Semantic Models for Parallel Systems

Ellis S. Cohen
Department of Computer Science
Carnegie Mellon University
January 1975

ABSTRACT

This paper presents a semantic model for parallel systems with a scheduling mechanism that is useful for expressing and proving a wider range of properties than semantic models which do not consider scheduling.

We formally describe a number of properties related to scheduling and deadlock, including "Fairness" and "Fullness", and show that schedulers with these properties behave in desireable ways.

Lastly, we prove and conjecture some proof rules for scheduled systems and outline briefly the relation of this work to modelling protection in parallel systems.

This work was supported by the Advanced Research Projects Agency of the Department of Defense under contract no. F44620-73-C-0074 and monitored by the Air Force Office of Scientific Research.

Semantic Models for Parallel Systems

INTRODUCTION

Based on Scott's Mathematical Theory of Computation [Scott 72], Cadiou & Levy [Cadiou & Levy 73] and Milner [Milner 73] have introduced a model of parallel processes based on processes that communicate by sharing memory, and have shown how to state and prove properties such as mutual exclusion formally within the mechanizable LCF system.

They treat nondeterminism by introducing an oracle from the domain TT* (sequence of truth values, see [Kahn 73]). The determination of which process to execute next depends on an initial sequence of the oracle, with the new oracle becoming the remainder.

In spite of the elegance of their system, they are unable to prove certain properties of parallel systems that one would expect to be true. Primarily this trouble stems from the difficulty of characterizing the well-behavedness of their oracle. By using a model derived from Lipton's work [Lipton 73], we replace the oracle with a scheduler and state a property of schedulers, fairness, which is shown to be adequate to prove a property of a particular parallel system that is difficult to express in Cadiou & Levy's system.

We first present a variation of Cadiou & Levy's model and note some of its problems. We then introduce a model with a scheduling formalism that solves these difficulties. The remainder of the paper contains properties and proofs using the scheduling model, as well as additional comments.

MODELS FOR PARALLEL PROCESSES

The models for parallel processes we will invetigate in this paper have 3 important features.

1) Processes - We will always consider a variable number of processes, each of which may be in one of three states, runnable, blocked or stopped.

2) Indivisibility - Processes are divided into indivisible actions (instructions) called elementary processes or EP's. When a process is selected to run, it executes exactly one EP, after which a new decision is made about which process should be scheduled. Concurrent execution of parallel processes is modelled by sequential interleaving of actions from the various processes.

3) Abstract Machine - Two main approaches have emerged for proving general properties about programs (i.e. - Termination and Equivalence as well as Correctness), the Functional approach [Scott & Strachey 73] (related is the Relational approach, see [deBakker 74]) and the Abstract

Machine approach [Wegner 72].

The Functional approach maps a program directly into a mathematical function; the meaning of a program is then just the value of the corresponding function. Not only is the technique elegant, but a formal system, LCF (Logic for Computable Functions) [Milner 72] has been developed and mechanized in which one can prove properties about computable functions. Cadiou & Levy and Milner use such an approach in their respective papers on semantics of parallel programs.

The Abstract Machine approach defines a programming system via a formal definition of an abstract machine. The meaning of a program is then the result of its execution on their abstract machine. Much of what might be considered inelegant about this technique is due to its awkwardness in modelling the execution of statements with complex control structures.

However, in the parallel systems we will be describing, there is only one language construct, the EP. We are thus in the unusual position of being able to produce an abstract machine definition that is as simple and somewhat less opaque than the corresponding functional semantics.

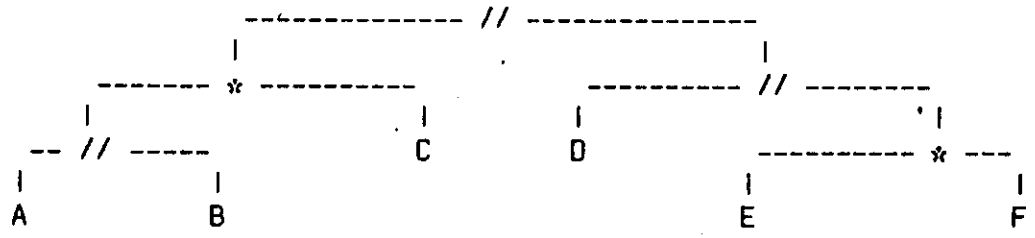
Of course, one question remains - how to define the Abstract Machine. We choose to define the machine interpreter as a computable function, thus making the tools of LCF available for our proofs.

(As we note in the conclusion, we expect work on semantics for parallel systems to come full circle, that is, back to languages that have the appropriate structures for parallel control. It is likely that an Abstract Machine approach would then be unsuitable.)

A VARIANT OF CADIOU & LEVY'S MODEL

In producing an Abstract Machine version of Cadiou & Levy's model, we divide the state of the model into 2 parts, S, the Data state and K, the Control state.

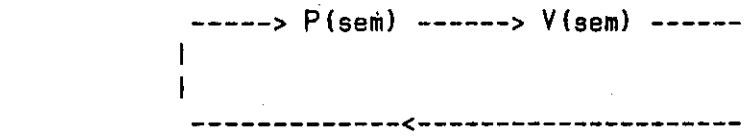
The Control state, K, can be viewed as a binary process tree whose leaf nodes represent processes. The interior nodes of the tree contain either "//" which indicates parallel execution of its two subtrees or "*" which indicates sequential execution, that is, no process in the right subtree can run until all processes in the left subtree have stopped. For example:



A, B, D and E are runnable. C is blocked until both A & B stop. F is blocked until E stops.

The Abstract Machine selects a leaf node representing a runnable process. It executes a single EP which first modifies the state S, and then produces a process tree which replaces the node selected, thus becoming a subtree of K. The subtree may be simply a single node, which can be used to represent the continuation of the same process, a "*" construct, which can be used to represent the call of a subroutine, or a "//" construct, which can be used to represent the spawning of a subprocess. In addition, a node can be the element {STOP} which indicates the process has stopped.

All processes execute the same program. We can view programs as labelled flowcharts, where it is the EPs that are labelled. For example, the flowchart

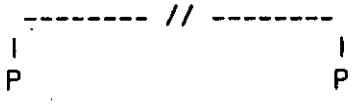


can be represented by the following program with labels P & V.

```
P: sem > 0 --> ( sem ← sem - 1  ==> V ),  ==> P
V: sem ← sem + 1  ==> P
```

(Note: Read "==>" as "goto" and "a --> b, c" as "if a then b else c")

The leaf nodes of K either contain STOP or the label of the EP the process was executing. So, the process tree for a system in which two processes are executing the P/V loop program above might be



The data state S contains an element sem.

In the formal model, the abstract machine, given S and K determines the "Next" state of S and K by selecting a runnable node from K and

executing the EP it represents thus changing both S and K.

To select the runnable EP, we use an oracle, an infinite sequence of truth values. We start at the root of K and work our way towards a leaf node. Each time we encounter a "//" with runnable nodes (not {STOP}) in each subtree, we pick off the first element of the oracle and use it to decide which subtree to continue down. In the formal model, the "Next" function implements the abstract machine as as recursive tree-walk.

FORMAL MODEL - Cadiou & Levy Adaptation

Primitive Domains

S - memory state
TT - truth value (elements tt, ff and uu -
· we also use "uu" to represent the least defined element of any domain and let the user rely on context to determine the appropriate domain)
LABEL - label

Constructed Domains

ORACLE = TT* (sequence of truth values)
EP = S --> K × S
K = {STOP} + LABEL + K × {*,//} × K
PROG = LABEL --> EP

The "Next" function uses the oracle to pick a runnable EP from K, returning the resulting process tree as well as the updated state and the remainder of the oracle.

```
Next: K x S x ORACLE --> K x S x ORACLE

Next(k,s,ora) <==

  Case k of

    STOP --> <k,s,ora>,

    <q,//,r> -->  (
         Stop(q) --> Next(r,s,ora),
         Stop(r) --> Next(q,s,ora),
         Hd(ora) --> Mk( λt.<t,//,r>, Next(q,s,Tl(ora)) ),
                     Mk( λt.<q,//,r>, Next(r,s,Tl(ora)) )  ),

    <q,*,r> -->  (
         Stop(q) --> Next(r,s,ora),
                     Mk( λt.<t,*,r>, Next(q,s,ora) )  ),

    lbl -->  <Exec(lbl)(s).K, Exec(lbl)(s).S, ora>.
```

(note that if AB = A x B, and ab: AB (ab is of type AB), then we use ab.A and ab.B to indicate the projections of ab onto it's A and B components respectively)

The "Exec" function for a particular program Prog gets the EP labelled by lbl and executes it in state s to produce a new k and s.

```
Exec: LABEL --> [ S --> K x S ]

Exec(lbl)(s) <== Prog(lbl)(s).


Mk: [ K --> K ] x [ K x S x ORACLE ] --> [ K x S x ORACLE ]

Mk(fk,<k,s,ora>) <== <fk(k),s,ora>.


Hd: TT* --> TT     and returns the first element of a sequence
Tl: TT* --> TT*    and returns the remainder of a sequence

Stop: K --> TT  and is defined so that
Stop(uu) = uu,  Stop(STOP) = tt, and for all other k,
Stop(k) = ff.
```

The result (final state) of running k0 with an initial state s0 and oracle ora0 is Mem(k0,s0,ora0), where Mem is

```
Mem(k,s,ora) <==
  Stop(k) --> s,
  Mem(Next(k,s,ora)).
```

(An alternate model perhaps closer to current languages and systems might use "&" instead of "//", where "&" spawns a totally independent process. Thus in <<p,//,q>,*,r>, r can only execute after both p and q STOP. In <<p,&,q>,*,r> r can execute after p STOPs, regardless of what happens to q. And, <<STOP,&,q>,*,r> would act like <r,&,q> if a semantic description were to be given. However, we will not pursue it further in this paper.)

The key departure from Cadiou & Levy is that K is represented by a "syntactic" data structure rather than by being embedded in a purely functional structure and "//" and "*" are used here as purely syntactic entities rather than as instances of more general process combinators. A number of other changes have been made to produce an Abstract Machine model from their functional model, but none significantly affect the problems of the model.

The main advantage of the adaptation has been that we have separated the selection of a process to be executed from its execution. This suggests the substitution of a scheduler for the oracle.

FACTORS IN CHOOSING A MODEL

There are three major concerns that have prompted the development of the scheduling model that will be the focus of the rest of the paper.

1) It is difficult (at best) to characterize anomalous oracles, since anomaly depends so heavily on the changing nature of the state and control. For example, in the 2 process P/V loop example, Cadiou & Levy are only able to prove that one or the other will run forever, while under a reasonably "fair" scheduler, we would expect both to run forever. By providing a model with a scheduler, we can characterize the scheduler in such a way that anomalous schedules can be avoided. Thus, we will replace the Oracle by a Scheduler which has access to the state of the system and specifies a particular process to be run as well as producing a new scheduler to schedule the next process (presumably by modifying internal variables or queues).

2) We wish to model situations where one process may arbitrarily start, stop or otherwise control another process. Thus, instead of K, the model contains a multiplexor M, which may viewed as a vector of processes. The Scheduler specifies a process to be run by supplying an integer index into M. M is also more general than K in that for each process we associate not only a label indicating the current control

point, but a separate program as well.

3) We wish to characterize processes which are blocked, so that the scheduler can choose not to attempt to run such a process. Thus, following Lipton [Lipton 73], we provide each EP with a synchronization part which can be used to determine which processes are blocked.

An EP consists of 3 parts, all executed indivisibly of course. The first part, (SYNCHFORM), represents a synchronization condition. If the Scheduler schedules a process, and the synchronization condition of its current EP is not met, no action is taken, and the Scheduler is simply invoked to schedule again. If the synchronization condition is met, the other 2 parts of the EP are executed. One part (STATEFORM) changes the data state (S) of the system, and one part (CONTROLFORM) changes the control state (M) of the system (specifying the label of the next EP of the current process or starting, stopping or otherwise controlling another process. There is one special label, STOP, which denotes the completion of a process).

Evaluation of "Next" proceeds in the following way: First the Scheduler produces an index into the Multiplexor (as well as a new Scheduler to schedule the next iteration). If the label indexed is "STOP", then no further action is taken this iteration. Otherwise, the labelled EP is executed. First its synchronization condition is tested. If false, no further action takes place with the EP. If true though, the rest of the EP is evaluated to update both the data state (S) and the multiplexor (M).

THE FORMAL MODEL

Primitive Domains

TT - truth values
N - natural numbers
LABEL - labels, including the element STOP
ARG - function argument
NAME - names of functions
S - states

Constructed Domains

```
SYNCHFORM = NAME x ARGS
STATEFORM = NAME x ARGS
CONTROLFORM = NAME x ARGS
EP = SYNCHFORM x STATEFORM x CONTROLFORM
M = N --> PROG x LABEL
PROG = LABEL --> EP
ARGS = {<>}  +  ARG x ARGS    (We will use standard tuple notation
   and thus represent <a,<b,<c,<>>>> as <a,b,c>)
SM = S x M
```

The Scheduler

```
SCHED = S x M --> N x SCHED
```

Primitive Functions

```
Synchfn: NAME --> [ ARGS --> [ S --> TT ] ]
Statefn: NAME --> [ ARGS --> [ S --> S x ARGS ] ]
Controlfn: NAME --> [ ARGS --> [ ARGS --> [ M --> M ] ] ]
```

For reasons discussed in the section on Scheduler Notes, we model the various FORMs as a function name and an argument list. To evaluate the function, we must provide a way of mapping the name of the function to the function itself. That is what the three primitive functions do. They are also guaranteed to be total. It is left to the reader to imagine how they can be extended reasonably to total functions in the cases where the name is undefined or the arguments are inappropriate. It is important to note that arguments to Synchfn's and Statefn's will not necessarily be values but will more likely represent variable names used to select a value from s. Thus we are not providing an abstract model of storage, but rather modelling at a higher level of abstraction.

The Interpreter

```
Next: S x M x SCHED --> S x M x SCHED

Next(s,m,sched) <==
  Let <n,sched'> be sched(s,m) in
    m(n).LABEL = STOP --> <s,m,sched'>,
    Let <s',m'> be Exec(n)(s,m) in <s',m',sched'>.
```

(note that if AB = A x B, and ab: AB (ab is of type AB), then we use ab.A and ab.B to indicate the projections of ab onto it's A and B components respectively)

Given an index into the multiplexor and a multiplexor, Action produces the designated EP.

```
Action: N x M --> EP

Action(n)(m) <== (m(n).PROG)(m(n).LABEL).
```

Given an index into M, as well as S & M, Exec executes the designated EP to produce a new S & M.

```
Exec: N --> [ S x M --> S x M ]

Exec(n)(s,m) <==
  Let <syfrm,stfrm,cfrm> be Action(n)(m) in
    Synchfn(syfrm.NAME)(syfrm.ARGS)(s) --> (
      Let <s',result> be Statefn(stfrm.NAME)(stfrm.ARGS)(s) in
        <s',Controlfn(cfrm.NAME)(cfrm.ARGS)(result)(m)> ),
    <s,m>.
```

The reader is encouraged to look ahead to the Applications section for an example of how a particular system would be modelled.

In this model (as in actual systems), it is not so clear when computation stops (for example, an idle process may run in an Operating system when nothing can otherwise be scheduled). However, for simplicity, we will assume a continuous predicate, Mstop.

```
Mstop: S x M x SCHED --> TT
```

For example, if the scheduler returns a zero index when there is nothing to schedule, then we could define Mstop as:

```
Mstop(s,m,sched) <==  ( sched(s,m).N = 0 ).
```

In any case, we can define the result (final state) of running m0 with state s0 and scheduler sched0 as Mmem(s0,m0,sched0) where Mmem is defined as

```
Mmem(s,m,sched) <==
  Mstop(s,m,sched) --> s,
  Mmem(Next(s,m,sched)).
```

PROPERTIES OF SCHEDULERS

Treatment of schedulers in this paper will be independent of any particular synchronization primitives (e.g. P/V, P/Vchunk, up/down) and any particular implementation or internal structure of the scheduler (e.g. FIFO queues, priority order), rather we simply express a number of scheduler properties using the model. The properties described are either ones that will be used later in the paper, or ones that have appeared already in the literature. A comparison of these properties by example can be found in the Applications section of this paper.

The properties as described are dependent heavily on S & M as well as the scheduler, whereas commonly, we are simply interested in a property of a scheduler independent of what it schedules. The section of this paper on Scheduler Notes indicates how this problem may be solved.

Notes: We will be using "process j" to indicate the continuing behavior of the contents of M(j).

We use the notation $\sqsubseteq$ to mean less defined than - also
- $\equiv$ - Strong equivalence ($a \equiv b$ iff $a \sqsubseteq b \ \wedge \ b \sqsubseteq a$)
- $\sqsubset$ - Strictly less defined than ($a \sqsubset b$ iff $a \sqsubseteq b \ \wedge \ \neg (a \equiv b)$)

Note that sequence domains (e.g. TT$\star$) are ordered by
$uu \sqsubseteq a \sqsubseteq (a \# b)$ and $a \equiv a \# uu$
where "#" is the concatenation operator.

1) Defined(sched)(s,m)

tt$\star$ <== tt # tt$\star$. (The symbol "tt$\star$" is to be the least fixed point of this equation - which can be seen to be the infinite string of "tt"s.)

Def(s,m,sched) <== tt # Def(Next(s,m,sched)).

Defined(sched)(s,m) iff Def(s,m,sched) $\equiv$ tt$\star$

2) Full(sched)(s,m) - A scheduler is full if it does not schedule an unrunnable process when a runnable process can be run.

```
Canrun(k)(s,m) <==
  m(k).LABEL = STOP --> ff,
  ( Let syn be Action(k)(m).SYNCHFORM in
      Synchfn(syn.NAME)(syn.ARGS)(s) ).
```

```
Runnable(j,k)(s,m,sched) <==
  ( j ¬= sched(s,m).N ∨ Canrun(j)(s,m) ∨ ¬Canrun(k)(s,m) )
  # Runnable(j,k)(Next(s,m,sched)).

Full(sched)(s,m) iff (∀j,k)( Runnable(j,k)(s,m,sched) ⊑ tt* )
```

3) Release(sched)(s,m) - A scheduler is a release scheduler [Lipton 73] if, when some action unblocks a set of processes, then some process from that set will be the next to run.

```
Unblock(k)(s,m,sched) <==
  Let <s',m',sched'> be Next(s,m,sched) in
    ( Canrun(k)(s,m) --> tt,
      Canrun(k)(s',m') --> (
        Let n' be sched'(s',m').N in
          n' = k --> tt,
          ¬Canrun(n')(s,m) ∧ Canrun(n')(s',m') ),
      tt )
    # Unblock(k)(s',m',sched').

Release(sched)(s,m) iff (∀k)( Unblock(k)(s,m,sched) ⊑ tt* )
```

4) Ready\Run(sched)(s,m) - A scheduler has the Ready Run property when no process has to wait forever to run from the time it becomes continuously capable of running. We actually state this in the logic as - any process which is unable to run at most a finite number of times must run infinitely often. Some thought should convince the reader that these are the same.

```
Run(j)(s,m,sched) <==
  ↑( j = sched(s,m).N ∧ Canrun(j)(s,m) )
  # Run(j)(Next(s,m,sched)).

↑(p) <== p --> tt, uu.

Cantrun(j)(s,m,sched) <==
  ↑(¬Canrun(j)(s,m)) # Cantrun(j)(Next(s,m,sched)).

Ready\Run(sched)(s,m) iff
  (∀j)( Cantrun(j)(s,m,sched) ⊑ tt* ⊃ Run(j)(s,m,sched) = tt* )
```

5) Pointer\Bounded(sched)(s,m) - A scheduler is pointer bounded [Lipton 73] when a process able to run infinitely often is scheduled infinitely often. (We will see in the Application section that both Ready\Run and Pointer\Bounded are too weak and that Fairness is a more appropriate property)

```
Tried(k)(s,m,sched) <==
  ↑( k = sched(s,m).N ) # Tried(k)(Next(s,m,sched)).

Infcan(k)(s,m,sched) <==
    ↑(Canrun(k)(s,m)) # Infcan(k)(Next(s,m,sched)).

Pointer\Bounded(sched)(s,m) iff
  (∀k)( Infcan(k)(s,m,sched) ≡ tt* ⊃ Tried(k)(s,m,sched) ≡ tt* )
```

6) Fair(sched)(s,m) - A scheduler is fair if any process able to run infinitely often, runs infinitely often at times that it canrun (is not blocked or stop)

```
Fair(sched)(s,m) iff
  (∀k)( Infcan(k)(s,m,sched) ≡ tt* ⊃ Run(k)(s,m,sched) ≡ tt* )
```

7) We say a scheduler sched' is an idling extension of sched if

a) (sched(s,m) ≡ uu ∧ (∀k)(¬Canrun(k)(s,m))) -->
sched'(s,m).N = 0,
sched'(s,m).N ≡ sched(s,m).N

b) sched'(s,m).SCHED is an idling extension of sched(s,m).SCHED

This corresponds nicely with the example definition of Mstop in the previous section. It is easily provable that every scheduler has an idling extension, that Run(j)(s,m,sched) ≡ Run(j)(s,m,sched') and Defined(sched')(s,m). Also Full(sched)(s,m) ⊢ Full(sched')(s,m) and similarly for Fair.

Fairness is in general the weakest property (along with definedness) that we would ever demand of a legitimate actual scheduler. Luckily, fairness (with definedness) will be adequate for proving properties that we are interested in. However, proving certain properties (in particular, the example proven in the next section) given fairness alone turns out to be somewhat difficult. The key problem is knowing exactly when a particular action will occur, even when it is known that it must occur eventually. This problem often disappears if the scheduler is full as well. So we will show that to prove:

A] Defined(sched)(s,m), Fair(sched)(s,m), Q(j,s,m) ⊢
Run(j)(s,m,sched) ≡ tt*

it is sufficient to show that

B] Defined(sched)(s,m), Fair(sched)(s,m), Full(sched)(s,m), Q(j,s,m) ⊢
Infcan(j)(s,m,sched) ≡ tt*

Proof:

Suppose there were a function Fullsched: SCHED --> SCHED s.t. for any scheduler sched,

1) Full(Fullsched(sched))(s,m)
2) Run(j)(s,m,Fullsched(sched)) ≡ Run(j)(s,m,sched)
3) Infcan(j)(s,m,Fullsched(sched)) ⊑ Infcan(j)(s,m,sched)

Now, suppose Defined(sched)(s,m), Fair(sched)(s,m), Q(j,s,m), but Run(j)(s,m,sched) ⊑ tt*

Since Fair(sched)(s,m), Infcan(j)(s,m,sched) ⊑ tt*

Thus by (1), (2) and (3),
Full(Fullsched(sched))(s,m),
Run(j)(s,m,Fullsched(sched)) ⊑ tt* and
Infcan(j)(s,m,Fullsched(sched)) ⊑ tt*

Then trivially, Fair(Fullsched(sched))(s,m), by defn of Fair

Now, let fsched be an idling extension of Fullsched(sched). Then Defined(fsched)(s,m), Fair(fsched)(s,m), Full(fsched)(s,m) and Run(j)(s,m,fsched) ⊑ tt*

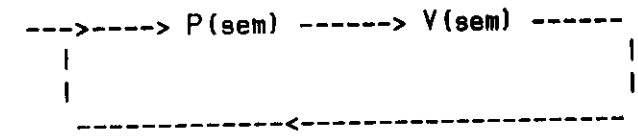
If we can prove [B], then Infcan(j)(s,m,fsched) ≡ tt*, and by defn of Fair, Run(j)(s,m,fsched) ≡ tt*.

Thus, we have a contradiction to Run(j)(s,m,fsched) ⊑ tt*, and therefore the original hypothesis that Run(j)(s,m,sched) ⊑ tt* must be false. Since it is easily shown that Run(j)(s,m,sched) ⊑ tt*, it must be the case that Run(j)(s,m,sched) ≡ tt* and [A] follows.

Definition of Fullsched and proofs of 1), 2) and 3) can be found in the Appendix.

APPLICATIONS

Some notion of the properties in the section above can be gained by consideration of the example (adapted from [Lipton 72]) of 3 processes, each executing the loop:



where the initial value of sem is 1.

(We will describe execution sequence as a sequence of pi and vi, i=1,2,3 to denote the execute of a P or V by the i'th process)

Under a scheduler that is merely defined and full, the execution could simply be

p1 v1 p1 v1 p1 v1 p1 v1 ...

that is, processes 2 and 3 might never execute.

If the scheduler is additionally a Release scheduler, the execution could be

p1 v1 p2 v2 p1 v1 p2 v2 p1 v1 p2 v2 ...

that is, v1 releases P of processs 2 and v2 releases p1, but again process 3 might never be executed.

If the scheduler additionally has the Ready\Run property, it helps matters not at all, since process 3 is never continuously capable of running. It is blocked each time process 1 or 2 executes a P. Likewise the Pointer\Bounded property does not help, since process 3 might only be tried when it is blocked.

If the scheduler though is merely defined and fair, each of p1, p2, p3, v1, v2 and v3 must execute infinitely often.

We'll prove that last statement for the more general case where there are n processes. As already noted, this is a problem that Cadiou & Levy would have difficulty proving.

To simplify, we'll assume that the state s is identically sem, and we'll define the following functions:

```
true()(s) <== tt.
tst()(s) <== ( s > 0 ).
inc()(s) <== <s+1,uu>.
dec()(s) <== <s-1,uu>.
go(<n,lbl>)(res)(m) <== λk.( k = n --> <m(n).PROG,lbl>, m(k) ).
```

Introducing some notation, we use

lbl: When syf(sya) do stf(sta) ==> cf(ca)

to represent the EP

<<syf,sya>,<stf,sta>,<cf,ca>>

where the EP is labelled by "lbl". Where sya, sta or ca are <> (no arguments), we eliminate parentheses as well. We further use the notation

:n=> lbl(args) for ==> GO(n,lbl,args)

(note: Function definitions, like "go", have their names in lower case. The formal name, like "GO" (from the domain NAME) is the same name written in upper case.)

So, we name the program described pictorially above, pvloop[j], where j is the process number (index into M). It has two labels, P & V, and its formal description using the shorthand notation developed above is:

P: When TST do DEC :j=> V
V: When TRUE do INC :j=> P

Now, the problem can be stated in the logic as, Prove:

Defined(sched0)(s0,m0), Fair(sched0)(s0,m0), Range(j) ⊢
Run(j)(s0,m0,sched0) ≡ tt*

where

m0 <== λj.(Range(j) --> <pvloop[j],P>, <uu,STOP>).
s0 <== 1.
Range(j) <== j ≥ 1 ∧ j ≤ n.

By the results of the previous section, we can also assume that Full(sched0)(s0,m0) and simply prove Infcan(j)(s0,m0,sched0) ≡ tt*.

PROOF:

Defined(sched0)(s0,m0), Fair(sched0)(s0,m0),
Full(sched0)(s0,m0), Range(j) ⊢
Infcan(j)(s0,m0,sched0) ≡ tt*

Infcan2k(j)(s,m,sched)(k) <==
↑(Canrun(j)(Desc(s,m,sched)(2*k).SM))
↑(Canrun(j)(Desc(s,m,sched)(2*k+1).SM))
Infcan2k(j)(s,m,sched)(k+1).

LEMMA 1
Defined(sched)(s,m) ∧ Full(sched)(s,m) ⊃
(∀k)(Let <s',m',sched'> be Desc(s,m,sched)(k) in
Defined(sched')(s',m') ∧ Full(sched')(s',m'))
Proof: Math Ind on k

LEMMA 2

Infcan2k(j)(s,m,sched)(k) ≡ Infcan(j)(Desc(s,m,sched)(2*k))

Proof: Parallel Comp Ind on Infcan2k & Infcan

LEMMA 3

Defined(sched0)(s0,m0), Full(sched0)(s0,m0) ⊢
Let <s',m',sched'> be Desc(s0,m0,sched0)(2*k),
<s",m",sched"> be Desc(s0,m0,sched0)(2*k+1),
j be sched'(s',m') in
Range(i) ⊃ Canrun(i)(s',m') ∧
Canrun(j)(s",m") ∧
i ≠ j ⊃ ¬Canrun(i)(s",m")

Proof: Math Ind on k using Lemma 1

LEMMA 3a

Defined(sched0)(s0,m0), Full(sched0)(s0,m0), Range(j) ⊢
Canrun(j)(Desc(s0,m0,sched0)(2*k).SM) ≡ tt

The proof of the theorem follows directly from Lemmas 2 & 3a

We can also state (though we will not prove) the mutual exclusion problem as

Range(j), Range(k), j≠k ⊢ Mutex(s0,m0,sched0) ≡ uu

Mutex(j,k)(s,m,sched) <==
↑(m(j).LABEL = m(k).LABEL = V) # Mutex(j,k)(Next(s,m,sched)).

DEADLOCK

Briefly, we can state some deadlock properties in the logic based on the model.

1) Starved(k)(sched)(s,m) - A process is starved [Dijkstra 72] if it is not "STOP" and is continuously incapable of running.

Infsafe(k)(s,m,sched) <==
↑(m(k).LABEL = STOP or Canrun(k)(s,m))
Infsafe(k)(Next(s,m,sched)).

Starved(k)(sched)(s,m) iff Infsafe(k)(s,m,sched) ⊏ tt*

2) Deadlock(sched)(s,m) - The system is deadlocked if some process becomes starved.

Deadlock(sched)(s,m) iff (∃k)(Blocked(k)(sched)(s,m))

3) Safe(s,m) - We are often interested, regardless of the scheduler whether or not a particular set of processes can ever lead to deadlock. If not, the system is safe. Yet, we cannot ignore the scheduler completely, as degenerate schedulers can lead to anomalous behavior as we noted in an earlier section. We take as a minimal requirement that the scheduler be fair and defined.

Safe(s,m) iff
  (∀sched)(Defined(sched)(s,m) ∧ Fair(sched)(s,m) ⊃
            (∀k)(Infsafe(k)(sched)(s,m) ≡ tt*))

Clearly, the P/V system of the previous section is safe.

Of course, it is in general undecideable whether or not <s,m> is safe even in simple systems such as P/V (with conditionals), and even knowing that under a particular fair, full, defined scheduler, deadlock cannot occur.

Consider s composed of a semaphore, sem, initially 0, integers k and n, initially 0, and f, a description of a total function of type N --> TT. And let m be running the two processes informally described by:

```
   Process 1                          Process 2

k := 0;                            k := 1;
n := 0;                            loop
V(sem);                              if k = 0 then V(sem);
loop                                 P(sem);
  if Eval(f)(n) then V(sem);         endloop
  n := n + 1;
  endloop
```

Now, under a scheduler that runs process 2 first, the eventual value of k will be 0 and there will never be deadlock, but if process 1 runs first, k will be 1, and determining Safe(s,m) becomes equivalent to deciding whether f is true infinitely often, which is reducible to the halting problem.

MODELLING PROTECTION SYSTEMS

In the model presented, each process operates on a common memory state S. Yet in programming systems, different processes do have different accessing rules for accessing the memory (e.g. Frames, Contours, Virtual or Local Name Spaces and Execution Domains). By passing the EP its multiplexor slot as an argument, differential accessing of S can easily

be achieved. For example, if S = N --> DOMAIN, then if p is executing in Multiplexor slot k, s(k) could represent its execution domain.

Now, consider the modeling of a segmented operating system. Process j's data segments would be part of S, whereas its code segment would be modeled directly by the PROG component of M(j). We could then model the starting of process n by the EP:

Start: When TRUE do CONTENTS(<seg>) ==> LOADGO(<n>)

where contents(<seg>)(s) returns as its result the contents of segment seg (in state s), and loadgo(<n>) loads up those contents in M(n) and begins executing the process.

loadgo(<n>)(segcontents)(m) <==
  λk.(k=n --> <link(segcontents,n),BEGIN>, m(k))

where link(x,n) assembles x into PROG form with start address, BEGIN in process n.

An interesting byproduct is that one can model a process changing a data segment of another process (possible in systems with shared data) by using a STATEFORM, whereas a change in an executing process's code segment (most likely a bug) can only be modeled by using a CONTROLFORM (like LOADGO). In fact, in pursuing this modified model, just such a bug was discovered in CMU's HYDRA system.

(The bug in HYDRA can be circumvented by the use of "frozen" pages (see [Rotenberg 74]). A frozen code page is permanently protected against modification.)

Other small changes in the model make it more useful for describing and proving properties about protection systems. [Cohen 75] will report further results.

A CONJECTURED INDUCTION RULE

We will often want to prove (for some predicate Q)

A] Defined(sched0)(s0,m0), Fair(sched0)(s0,m0), Q(j,s0,m0) ⊢
Run(j)(s0,m0,sched0) ≡ tt*

under more difficult conditions than in the simple example of the applications section. We note that in the P/V example, process j becomes blocked when some other process, say k, has successfully executed a "P". Process k's subsequent execution of a "V" will then make process j runnable once more.

This is an instance of a more general observation. Suppose that

whenever process j is blocked, we are able to find a runnable process whose execution brings process j "closer" to becoming runnable and furthermore execution of any other process does not take process j farther away from becoming able to run. If we can show that after doing this a finite (though not necessarily bounded) number of times, process j actually becomes runnable, then under a fair scheduler we should be able to show that process j runs forever. Formally, we have the following induction principle:

Suppose that (W, ≤) is a well-founded set with a set of least elements W0 in which all intervals are of finite length. We write |w| for the maximum distance from w to an element of W0. Furthermore, let Assoc: W --> [S x M --> TT] and Closer: W --> N be total functions. Then to prove [A], it is sufficient to prove:

a) $Q(j,s,m) \vdash (\exists w)(\ Assoc(w)(s,m)\ )$

b) $w0 \in W0,\ Assoc(w0)(s,m) \vdash Canrun(j)(s,m)$

c) $w0 \in W0,\ Assoc(w0)(s,m) \vdash (\forall k)(\exists w)(\ Assoc(w)(Exec(k)(s,m))\ )$

d) $w \neg\in W0,\ Assoc(w)(s,m) \vdash$
$(\exists w')(\ |w'| < |w| \ \wedge\ Assoc(w')(Exec(Closer(w))(s,m))\ )$

e) $w \neg\in W0,\ Assoc(w)(s,m) \vdash\ (\forall k)(\exists w')($
$Assoc(w')(s,m)\ \wedge$
$(\ |w'| < |w| \ \vee\ (\ |w'| = |w| \wedge Closer(w') = Closer(w)\ )\ )\ )$

Intuitively, we use an abstraction of a token machine to determine whether or not process j can run forever. A token is always associated with some element w of W depending on s & m. As EP's are executed, s & m change, thus the token becomes associated with different elements of W. By proving properties about the movement of the token in W, we can prove that process j runs forever.

The basic idea is to associate the bottom elements of W, that is W0, with the states in which process j canrun. Then when the token is not associated with an element of W0, we must show that the token is eventually forced down towards an element of W0. We do this by demanding that when $w \neg\in W0$, there is some process Closer(w), such that the execution of that process will force the token to an element w' such that |w'| < |w|. Furthermore executing any other process must have the effect that either the token is forced to a w' lower than w anyway or the token moves to a w' at the same distance from the bottom (|w'| = |w|) but such that Closer(w') = Closer(w). Thus in the case that we have a fair scheduler, process k will eventually run and the token will eventually be pushed down closer toward W0. Since all intervals are of finite length, the token will eventually end up in W0. This will go on

forever, thus, process j will be runnable forever, and again, given a fair scheduler, process j will actually run forever.

Using this conjectured induction principle, we can easily prove the PVloop example. Define
W = {☆} + {w1,...,wn} and W0 = {☆}, under the ordering,
☆ < wi, i = 1,...,n.

Let Assoc(☆)(s,m) ≡ s = 1 ∧
m(k).LABEL = (Range(k) --> P, STOP)

and Assoc(wi)(s,m) ≡ s = 0 ∧
m(k).LABEL = (k=i --> V, Range(k) --> P, STOP)

and Closer(wi) = i.

Then, it is relatively trivial to prove that:

a) Assoc(☆)(s0,m0)

b) Assoc(☆)(s,m) ⊢ Canrun(j)(s,m)

c) Assoc(☆)(s,m) ⊢
(∀k)(Range(k) --> Assoc(wk)(Exec(k)(s,m)),
Assoc(☆)(Exec(k)(s,m)))

d) Assoc(wi)(s,m) ⊢
(∀k)(k=i --> Assoc(☆)(Exec(k)(s,m)),
Assoc(wi)(Exec(k)(s,m)))

which is easily seen to satisfy the induction predicates.

To simplify proofs, it may be useful to partition the system. We would have to define the notion of an "independent partition", and then prove that if <m1,...,mj> was an independent partition of m under s, then

Safe(s,m1), ..., Safe(s,mj) ⊢ Safe(s,m)

SCHEDULER NOTES

1) As noted in an earlier section, scheduler properties depend heavily on S and M as well as SCHED. Since future behavior of the system is completely determined by the initial system, all we need do is allow the scheduler to be tailor made to the initial configuration. Suppose that we demand that in the initial state of the system, n < j ⊃ m0(j).LABEL = STOP, and call this property

Init(m0,n).· The use of n, fixing an upper bound to the initial number of runnable processes allows us to define a recursive scheduler prototype:

PROTOSCHED = N × S × M --> SCHED

and a scheduler maker

Makesched: PROTOSCHED --> [N × S × M --> SCHED]

We say that PROTOSCHED is fair if

Init(m0,n) ⊃ Fair(Makesched(protosched)(n,s0,m0))(s0,m0)

and similarly for other properties.

2) Because the scheduler gets its information by looking at EP's, EP must be a domain over which a continuous "=" predicate is defined so that the scheduler can actually look at the components of the EP. Hence, the various FORM's of the EP are specified as NAMEs and list of ARGuments, rather than directly as functions.

CONCLUSION

We have introduced a semantic model for parallel systems and have presented a number of properties of parallel systems based on the model as well as some proofs and proof rules.

The development with the most potential appears to be the conjectured induction rule based on well founded sets As Cadiou & Levy note, LCF proofs force the program prover to (sometimes tediously) explicate all the possible states of the system. To make proofs of complex parallel programs more tractable, and especially to make the proofs more amenable to automatic verification, it seems clear that some (elegant) embedded or externally imposed (see [Milner & Weyrauch 72]) structure is critical. Well founded sets may be a useful structure for proofs of deadlock; for other properties of parallel programs, further exploration is necessary.

There is a different kind of structuring choice more directly related to this paper - what can be an indivisible operation embodied by an EP? If we assume an implementation on a sequential machine, the safest choice is the smallest action that cannot be interrupted. The obvious difficulty is that sequential machines are rare; even conventional machines often have an I/O processor and both may simultaneously be accessing memory. At best machines that use cycle-stealing force us to safely choose as indivisible actions those which take place in a single cycle.

We have assumed in this paper that actions as complex as synchronization operators may be viewed indivisibly and thus our proofs must therefore be viewed as correct only for models in which that is the case, thus we separate the model of indivisibility from its implementation. In the case of a multiprocessor, the code implementing synchronization may be running in parallel with other processes, perhaps even executing the same code. What must be shown in such a case is that the model of indivisibility is nonetheless valid regardless of such concurrency as may be introduced by the implementation. Such proofs are beyond the scope of this paper.

A somewhat serious deficiency of the scheduler model (and other models as well) is its inability to model time dependent behavior - for example timer interrupts in programming systems and timing considerations in machine architecture. While the nature of problems to be studied with respect to time dependencies would likely call for a different model in any case, proving the correctness of something like a multiplexor/scheduler for a multiprocessor would likely require a scheduler model modified in some way to handle time dependencies.

Perhaps the most serious problem with the model described here is in the nature of the assumptions made about how processes interact (or should interact). A formal semantics for a sequential programming language with structured control provides a better base for various proofs than a semantics for a language with GOTO's. Similarly, suitably restricted interactions between processes should provide a better semantic system than the one described here in which arbitrary interactions are allowed. A solution is to provide additional axioms which restrict the possible schedules. P/V disciplines are too unstructured. Work along the lines of Path expressions [Campbell & Haberman 74] appear to be more promising in providing a semantic basis in which proofs will be less tedious.

ACKNOWLEDGEMENTS

I wish to thank Bill Wulf, Nico Haberman and J. W. de Bakker for their comments on earlier drafts of this paper.

BIBLIOGRAPHY

[Cadiou & Levy 73] Cadiou, J. Levy, J. "Mechanizable Proofs about Parallel Processes" 14th Symposium on Switching Theory and Automata, Oct 73

[Campbell & Haberman 74] Campbell, R.H. Haberman, A. N. "The Specification of Process Synchronization by Path Expressions" Proc. Int. Symp. on Operating System Theory and Practice, Apr 74

[Cohen 75] Cohen, E. "A Semantic Model for Parallel Systems with Scheduling" Proc. 2nd ACM Symp. Princ. Prog. Langs., Jan 75

[Cohen 75] Cohen, E. "Modelling Protection Systems", CMU PhD Thesis, forthcoming

[deBakker 74] deBakker, J. W. "The Fixed Point Approach to Semantics: Theory and Applications" Mathematical Centre Tract 63, Mathematical Centre and Free University Amsterdam, 1974

[Dijkstra 72] Dijkstra, E. "A class of Allocation Strategies Inducing Bounded Delay Only" SJCC 72

[Kahn 73] Kahn, G. "A Preliminary Theory for Parallel Programs", I.R.I.A. Report, Jan 73

[Lipton 73] Lipton, R. "On Synchronization Primitive Systems", CMU PhD Thesis, June 73 or see Proceedings 6th Annual Symposium on the Theory of Computing, May 74

[Manna & Viullemin 72] Manna, Z. Viullemin, J. "Fixpoint Approach to the Theory of Computation" CACM v15,#7 July 72

[Milner 71] Milner, R. "An Algebraic Definition of Simulation Between Programs" I.J.C.A.I. 2, 1971

[Milner 72] Milner, R. "Implementation and Application of Scott's Logic for Computable Functions", Proceedings of a Conference on Proving Assertions about Programs, Jan 72

[Milner & Weyrauch 72] Milner R. Weyrauch R. "Proving Compiler Correctness in a Mechanized Logic" Machine Intelligence 7

[Milner 73] Milner, R. "An Approach to the Semantics of Parallel Programs" Proc. Convegno Informatica Teorica, Mar 73

[Newey 73] Newey, M. "Axioms and Theorems for Integers, Lists and Finite Sets in LCF", Stanford AIM-184, Jan 73

[Rotenberg 74] Rotenberg L. "Making Computers Keep Secrets" MIT PhD Thesis, MAC TR 115, Feb 74

[Scott 72] Scott, D. "The Lattice of Flow Diagrams", Symposium on Semantics of Algorithmic Languages, Springer Verlag Lecture Notes in Mathematics 188, 1971

[Scott 72] Scott, D. "Mathematical Concepts in Programming Language Semantics" SJCC 72

[Scott & Strachey 72] Scott, D. Strachey, C. "Toward a Mathematical Semantics for Computer Languages", Oxford Univ. Computing Lab PRG-6, 1972

[Wegner 72] Wegner, P. "The Vienna Definition Language", ACM Computing Surveys v4,#1 Mar 72

APPENDIX

The proofs here are presented as a series of Lemmas. Except for some difficult cases, an outline of the proof of each Lemma is all that is given. Only two inductive proof rules are used here, Computational Induction [Milner & Viullemin 72, Manna 72] and Mathematical Induction [Manna 72].

We use the abbreviations introduced by Milner [Milner72].
a :: b ≡ c for (a --> b,uu) ≡ (a --> c,uu).
and ∂(x) is the definedness predicate,
∂(uu) ≡ uu, otherwise, ∂(x) ≡ tt. We also use
↑(a) <== a --> tt,uu.

We also assume an extended LCF theorem prover with a knowledge of arithmetic (see axioms by Newey [Newey 73]) built in and therefore, when we are clearly dealing with a natural number, we dispense with the additional predicate isnat, e.g.
we write a :: b(n) ≡ c(n) instead of
a ∧ isnat(n) :: b(n) ≡ c(n).

We have not formally shown that Computational Induction is legitimate as we use it over the domains introduced in this paper. A proof in the style of Scott [Scott 72] is left to the reader.

We use "#" [Kahn 73] as a general concatenation operator, and leave proofs about its obvious properties to the reader.

THEOREM 1

```
Full(Fullsched(sched))(s,m)

Fullsched(sched) <== λ(s,m).Kfs(sched,0)(s,m).

Kfs(sched,n)(s,m) <==
  Let <s',m',sched'> be Desc(s,m,sched)(Kfn(s,m,sched)'(n)) in
    < sched'(s',m').N, Fullsched(sched'(s',m').SCHED) >

Kfn(s,m,sched)(n) <==
  Cr(Desc(s,m,sched)(n)) --> n,
  Kfn(s,m,sched)(n+1).

Desc(s,m,sched)(n) <==
  n = 0 --> <s,m,sched>,
  Next(Desc(s,m,sched)(n-1)).

Cr(s,m,sched) <== Canrun(sched(s,m).N)(s,m).
```

```
Aex(j,s,m) <== Exec(Action(j)(m))(s,m).

  LEMMA 1
Next(s,m,sched) ≡
  ¬Cr(s,m,sched) --> < s, m, sched(s,m).SCHED >,
  < Aex(sched(s,m).N,s,m), sched(s,m).SCHED >.
        Proof: by definitions

  LEMMA 2
¬Cr(Desc(s,m,sched)(n)) ::
  Desc(s,m,sched)(n).SM ≡ Desc(s,m,sched)(n+1).SM
        Proof: Defined of Desc & Lemma 1

  LEMMA 3
Canrun(Kfs(sched,n)(s,m).N)(Desc(s,m,sched)(n).SM) ⊑ tt
        Proof: Substitute Defn of Kfs, then use Computational
        Induction on Kfn, using Lemma 2 & Defn of Cr

  LEMMA 3a
Canrun(Fullsched(sched)(s,m).N)(s,m) ⊑ tt

  LEMMA 3b
Cr(s,m,Fullsched(sched)) ⊑ tt

  LEMMA 4
Desc(s,m,sched)(Kfn(s,m,sched)(n)).SM ⊑ Desc(s,m,sched)(n).SM
        Proof: Comp Ind on Kfn using Lemma 2

  LEMMA 4a
Desc(s,m,sched)(Kfn(s,m,sched)(0)).SM ⊑ <s,m>

  LEMMA 5
Cr(Desc(s,m,sched)(Kfn(s,m,sched)(n))) ≡
  Canrun(Kfs(sched,n)(s,m).N)(Desc(s,m,sched)(n).SM)
        Proof: Defn of Kfs & Cr and Lemma 4

  LEMMA 5a
Cr(Desc(s,m,sched)(Kfn(s,m,sched)(n))) ⊑ tt
        Proof: Lemmas 3 & 5

  LEMMA 5b
Cr(Desc(s,m,sched)(Kfn(s,m,sched)(0))) ≡ Cr(Fullsched(sched),s,m)

  LEMMA 6
Next(s,m,Fullsched(sched)) ≡
  Let <s',m',sched'> be Desc(s,m,sched)(Kfn(s,m,sched)(0)+1) in
    < s', m', Fullsched(sched') >
        Proof:
```

```
Next(s,m,Fullsched(sched))

  ≡    Cr(s,m,Fullsched(sched)) -->
     < Aex(Fullsched(sched)(s,m).N,s,m), Fullsched(sched)(s,m).SCHED >,
     uu.          Lemma 1 & 3b

  ≡    Cr(s,m,Fullsched(sched)) -->
     Let <s,m',sched'> be Desc(s,m,sched)(Kfn(s,m,sched)(0)) in
       < Aex(sched'(s',m').N,s,m), Fullsched(sched'(s',m').SCHED) >,
       uu.        Defn of Fullsched, Kfs

  ≡    Let <s',m',sched'> be Desc(s,m,sched)(Kfn(s,m,sched)(0)) in
     Cr(s',m',sched') --> < Aex(sched'(s',m').N,s',m'),
       Fullsched(sched'(s',m').SCHED) >,
     uu.        Lemmas 4a & 5b

  ≡    Let <s',m',sched'> be Desc(s,m,sched)(Kfn(s,m,sched)(0)) in
     < Next(s',m',sched').SM, Fullsched(Next(s',m',sched').SCHED) >.
                Lemmas 1 & 5a

  ≡    Let <s',m',sched'> be Desc(s,m,sched)(Kfn(s,m,sched)(0)+1) in
     < s', m', Fullsched(sched') >          Defined of Desc      QED .
```

Proof of THEOREM 1

```
Full(Fullsched(sched))(s,m)    by Defn of Full, we must prove

Runnable(j,k)(s,m,Fullsched(sched)) ⊑ tt*
        Proof: Computational Ind on Runnable

( j ≠ Fullsched(sched)(s,m).N or Canrun(j)(s,m) or ¬Canrun(k)(s,m) )
# Runnable(j,k)(Next(s,m,Fullsched(sched)))

  ⊑ tt # Runnable(j,k)(Next(s,m,Fullsched(sched)))      Lemma 3a

  ≡ tt # Let <s',m',sched'> be Desc(s,m,sched)(Kfn(s,m,sched)(0)+1) in
            Runnable(j,k)(s',m',Fullsched(sched'))      Lemma 6

  ⊑ tt # tt*    Induction

  ≡ tt*
```

THEOREM 2

```
Run(j)(s,m,sched) ≡ Run(j)(s,m,Fullsched))

Rbl(j)(s,m,sched) <==  ↑( (j = sched(s,m).N) ∧ Canrun(j)(s,m) )
```

```
Col(j)(s,m,sched)(n) <==
  n = 0 --> <>,
  Col(j)(s,m,sched)(n-1) # Rbl(j)(Desc(s,m,sched)(n-1)).

Crun(j)(s,m,sched) <== Let n be Kfn(s,m,sched)(0) + 1 in
  Col(j)(s,m,sched)(n) # Crun(j)(Desc(s,m,sched)(n)).
```

LEMMA 7

$Desc(Desc(s,m,sched)(a))(b) \equiv Desc(s,m,sched)(a+b)$

Proof: Math Ind on b

LEMMA 8

$\partial(Desc(s,m,sched)(n+k)) \sqsubseteq \partial(Desc(s,m,sched)(n))$

Proof: Lemma 7 & Axioms for ∂

LEMMA 9

$k \leq n \wedge Cr(Desc(s,m,sched)(n)) \supset Kfn(s,m,sched)(n-k) \leq n$

Proof: Math Ind on k using Lemma 8

LEMMA 9a

$Cr(Desc(s,m,sched)(n)) \supset Kfn(s,m,sched)(0) \leq n$

LEMMA 10

$Rbl(j)(s,m,sched) \sqsubseteq \uparrow(Cr(s,m,sched))$

Proof: Defn of Rbl & Cr

LEMMA 11

$n \leq Kfn(s,m,sched)(0) :: Col(j)(s,m,sched)(n) \equiv <>$

Proof: Math Ind on n using Lemma 9a & 10

LEMMA 11a

$Col(j)(s,m,sched)(Kfn(s,m,sched)(0)) \equiv <>$

LEMMA 12

$Rbl(j)(s,m,Fullsched(sched)) \equiv$
$Rbl(j)(Desc(s,m,sched)(Kfn(s,m,sched)(0))$

Proof: Lemma 5b & Defn of Fullsched

LEMMA 13

$Rbl(s,m,Fullsched(sched)) \equiv$
$Col(j)(s,m,sched)(Kfn(s,m,sched)(0)+1)$

Proof: Lemmas 11a & 12 by Defn of Col

THEOREM 2a

$Run(j)(s,m,Fullsched(sched)) \equiv Crun(j)(s,m,sched)$

Proof: Parallel Comp Ind on Run & Crun

LEMMA 14

Run(j)(s,m,sched) ≡
  Col(j)(s,m,sched)(s,m,sched)(n) # Run(j)(Desc(s,m,sched)(n))
        Proof: Math Ind on n

LEMMA 15

∂(Run(j)(Desc(s,m,sched)(n)) ⊑ ∂(Kfn(s,m,sched)(n))
        Proof: Parallel Comp Ind on Run & Kfn using Lemmas 7 & 10

LEMMA 16

Run(j)(s,m,sched) ≡ Let n be Kfn(s,m,sched)(0) + 1 in
  Col(j)(s,m,sched)(n) # Run(j)(Desc(s,m,sched)(n))
        Proof: By cases of ∂(Kfn(s,m,sched)(0)) using Lemmas 14 & 15

THEOREM 2b

Run(j)(s,m,sched) ≡ Crun(j)(s,m,sched)
        Proof: Parallel Comp Ind on Run & Crun using Lemma 16

Proof of THEOREM 2

Run(j)(s,m,sched) ≡ Run(j)(s,m,Fullsched(sched))
        Proof: Theorem 2a & 2b

THEOREM 3

Infcan(j)(s,m,Fullsched(sched)) ⊑ Infcan(j)(s,m,sched)
        Proof: Similar to proof of Theorem 2
        without use of Lemmas 11 & 11a and
        weaker versions of Lemma 13 and Theorem 2a

TIMINGS

The Scheduler formalism used in this paper is related closely to the Timings that appear in Lipton's work. The following section clarifies the relationship.

```
SEP = EP + {STOP}
TIMING = {<>}  +  ( N x SEP ) x TIMING
```

Thus, a Timing is a list of EP's (or {STOP}), with each EP associated the index of the process that executed it.

Since a timing is a list, there are three functions predeclared with the usual interpretation:

```
Car: TIMING --> N x SEP
Cdr: TIMING --> TIMING
Empty: TIMING --> TT
```

```
History: S x M x SCHED --> [ N --> TIMING ]

History(s,m,sched)(k) <==
  k = 0  -->  {<>},
  Let n be sched(s,m).N in
    Let sep be
      m(n).LABEL = STOP --> STOP, Action(n)(m)
    in
      < <n,sep>, History(Next(s,m,sched))(k-1) >.
```

```
Apply: SEP --> [ S --> S ]

Apply(sep)(s) <==
  sep = STOP --> s,
  Let < <syname,syargs>, <stname,stargs>, <cname,cargs> > be sep in
    Synchfn(syname)(syargs)(s) --> Statefn(stname)(stargs)(s).S, s.
```

```
Value: TIMING --> [ S --> S ]

Value(timing)(s) <==
  Empty(timing) --> s, Value(Cdr(timing))(Apply(Car(timing).SEP)(s)).
```

We use the predicate "Valid" for what Lipton calls "Semi-Active"

```
Valid(timing)(s,m)  iff
  (∃sched,k)( History(s,m,sched)(k) = timing )
```

```
Active: TIMING --> [ S --> TT ]

Active(timing)(s) <==
  Empty(timing) --> tt,
  Let < <n,sep>, rtiming > be timing in
    sep = STOP --> ff,
    Let <syname,syargs> be sep.SYNCHFORM in
    . ¬Synchfn(syname)(syargs) --> ff,
      Active(rtiming)(Apply(sep)(s)).
```

Timings form a partial order described in the following way:

```
≤: TIMING × TIMING --> TT

t1 ≤ t2  <==
  Empty(t1) --> tt,
  Empty(t2) --> ff,
  Let  < <n1,e1>, rt1 > be t1  and  < <n2,e2>, rt2 > be t2 in
    n1 = n2 ∧ e1 = e2  -->  rt1 ≤ rt2,  ff.
```

Conjecture:

```
Valid(timing)(s,m) ⊢    Active(timing)(s)  iff
    (∃k,sched)( Full(sched)(s,m) ∧ History(s,m,sched)(k) = timing )
```