence q may be replaced by a fair bag, as was done for the unordered channel, *nch*.

Note: 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.

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 solution for the strong semaphore works 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[11]. A reasonable compromise is to add a new total method to the strong semaphore cat, which a transient caller may call to remove its process id from the queue of callers.

3.5.3. 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 the semaphore cat. 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, a 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 *SnoopySemaphore1*
  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 {*SnoopySemaphore1*}

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) ::
      $avail \wedge i = q.head \;\rightarrow\; avail, q := false\ , tail.q$
    $\not\|\ i \notin q \;\rightarrow\; q := q : i$
  total method V:: $avail := true$
  partial method S:: $q \neq \langle\rangle \;\rightarrow\; skip$
end {*StrongSnoopySemaphore*}

4. Distributed Implementation

We have thus far considered program executions where each action completes before another one is started. In section 2.4.1, we defined a *tight execution* of a Seuss program to be an infinite sequence of steps where each step consists of executing an action of a box. The choice of actions is arbitrary except that each action of each box is chosen eventually. This model of execution was chosen because it makes programming easier. Now, we consider another execution model, *loose execution*, where the executions of actions may be interleaved. A loose execution exploits the available concurrency. We restrict loose executions in such a manner that any loose execution may be simulated by a tight execution.

Crucial to loose executions is the notion of *compatibility* among actions: if a set of actions are pair-wise compatible then their executions are non-interfering, and their concurrent execution is equivalent to some serial execution of these actions. The precise definition of compatibility and the central theorem that establishes the correspondence between loose and tight executions are treated in section 4.4. We note that compatibility is a weaker notion than commutativity, and it holds for put, get over channels (see section 4.4), and for P, V operations on semaphores, for instance.

First, we describe a multiprocessor implementation in which the scheduler may initiate several compatible actions for concurrent executions. We also describe a "most general" scheduling strategy for this problem and implementations of the scheduling strategy on uniprocessors as well as multiprocessors. Then we define the notion of compatibility and state the fundamental Reduction Theorem that establishes a correspondence between loose and tight executions.

4.1. Outline of the Implementation Strategy

The implementation consists of (1) a *scheduler* that decides which action may next be scheduled for execution, and (2) *processors* that carry out the actual executions of the actions. The boxes of a program are partitioned among the processors. Each processor thus manages a set of boxes and it is responsible for executions of the actions of those boxes. The criterion for partitioning of boxes into processors is arbitrary though heuristics may be employed to minimize message transmissions among processors.

- The scheduler repeatedly chooses some action for execution. The choice is constrained by the requirement that only compatible procedures may be executed concurrently and by the fairness requirement. The scheduler sends a message to the corresponding processor to start execution of this action.

- A processor starts executing an action upon receiving a message from the scheduler. It may call upon methods of other processors by sending messages and waiting for responses. Each call includes values of procedure parameters, if any, as part of the message. It is guaranteed that each call elicits a response, which is either a *accept* or a *reject*. The accept response is sent when the call is accepted (which is always the case for calls upon total methods), and parameter values, if any, are returned with the response. A reject response is possible only for calls upon partial methods; no parameter values accompany such a response.

4.2. Design of the Scheduler

The following abstraction captures the essence of the scheduling problem. Given is a finite undirected graph; the graph need not be connected. Each vertex in the graph is *black* or *white*; all vertices are initially white. In this abstraction, a vertex denotes an action and a black vertex an executing action. Two vertices are neighbors if they are incompatible. We are given that

- (E) Every black vertex becomes white eventually (by the steps taken by an environment over which we have no control).

It is required to devise a coloring (scheduling) strategy so that

- (S1) No two neighbors are simultaneously black (i.e., only compatible actions may be executed simultaneously).
- (S2) Every vertex becomes black infinitely often (thus ensuring fairness).

Note that the scheduler can only blacken vertices; it may not whiten a vertex.

A simple scheduling strategy is to blacken a single vertex, wait until the environment whitens it, and then blacken another vertex. Such a strategy implements (S1) trivially because there is at most one black vertex at any time. (S2) may be ensured by blackening the vertices in some fixed, round-robin order. Such a protocol, however, defeats the goal of concurrent execution. So, we impose the additional requirement that the scheduling strategy be *maximal*: it should allow all valid concurrent executions of the actions; that is, any infinite sequence that satisfies (E,S1,S2) is a possible execution of our scheduler. A maximal scheduler is a most general scheduler, because any execution of another scheduler is a possible execution of the maximal scheduler. By suitable refinement of our maximal scheduler, we derive a centralized scheduler and a distributed scheduler. See [12] for a formal definition of the maximality condition.

A Scheduling Strategy Assign a natural number, called *height*, to each vertex; let $x.h$ denote the height of vertex x. We will maintain the invariant that neighbors have different heights:

Invariant D: $(\forall x, y : x, y \text{ are neighbors}: x.h \neq y.h)$

For vertex x, $x.low$ holds if the height of x is smaller than all of its neighbors, i.e., $x.low \equiv (\forall y : x, y \text{ are neighbors}: x.h < y.h)$. We write $v.black$ to denote that v is black in a given state. The scheduling strategy is:

- (C1) Consider each vertex, v, for blackening eventually; if $\neg v.black \wedge v.low$ holds then blacken v.
- (C2) Simultaneous with the whitening of a vertex v (by the environment), increase $v.h$ (while preserving the invariant D).

It is shown in [12] that this scheduling strategy satisfies (S1,S2) and, further, it is maximal in the sense described previously.

4.3. Implementation of the Scheduling Strategy

4.3.1. Central scheduler A central scheduler that implements the given strategy may operate as follows. The scheduler scans through the vertices and blackens a vertex v provided $v.low \wedge \neg v.black$ holds. The effect of blackening is to send a message to the appropriate processor specifying that the selected action may be executed. Upon termination of the execution of the action, a message is sent to the scheduler; the scheduler whitens the corresponding vertex and increases its height, ensuring that no two neighbors have the same height. The scheduler may scan the vertices in any order, but every vertex must be considered eventually, as required in (C1).

This implementation may be improved by maintaining a set, L, of vertices that are both white and low, i.e., L contains all vertices v for which $\neg v.black \wedge v.low$ holds. The scheduler blackens a vertex of L and removes it from L. Whenever a vertex x is whitened and its height increased, the scheduler checks x and all of its neighbors to determine if any of these vertices qualify for inclusion in L; if some vertex, y, qualifies then y is added to L. It has to be guaranteed that every vertex in L is eventually scanned and removed; one way is to keep L as a list in which additions are done at the rear and deletions from the front. Observe that once a vertex is in L it remains white and low until it is blackened.

4.3.2. Distributed scheduler The proposed scheduling strategy can be distributed so that each vertex blackens itself eventually if it is white and low. The vertices communicate by messages of a special form, called *token*. Associated with each edge (x, y) is a token. Each token has a *value* which is a positive integer; the value of token (x, y) is $|x.h - y.h|$. This token is held by either x or y, whichever has the smaller height.

It follows from the description above that a vertex that holds all incident tokens has a height that is smaller than all of its neighbors; if such a vertex is white, it may color itself black. A vertex, upon becoming white, increases its height by d, $d > 0$, effectively reducing the value of each incident token by d (note that such a vertex holds all its incident tokens, and, hence, it can alter their values). The quantity d should be different from all token values so that neighbors will not have the same height, i.e., no token value becomes zero, after a vertex's height is increased. If token (x, y)'s value becomes negative as a result of reducing it by d, indicating that the holder x now has greater height than y, then x resets the token value to its absolute value and sends the token to y.

Observe that the vertices need not query each other for their heights, because a token is eventually sent to a vertex of a lower height. Also, since the token value is the difference in heights between neighbors, it is possible to bound the token values whereas the vertex heights are unbounded over the course of the computation. Initially, token values have to be computed and the tokens have to be placed appropriately based on the heights of the vertices. There is no need to keep the vertex heights, explicitly, from then on.

We have left open the question of how a vertex's height is to be increased when it is whitened. The only requirement is that neighbors should never have the same height. A particularly interesting scheme is to increase a vertex's height beyond all its neighbors' heights whenever it is whitened; this amounts to sending all incident tokens to the neighbors when a vertex is whitened. Under this strategy, the token values are immaterial: a white vertex is blackened if it holds all incident tokens and upon being whitened, a vertex sends all incident tokens to the neighbors. Assuming that each edge (x, y) is directed from the token-holder x to y, the graph is initially acyclic, and each blackening and whitening move preserves the acyclicity. This is the strategy that was employed in solving the distributed dining philosophers problem in Chandy and Misra [4]; a black vertex is *eating* and a white vertex is *hungry*; the constraint (S1) amounts the well-known requirement that neighboring philosophers do not eat simultaneously. Our current problem has no counterpart of the thinking state, which added slight complication to the solution in [4]. The tokens are called *forks* in that solution.

As described in section 4.1, the actions (vertices) are partitioned among a group of processors. The distributed scheduling strategy has to be modified slightly, because the steps we have prescribed for the vertices are to be taken by the processors on behalf of their constituent actions. Message transmissions among the vertices at a processor can be simulated by simple manipulations of the data structures of that processor.

4.4. Compatibility

A loose execution of a program allows only compatible actions to be executed simultaneously. In this section, we give a definition of compatibility and state the Reduction Theorem, which says, in effect, that a loose execution may be simulated by a tight execution (in which executions of different actions are not interleaved). We expect the user to specify the compatibility relation for procedures within each box; then the compatibility relation among all procedures can be computed efficiently from the definition given below.

The states of a box are given by the values of its variables; the state of a program is given by its box states. With each procedure (partial and total) we associate a binary relation over program states. Informally, $(u, v) \in p$, for program states u, v and procedure p denotes that there is a tight execution of p that moves the state of the system from u to v. In the following, concatenation of procedure names corresponds to their relational product. For strings x, y, we write $x \subseteq y$ to denote that the relation corresponding to x is a subset of the relation corresponding to y.

Procedures p, q are *compatible*, denoted by $p \sim q$, if all of the following conditions hold. Observe that $\sim$ is a symmetric relation.

C0. p *calls* $p' \Rightarrow p' \sim q$, and q *calls* $q' \Rightarrow p \sim q'$.

C1. If p, q are in the same box,
(p is total $\Rightarrow qp \subseteq pq$), and
(q is total $\Rightarrow pq \subseteq qp$).

C2. If p, q are in different boxes, the transitive closure of the relation $(\geq_p \cup \geq_q)$ is a partial order over the boxes.

Condition C0 requires that procedures that are called by compatible procedures be compatible; this condition is well-grounded because, p *calls* $p' \Rightarrow p.box >_p p'.box$. Condition C1 says that for p, q in the same box, the effect of executing a partial procedure and then a total procedure can be simulated by executing them in the reverse order. Condition C2 says that compatible procedures impose similar (i.e., non-conflicting) partial orders on boxes.

Notes:
(1) For procedures with parameters, compatibility is checked with all possible values of parameters.
(2) Partial procedures of the same box are always compatible.
(3) Total procedures p, q of the same box are compatible provided $pq = qp$.

Example of Compatibility Consider the unbounded fifo channel of Section 3.1. We show that $get \sim put$, i.e., for any x, y, $get(x)\ put(y) \subseteq put(y)\ get(x)$. Note that the pair of states (u, v), where u represents the empty channel, does not belong to the relation $get(x)$.

$$\{r = a : S\}\ put(y)\ \{r = a : S : y\}\ get(x)\ \{x : r = a : S : y\}$$
$$\{r = a : S\}\ get(x)\ \{x : r = a : S\}\ put(y)\ \{x : r = a : S : y\}$$

The final states, given by the values of x and r, are identical.

The preceding argument shows that two procedures from different boxes that call *put* and *get* (i.e., a sender and a receiver) may execute concurrently. Further, since $get \sim get$ by definition, multiple receivers may also execute concurrently. However, it is not the case that $put \sim put$, that is,

$$put(x)\ put(y) \neq put(y)\ put(x)$$

because a fifo channel is a sequence, and appending a pair of items in different orders result in different sequences. Therefore, multiple senders may not execute concurrently.

Lemma 1: Let $p \sim q$ where p is total (p, q may not belong to the same box). Then, $qp \subseteq pq$.

This is the crucial lemma in establishing the Reduction Theorem, given below. The lemma permits a total procedure p to be moved left over any other procedure q provided p, q are compatible. This strategy can be employed to bring all the components of a single procedure together, thereby converting a loose execution to a tight execution. Observe that the resulting tight execution establishes identical final state starting from the same initial state as the original loose execution. Therefore, properties of loose executions may be derived from those of the tight executions. For a proof of the following theorem, see chapter 10 of [20].

Reduction Theorem: Let E denote a finite loose execution of some set of actions. There exists a tight execution, F, of those actions such that $E \subseteq F$.

5. Concluding Remarks

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.

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. The Seuss view that *all* multiprogramming can be regarded as (1) coding of the action-bodies, and (2) specifying the conditions under which each action-body is to be executed, differs markedly from the conventional view; we consider and justify some of these differences below.

First, Seuss insists that a program execution be understood by a single thread of control, avoiding interleaved executions of the action-bodies, because it is simpler to understand a single thread and formalize this understanding within a logic. An implementation, however, need not be restricted to a single thread as long as it achieves the same effect as a single-thread execution. We will show how implementations may exploit the structures of Seuss programs (and user supplied directives) to run concurrent threads. A consequence of having a single thread 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 [9, 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. For instance, a call upon a P operation on a semaphore (which could cause the caller to wait) is now replaced by the call being rejected if the semaphore is not in the appropriate state; the caller then attempts the call repeatedly during the ensuing execution.

Second, 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 *multiplex* and the *database*). 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.

Third, 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. Separation of sequential and multiprogramming features has also been advocated in Browne et. al. [3].

Fourth, 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 abstractions. 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.

Our work incorporates ideas from serializability and atomicity in databases[2], notions of objects and inheritance[16], Communicating Sequential Processes[9], 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, 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[9]. 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, and imposing a hierarchy over the cats. Executing actions as indivisible units would extract a heavy penalty in performance; therefore, we have developed the 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. In a sense, cats may only share cats. 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], [6]) that is applicable to Seuss as well.

Acknowledgments: I am profoundly grateful to Rajeev Joshi who has provided a number of ideas leading up to the formulation of the concept of compatibility and the Reduction Theorem. Many of the ideas in this paper were developed after discussions with Lorenzo Alvisi, Will Adams and Calvin Lin. I am also indebted to the participants at the Marktoberdorf Summer School of 1998, particularly Tony Hoare, for interactions.

References

1. Kenneth 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, 1987.
3. J.C. Browne, A. Tripathi, S. Fedek, A. Adiga, and R. Kapur. A language for specification and programming of reconfigurable parallel computation structures. In *Int. Conf. of Parallel Processing, Bellaire, Michigan*, pages 142–149. IEEE, August 1982.
4. K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, 1984.
5. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
6. K. Mani Chandy and Beverly A. Sanders. Towards Compositional Specifications for Parallel Programs. In *DIMACS Workshop on Specifications of Parallel Algorithms*, Princeton, NJ, May 1994.
7. 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.
8. E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
9. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1984.
10. Rajeev Joshi. Personal communication. 1996.
11. Rajeev Joshi and Jayadev Misra. On the impossibility of robust solutions for fair resource allocation. Technical Report CS-TR-99-14, University of Texas at Austin, Department of Computer Sciences, June 1999. Available for download at `ftp://ftp.cs.utexas.edu/pub/techreports/tr99-14.ps.Z`.
12. Rajeev Joshi and Jayadev Misra. Maximally concurrent programs. *Formal Aspects of Computing*, 12(2):100–119, 2000.
13. Leslie 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, September 1989.
15. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification*. Springer-Verlag, 1992.
16. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.
17. R. Milner. *Communication and Concurrency*. International Series in Computer Science, C.A.R. Hoare, series editor. Prentice-Hall International, 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. Jayadev Misra. *A Discipline of Multiprogramming*. Monographs in Computer Science. Springer-Verlag, 2001. The first chapter is available at `http://www.cs.utexas.edu/users/psp/discipline.ps.gz`.
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.