Toward a Theory of Maximally Concurrent Programs *

[Shortened Version]

Rajeev Joshi
Compaq Systems Research Center
Palo Alto, CA
rjoshi@pa.dec.com

Jayadev Misra
The University of Texas at Austin
Austin, TX
misra@cs.utexas.edu

ABSTRACT

Typically, program design involves constructing a program P that *implements* a given specification S; that is, the set $\overline{P}$ of executions of P is a subset of the set $\overline{S}$ of executions satisfying S. In many cases, we seek a program P that not only implements S, but for which $\overline{P} = \overline{S}$. Then, every execution satisfying the specification is a possible execution of the program; we then call P *maximal* for the specification S. We argue that maximality is an important criterion in the context of designing concurrent programs because it disallows implementations that do not exhibit enough concurrency. In addition, a maximal solution can serve as a basis for deriving a variety of implementations, each appropriate for execution on a specific computing platform.

This paper also describes a method for proving the maximality of a program with respect to a given specification. Even though we prove facts about possible executions of programs, there is no need to appeal to branching time logics; we employ a fragment of linear temporal logic for our proofs. The method results in concise proofs of maximality for several non-trivial examples. The method may also serve as a guide in constructing maximal programs.

Categories and Subject Descriptors

D3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent Programming Structures*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms

Parallel Programs, Concurrency

Keywords

Concurrent Program Design, Maximal Solution

1. INTRODUCTION

Traditionally, a program specification is given by *safety* and *progress* properties. A safety property – of the form that no two neighbors eat simultaneously in a dining philosophers solution – is used to exclude certain undesirable execution sequences. A specification with safety properties alone can be implemented by a program that does nothing; then, the safety constraints have been implemented by excluding all non-trivial executions. Therefore, it is necessary to specify progress properties – of the form that some hungry philosopher eats eventually – requiring that some execution sequences be included. Safety and progress requirements are sufficient for specifying non-trivial sequential programming tasks, but they are not sufficient for concurrent program design because, for instance, in the case of the dining philosophers, the solution may allow only one philosopher to eat at a time, thus eliminating all concurrency. We propose a new requirement, called *maximality*, to ensure that only the most concurrent executions are included. Thus, the sequential solution to the dining philosophers problem will be unacceptable as a solution since it does not meet the maximality requirement.

Program design, typically, involves constructing a program P that *implements* a given specification S; that is, the set $\overline{P}$ of executions of P is a subset of the set $\overline{S}$ of executions satisfying S. For instance, given a specification to generate an infinite sequence of natural numbers, any program that generates a sequence of zeroes implements the specification. So does the program that generates the natural numbers in order. In many cases, we seek a program P that not only implements S – i.e., $\overline{P} \subseteq \overline{S}$ – but for which $\overline{P} = \overline{S}$. Then every execution satisfying the specification S is a possible execution of P; we call P *maximal* for specification S. For instance, the program that generates a stream of zeroes is *not* maximal for the specification to generate an infinite sequence of natural numbers; nor is the solution that allows a single philosopher to eat at a time maximal for the dining philosophers problem.

There are at least three reasons why we are interested in maximal solutions. First, as we have remarked above, we exploit maximality to eliminate those undesirable solutions for a given specification that restrict concurrency. (Since a maximal solution admits maximal concurrency, it suffers from no such restriction.)

Second, we often simulate an artifact by a program and the

latter has to simulate all behaviors of the former; in this case, the simulation program has to be maximal for the specification of the artifact. Such constructions are common in certain verification methods, such as model-checking. For instance, consider the problem of proving the correctness of a protocol for communication over a faulty channel (e.g., the Alternating Bit Protocol). A typical approach to showing the correctness of this protocol is to show that the programs describing the sender and receiver satisfy certain properties when composed with a program describing the faulty channel. For such a proof, we clearly require that the program used for the faulty channel be maximal for its specification.

The third reason for designing a maximal solution is that we often develop (and prove correct) such a solution, and then refine it – by eliminating some non-determinism, for instance – to obtain a program that is actually implemented. This strategy may be easier than developing the implemented program directly. A single maximal program for a problem may be the basis for a family of interrelated programs, each of which may be appropriate for a different computing platform. We show several refinements of a maximal solution for task scheduling in section 5.5.

A maximal solution is, typically, non-deterministic; in many cases the non-determinism is unbounded.

1.1 Overview of the paper

In this paper, we suggest a method for proving the maximality of a program with respect to a given specification. Given a program P to be proven maximal, we have to show that any sequence of states, σ, meeting the specification is a possible output of the program. We first construct a *constrained* program, P', from P and σ; the constrained program retains the structure of P, but its actions are restricted by guards and augmented by assignments to certain auxiliary variables. Next, we show that all fair executions of P' produce σ and that any such execution corresponds to a fair execution of P; hence, σ is a possible output of P.

Even though we prove facts about possible executions of programs there is no need to appeal to branching time logics; we employ a fragment of linear temporal logic for our proofs. The method seems to be quite effective in practice, resulting in concise proofs for non-trivial examples such as the task scheduler of section 5. The proposed method may also serve as a guide in constructing maximal programs from specifications.

2. PROGRAMS AND THEIR SPECIFICATIONS

2.1 Programming Model

We adopt a programming model based on UNITY [2]. A program has a set of variables that define its state, an *initial condition* which is a predicate over program states, and a nonempty set of *actions*, where each action is a relation over program states.

A program *execution* is an infinite sequence of the form $\tau_0 A_0 \tau_1 ... \tau_i A_i \tau_{i+1} ..$ where each τ_i is a program state and A_i is an action; τ_0 satisfies the initial condition and, for all i, $(\tau_i, \tau_{i+1}) \in A_i$. In addition, each execution satisfies the following **fairness requirement**: each action appears infinitely often.

We employ the following notation to describe the programming examples in this paper. The initial condition is defined in an *initially* section where the initial values of some of the variables are declared; the uninitialized variables have arbitrary initial values. The program actions are written as guarded commands, preceded optionally by a label, as in

$\alpha :: \ g \rightarrow s \quad .$

Execution of α has no effect in a state if g does not hold in that state; otherwise s is executed. We assume that execution of s terminates from any state where g is defined.

As an example, consider the following program having two integer variables x, n.

Program *FairNatural*
  var x, n: integer
  initially $n = 0$
  α:: $n := n + 1$
  β:: $x, n := n, 0$
end {*FairNatural*}

We claim that x is assigned only natural numbers as values, and x is infinitely often positive. Additionally, we can show that any sequence of states satisfying these two properties is the result of some execution of this program. Thus, this program is maximal for the specification that requires generation of an infinite sequence of natural numbers where, eventually, a positive number is generated. [1] □

2.2 Specifications

A specification is a set of program *properties*. We use the following operators of UNITY to specify the properties; see [2, 7, 8] for details. In the following, p, q are predicates over the program states and s is quantified over the actions of the program.

Safety properties are expressed using **co** and its derivatives. Property p **co** q holds for a program if in every execution a state in which p holds is followed immediately by a state in which q holds. A program has the property **stable** p if p continues to hold once it becomes true, and **invariant** p holds in a program if p is always true. For example, program *FairNatural* has the property **stable** $x \geq 0$, i.e., once x is nonnegative, it stays nonnegative.

Progress properties are described using the relation $\mapsto$ (*leads-to*): $p \ \mapsto \ q$ means that any state in which p holds is eventually followed by a state in which q holds. This operator is defined inductively; see [2] for details. For example, the program *FairNatural* of section 2.1 has the progress property *true* $\mapsto \ x > 0$, i.e., if *true* holds now, eventually x becomes positive.

[1] Consequently, this program can assign arbitrary natural numbers to x, i.e., it has unbounded non-determinism.

3. MAXIMALITY

Given a program P and a specification S it is possible to show that P *satisfies* S (i.e., P meets all the properties in S) using the UNITY logic [8, 7] (as outlined in section 2.2). To prove maximality, we show that any sequence that satisfies S may be obtained from an execution of P, in the sense described below. First, we define what it means for an infinite sequence σ of states to satisfy S. A sequence σ satisfies S if it satisfies each property in S, as described below. We consider only the following types of properties in S: **initially** p, p **co** q and $p \ \mapsto \ q$. In the following description, σ_i denotes element i of σ (with σ_0 denoting the first element), and $p(\sigma_i)$ means that p holds in the state σ_i.

σ satisfies **initially** p means $p(\sigma_0)$ holds.
σ satisfies p **co** q means $(\forall\, i :: p(\sigma_i) \Rightarrow q(\sigma_{i+1}))$.
σ satisfies $p \ \mapsto \ q$ means $(\forall\, i :: (\exists\, j : i \leq j : p(\sigma_i) \Rightarrow q(\sigma_j)))$.

Given an execution τ of a program, and a subset V of the set of program variables, the "projection of τ over V" is the (infinite) sequence of states obtained by removing the action labels from τ, and projecting the resulting sequence of states to the set V.

For any execution τ, subset V of program variables, and sequence σ of states over V, we say that "τ reduces to σ over V" provided the projection of τ over V is equivalent to σ upto finite stuttering [5].

For example, in program *FairNatural*, the fragment of the execution sequence $(1,0)\alpha(1,1)\alpha(1,2)\beta(2,0)\alpha(2,1)\beta(1,0)$ – where each state is a pair of values, of the form (x, n) – reduces to the fragment 121, and also to 1121, over the set of variables $\{x\}$.

Definition. Program P is *maximal* for specification S and variable set V provided P satisfies S and for any sequence σ satisfying S there is an execution τ of P such that τ reduces to σ over V.

To keep the notation simple, we will henceforth assume that V is understood in the specification, and we will write "P is maximal for specification S".

3.1 Constrained Program

We next describe a method to prove maximality of a program P for a specification S. Let σ be a sequence of states that satisfies S; we have to show that some execution of P reduces to σ. Our strategy is to construct a *constrained* program P' such that all executions of P' reduce to σ, and all executions of P' correspond to fair executions of P, in the sense defined later.

The constrained program P' is constructed from P as follows.

1. The variables of P are retained in P'; they are called the *original* variables.
2. New constants, called *chronicles*, are introduced in P'. Chronicles are like history variables: they encode the given state sequence σ. They are not altered in the constrained program; their values are only read. There may be several chronicles, one corresponding to each variable of the specification, to encode the sequence of values taken on by the variables during a computation.
3. Additional variables, called *auxiliary* variables, are introduced in P'. Auxiliary variables are used in the proof. In our examples, we use a special auxiliary variable, which we call a *point*, to show the position in the chronicle that matches the current state of P'.
4. An action α of P is modified to $\alpha' :: g \rightarrow \alpha \ ; \ \beta$ where g is a guard that may name any variable of P', and β, which is optional, may assign only to the auxiliary variables. Action α' is an *augmented* action corresponding to α and g is the *augmenting* guard of α'. Augmenting an action may eliminate some of the executions of P. Note: If α has a guard h then, effectively, α' has guard $g \ \wedge \ h$.
5. Constrained program P' may also include additional actions of the form $g \rightarrow \beta$ where g names any variable of P' and β assigns only to the auxiliary variables.
6. The initial condition of P' implies the initial condition of P.

Note that in P', chronicles remain unchanged during execution, since they appear only in guards and other tests. Auxiliary variables appear only in guards (and other tests) and in assignments to themselves. Original variables of P are assigned values exactly as they were assigned in P, except that some of the variables that were uninitialized in P may be initialized in P'.

3.2 Example

Consider the program *FairNatural* of section 2.1. To prove its maximality for the specification

stable $x \geq 0$, and $\mathit{true} \ \mapsto \ x > 0$

choose any sequence X that satisfies the specification, i.e,

$(\forall\, i :: X_i \geq 0 \Rightarrow X_{i+1} \geq 0)$, and
$(\forall\, i :: (\exists\, j : i \leq j : X_j > 0))$.

Now, construct a constrained version *FairNatural*$'$ of the program, by including the chronicle X and an auxiliary variable j denoting the point. The augmented actions corresponding to α and β are α' and β'. There are no superposed actions.

Program *FairNatural*$'$
  var x, n: integer; X: sequence of integer; j: integer
  initially $n = 0 \wedge x = X_0 \wedge j = 1$
  α':: $n < X_j \rightarrow n := n + 1$
  β':: $n = X_j \rightarrow x, n := n, 0; \ j := j + 1$
end {*FairNatural*$'$}

We claim that in every fair execution of *FairNatural′* the sequence of values assigned to x is X, i.e., **invariant** $j > 0 \wedge x = X_{j-1}$. We also show that every fair execution of *FairNatural′* corresponds to a fair execution of *FairNatural*. Hence, X is the outcome of a possible execution of *FairNatural*.

3.3 Proving Maximality

We describe the proof steps required to establish the maximality of a program for a given specification. The constrained program inherits all safety properties of the original program since the assignments to the original variables are not modified. We have to establish the following facts in the constrained program.

1. **Chronicle Correspondence:** Show that every fair execution of the constrained program assigns a sequence of values to the original variables that match the values in the respective chronicles.
   - (Safety) Show that the values of the original variables are identical to those of the chronicles at the current point (recall that the point is, typically, given by an auxiliary variable, such as j in *FairNatural*). This proof obligation is stated as an invariant of the constrained program.
   - (Progress) The current value of the point will be incremented eventually. (This often follows from the progress proof for execution correspondence.)
2. **Execution Correspondence:** Show that every fair execution of the constrained program corresponds to a fair execution of the original program such that both executions compute the same values in the original variables.
   - (Safety) The truth of the augmenting guard of each action is preserved by all other actions. That is, the augmenting guard of α' may be falsified by executing α' only.
     This condition is met trivially if all augmenting guards are pairwise disjoint; in this case, each guard is falsifiable only by the action it is associated with.
   - (Progress) Show that each augmenting guard is true infinitely often.

Example

For *FairNatural′* our proof obligations are as follows. The detailed proof is given in the full paper [4].

1. Chronicle Correspondence:
   (Safety) **invariant** $j > 0 \wedge x = X_{j-1}$.
   (Progress) $j = J \mapsto j = J + 1$, for any natural J.
2. Execution Correspondence:
   (Safety) $n < X_j$ is preserved by β', and $n = X_j$ is preserved by α'. (These follow because the guards are disjoint.)
   (Progress) $true \mapsto n < X_j$, $true \mapsto n = X_j$.

We omit justification of these proof rules due to space constraints; see the full paper [4] for details.

4. RANDOM ASSIGNMENT

A maximal solution is, typically, highly non-deterministic. In our previous example, *FairNatural*, we exploited the non-determinacy of action execution; an arbitrary natural number is computed because n is incremented an indeterminate number of times. In many cases, it is convenient to have non-determinacy in the code itself. To this end, we introduce *random assignment statements* of the form

$x :=?$ **st** p

where variable x is assigned any value such that predicate p holds after the assignment. It is the programmer's responsibility to ensure that this assignment is feasible. A refinement of this statement will assign a specific value to x that satisfies p. For instance, for integer x

$x :=?$ **st** $x > {}'x$,
where ${}'x$ denotes the value of x before the assignment

increases the value of x arbitrarily.

There is one caveat in constructing these proofs. Earlier, we had said that a constrained program inherits all safety properties of the original program. This is true only if the random assignments have been correctly constrained. Therefore, it cannot be assumed that the constrained program inherits the safety properties until the correctness of these assignments in the constrained program have been shown. In particular, the proof of correctness of these assignments can not assume any safety properties of the original program; any such assumption has to be proven explicitly in the constrained program.

5. A TASK SCHEDULER

In this section, we consider a scheduling problem in which concurrency is essential; the requirement of concurrency can be succinctly stated using maximality. The following scheduling problem is from [9]. We are given a finite number of *tasks* and a *compatibility* relation among the tasks. Two tasks may be concurrently executed provided they are compatible. It is given that an executing task will terminate eventually. The goal is to design a task scheduler that repeatedly selects tasks for execution so that: (1) only compatible tasks are executed concurrently, and (2) each task is executed infinitely often.

The following abstraction captures the essence of the scheduling problem. We are given a simple, finite undirected graph in which there are no self-loops; the graph need not be connected. Each node in the graph is *black* or *white*; all nodes are initially white. In this abstraction, a node denotes a task and a black node an executing task. Two nodes are neighbors if they are *incompatible*, i.e., not compatible. We are given that every black node becomes white eventually, i.e., each task terminates. It is required to devise a coloring (scheduling) strategy so that

- No two neighbors are simultaneously black (i.e., only compatible tasks may be executed simultaneously).
- Every node becomes black infinitely often.

Note that the scheduler can only blacken nodes; it may not whiten a node.

A simple scheduling strategy is to blacken a single node, wait until it is whitened, and then blacken another node. Such a strategy implements the first requirement trivially because there is at most one black node at any time. The second requirement may be met by blackening the nodes in some fixed, round-robin order. Such a protocol, however, defeats the goal of concurrent execution of tasks. So, we impose the additional requirement that the scheduling strategy be maximal: any valid blackening of the tasks may be obtained from a possible execution of our scheduler. By suitable refinement of our maximal scheduler we derive a centralized scheduler and a distributed scheduler, as described in section 5.5.

5.1 Specification

Let b denote the set of black nodes at any stage in the execution. For sets x, y and a node v, we write $x = y + v$ to denote that $v \notin y \ \wedge x = y \cup \{v\}$.

S0. **initially** $b = \emptyset$.
S1. $(\forall\, u, v : u \ neighbor \ v : \neg(u \in b \wedge v \in b))$.
S2. $b = B$ **co** $b = B \vee (\exists\, v :: b = B + v \ \vee \ B = b + v)$, for any B.
S3. For all v, $true \ \mapsto v \in b$ and $true \ \mapsto v \notin b$.

The specification S0 states that initially no tasks are executing; S1 states that neighbors are never simultaneously black; S2 says that in a step at most one node changes color. In S3, $true \ \mapsto v \notin b$ is established by the tasks themselves (each task terminates, and, hence, becomes white, eventually), and the scheduler has to implement the remaining progress property, $true \ \mapsto v \in b$.

5.2 A Scheduling Strategy

Assign a natural number, called *height*, to each node; let $H[u]$ denote the height of node u. The predicate $u.low$ holds if the height of u is smaller than all of its neighbors, i.e.,

$$u.low \equiv (\forall\, v : u \ neighbor \ v : H[u] < H[v]).$$

The scheduling strategy is to set b to $\emptyset$ initially, and the node heights in such a way that neighbors have different heights. Then, the following steps are repeated.

- (Blackening Rule) Eventually consider each node, v, for blackening; if $v \notin b \wedge v.low$ holds then blacken v.
- (Whitening Rule) Simultaneous with the whitening of a node v, increase $H[v]$ to a value that differs from $H[u]$, for all neighbors u of v.

Formally, the coloring strategy is described by the following program. There is an action $add(v)$, for each node v, that adds v to b provided $v \notin b \wedge v.low$. The termination of task v is simulated by $remove(v)$, that removes v from b and increases $H[v]$ to a value that differs from $H[u]$, for all neighbors u of v.

Program *Scheduler*
  var u, v: node; b: set of node
  var H: **array** item **of** natural
  initially $b = \emptyset \ \wedge \ (\forall\, u, v : u \ neighbor \ v : H[u] \neq H[v])$
  $\langle \forall v ::$
    $add(v) ::\ v \notin b \wedge v.low \rightarrow b := b \cup \{v\}$
    $remove(v) ::\ v \in b \ \rightarrow b := b - \{v\};$
        $H[v] :=?$ **st** $H[v] > {}'H[v]$
            $\wedge \ (\forall\, u : u \ neighbor \ v : H[u] \neq H[v])$
  $\rangle$
end $\{Scheduler\}$

Note: ${}'H[v]$ is the value of $H[v]$ before the assignment.

5.3 Correctness of the Scheduling Strategy

We show that neighbors have different heights at all times, i.e.,

P0. **invariant** $(\forall\, x, y : x \ neighbor \ y : H[x] \neq H[y])$.

Proposition P0 holds initially. If P0 holds prior to the execution of $add(v)$ then it holds following the execution, because $add(v)$ does not affect heights. If P0 holds prior to the execution of $remove(v)$ it holds afterwards, because only $H[v]$ changes and $H[v] \neq H[u]$, for any neighbor u of v, following $remove(v)$.

Proof of S0

Follows from the initialization.

Proof of S1

The coloring strategy described above maintains the following invariant: for all v, $v \in b \ \Rightarrow \ v.low$. Observe that this proposition holds initially since all nodes are initially white. A blackening step (add) preserves the proposition because $v.low$ is a precondition for blackening. A whitening step ($remove$) preserves the proposition because the antecedent of the proposition becomes false.

From this invariant, if u, v are both black then they are both low, and from the definition of low, it follows that u, v are not neighbors. Therefore, neighbors are not simultaneously black.

Proof of S2

In $add(v)$, the assignment $b := b \cup \{v\}$ has the precondition $v \notin b$. In $remove(v)$, the assignment $b := b - \{v\}$ has the precondition $v \in b$. Hence, S2 is satisfied.

Proof of S3

We show that every node becomes black infinitely often in every execution. Suppose that there is a node x that becomes black only a finite number of times in a given execution. Each blackening and the subsequent whitening increases the height of a node. Therefore, if some neighbor y of x becomes black infinitely often then its height will eventually exceed $H[x]$, establishing $\neg y.low$, and y will never be blackened subsequently. Hence, every neighbor of x is blackened finitely often. Applying this argument repeatedly, no node connected to x can become black infinitely

often. Therefore, beyond some stage, q, in an execution, all nodes in the component of the graph to which x belongs will remain white forever. Let v be a node with the smallest height in this component at q in the execution; since all nodes remain white beyond q their heights do not change and v remains a node with the smallest height. Whenever v is considered for blackening beyond q, it will meet all the conditions for blackening (v is white and $v.low$ holds); thus v will be blackened, contradicting the conclusion that v remains white forever beyond q.

The proof by contradiction, given above, is typical of the style in which many concurrent algorithms are proven in the literature. We present an alternative proof, based on the style of UNITY, that avoids arguments by contradiction; see appendix B.

5.4 Proof of Maximality

Let z be a sequence of sets, denoting a possible sequence of values of b in an execution; assume that z is stutter-free, i.e., successive values in z are distinct. Let z satisfy the specification (S0, S1, S2, S3), i.e., (S0′, S1′, S2′, S3′) hold.

S0′. $z_0 = \emptyset$.
S1′. For all i, $(\forall\, u, v : u \textit{ neighbor } v : \neg(u \in z_i \wedge v \in z_i))$.
S2′. For all i, $(\exists\, v :: z_{i+1} = z_i + v \vee z_i = z_{i+1} + v)$.
S3′. For all v, $\quad(\forall\, i :: (\exists\, j : i \leq j : v \in z_j))$,
and $(\forall\, i :: (\exists\, j : i \leq j : v \notin z_j))$.

We create the following constrained program that includes a variable t, denoting the current point of computation. The variable $u.next$ is an abbreviation for the next value, j, above t where u is in z_j. Formally,

$u.next = (\min j : j > t \wedge u \in z_j : j)$.

Note that $u.next$ is always defined, on account of S3′.

Program *Scheduler′*
var u, v: node; b: set of node; t: integer
initially $b = \emptyset \;\wedge\; t = 0 \;\wedge\; (\forall\, v :: H[v] = v.next)$
⟨∀v::
$add'(v)$:: $z_{t+1} = z_t + v \rightarrow$
$v \notin b \wedge v.low \rightarrow b := b \cup \{v\};\ t := t + 1$
$remove'(v)$:: $z_t = z_{t+1} + v \rightarrow$
$v \in b \;\rightarrow b := b - \{v\};\ H[v] := v.next;$
$t := t + 1$
⟩
end {*Scheduler′*}

5.4.1 Invariants of the Constrained Program

The following invariants hold for *Scheduler′*. The variable v is quantified over all nodes.

P1. $b = z_t$.
P2. $z_{vh} = z_{vh-1} + v$ where vh denotes $H[v]$
P3. $(\forall\, u, v : u \textit{ neighbor } v : H[u] \neq H[v])$.
P4. $v.next \geq H[v] \wedge v.next > t$.
P5. $(H[v] = v.next) \equiv v \notin b$.

Proof of P1

Initially, $b = \emptyset$, $t = 0$, and from (S0′) $z_0 = \emptyset$. Each action increments t and modifies b appropriately.

Proof of P2

This follows from the text of *Scheduler′* and S2′.

Proof of P3

This property is similar to invariant P0 proved for *Scheduler*. However, we can not assert that this property is inherited by *Scheduler′* until we show that the random assignment is correctly implemented. Therefore, we have to construct a new proof. Let uh, vh denote $H[u], H[v]$ respectively, and suppose that $uh = vh$. Then, from P2

$z_{vh} = z_{vh-1} + v \;\wedge\; z_{uh} = z_{uh-1} + u$
⇒ {By assumption, $uh = vh$}
$z_{uh} = z_{uh-1} + v \;\wedge\; z_{uh} = z_{uh-1} + u$
⇒ {Set theory}
$u = v$

Thus, for distinct nodes u, v, we have $H[u] \neq H[v]$. Hence, the same result applies for neighbors u, v.

Proof of P4

To see the first conjunct, note that initially, $(\forall\, v :: H[v] = v.next)$. The only assignment to $H[v]$ is $H[v] := v.next$ in $remove'(v)$; so $v.next \geq H[v]$ is preserved by this assignment. Also, $v.next$ is monotone in t; therefore, $v.next$ never decreases in *Scheduler′* because t never decreases.

The second conjunct follows from the definition of $v.next$.

Proof of P5

Initially P5 holds because b is $\emptyset$ and $(\forall\, v :: H[v] = v.next)$. First, we show that P5 is preserved by the execution of $add'(v)$.

Define $v.next.i = (\min j : j > i \wedge v \in z_j : j)$. Thus, $v.next = v.next.t$. Rewrite condition P5 as $(H[v] = v.next.t) \equiv v \notin b$. This holds as a postcondition of the assignments

$b := b \cup \{v\};\ t := t + 1$

provided $H[v] \neq v.next.(t+1)$ holds as a precondition. We show below that the precondition of $add'(v)$, $z_{t+1} = z_t + v \wedge v \notin b \wedge v.low$ and P5, implies $H[v] \neq v.next.(t+1)$.

$z_{t+1} = z_t + v \wedge v \notin b$
⇒ {From the definition of $v.next$,
$(z_{t+1} = z_t + v) \Rightarrow (v.next = t + 1)$}
$v.next = t + 1 \wedge v \notin b$
⇒ {P5: $(H[v] = v.next) \equiv v \notin b$}
$H[v] = t + 1$
⇒ {from definition, $v.next.(t+1) > t + 1$}
$H[v] \neq v.next.(t+1)$

It can be shown that $H[u]$ and $u.next$ are unaffected by the execution of $add'(v)$, for $v \neq u$. Also, from the text of $remove'(v)$ it is seen that $v \notin b \wedge (H[v] = v.next)$ is established.

5.4.2 Rewriting the guard of $add'(v)$

We show from the given invariants that the augmenting guard of $add'(v)$, $z_{t+1} = z_t + v$, implies the original guard, $v \notin b \wedge v.low$. Hence, the original guard may be dropped in the constrained program. This result is needed for the proof of progress in chronicle correspondence; see (2) of section 5.4.4.

From $b = z_t$ (see P1) and $z_{t+1} = z_t + v$, we have $v \notin b$. We show that $v.low$ holds, i.e., for neighboring nodes u, v, $H[v] < H[u]$.

$$
\begin{array}{ll}
 & z_{t+1} = z_t + v \\
\Rightarrow & \{b = z_t \text{ from P1}\} \\
 & v \notin b \wedge v \notin z_t \wedge v \in z_{t+1} \\
\Rightarrow & \{\text{Definition of } v.next\} \\
 & v \notin b \wedge v.next = t+1 \wedge v \notin z_t \wedge v \in z_{t+1} \\
\Rightarrow & \{\text{From P5, } (H[v] = v.next) \equiv v \notin b\} \\
 & H[v] = t+1 \wedge v \notin z_t \wedge v \in z_{t+1} \\
\Rightarrow & \{u, v \text{ are neighbors,} \\
 & \quad v \in z_{t+1} \Rightarrow u \notin z_{t+1}, \text{ from S1}'\} \\
 & H[v] = t+1 \wedge v \notin z_t \wedge v \in z_{t+1} \wedge u \notin z_{t+1} \\
\Rightarrow & \{v \notin z_t \wedge v \in z_{t+1} \wedge u \notin z_{t+1} \text{ from S2}', u \notin z_t\} \\
 & H[v] = t+1 \wedge v \notin z_t \ \wedge \ v \in z_{t+1} \\
 & \qquad\qquad \wedge u \notin z_t \ \wedge \ u \notin z_{t+1} \\
\Rightarrow & \{\text{from P1, } b = z_t, \text{ from P4, } u.next > t, \text{ and} \\
 & \quad \text{from P5, } (H[u] = u.next) \equiv u \notin b\} \\
 & H[v] = t+1 \wedge H[u] = u.next \wedge u.next > t \\
\Rightarrow & \{H[v] = t+1 \wedge H[u] > t. \text{ Apply }\} \\
 & H[v] < H[u]
\end{array}
$$

5.4.3 Correctness of the Implementation of Random Assignment

The random assignment

$$H[v] :=? \ \mathbf{st} \ H[v] > {}'H[v] \wedge (\forall\, u : u \ neighbor \ v : H[u] \neq H[v])$$

is implemented in the constrained program by

$$H[v] := v.next.$$

The precondition of the assignment, $z_t = z_{t+1} + v$ and (from P1) $b = z_t$, imply that $v \in b$. Hence, from P4 and P5, $H[v] < v.next$ prior to the assignment; now $H[v] = v.next$ after the assignment, thus establishing $H[v] > {}'H[v]$. The condition $(\forall\, u : u \ neighbor \ v : H[u] \neq H[v])$ follows from P3.

5.4.4 Proof of Chronicle Correspondence

1. (Safety) $b = z_t$ follows from P1.

2. (Progress) $t = N \mapsto t = N + 1$, for any natural N: exactly one guard of $Scheduler'$ holds at any stage in the computation because the guards are disjoint and their disjunction is *true*. Execution of any action whose guard is true increments t.

5.4.5 Proof of Execution Correspondence

1. (Safety) Guards of all the actions are disjoint.

2. (Progress) We have to show
   $true \mapsto z_{t+1} = z_t + v$, and
   $true \mapsto z_t = z_{t+1} + v$.

   We sketch a proof. From S3′ we can deduce that
   $(\forall\, i :: (\exists\, j : i \leq j : z_{j+1} = z_j + v))$, and
   $(\forall\, i :: (\exists\, j : i \leq j : z_j = z_{j+1} + v))$.

   From (2) of section 5.4.4, t assumes values of successive natural numbers. Therefore, eventually, $z_{t+1} = z_t + v$ and also eventually, $z_t = z_{t+1} + v$.

5.5 Refining a Maximal Solution: Implementation of the Scheduling Strategy

We consider the situation where each task (node) is executed on a separate processor. First, we show how a central scheduler may schedule the tasks given the compatibility relation. Next, we show how the scheduling may be distributed over the processors.

5.5.1 Central scheduler

A central scheduler maintains a list of nodes and their current colors and heights. Periodically, it scans through the nodes and blackens a node v provided $v.low \wedge v \notin b$ holds. Whenever it blackens a node it sends a message to the appropriate processor specifying that the selected task may be executed. Upon termination of the task, the processor sends a message to the scheduler; the scheduler whitens the corresponding node and increases its height, ensuring that no two neighbors have the same height. The scheduler may scan the nodes in any order, but every node must be considered eventually.

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

5.5.2 Distributed scheduler

The proposed scheduling strategy can be distributed so that each node blackens itself eventually if it is white and low. The nodes communicate by messages of a special form, called *tokens*. Associated with each edge (x, y) is a token. Each token has a *value*, a positive integer equal to $|H[x] - H[y]|$. This token is held by either x or y, whichever has the smaller height.

It follows then that a node that holds all incident tokens has a height that is smaller than all of its neighbors; if such a node is white, it may color itself black. A node, upon becoming white, increases its height by a positive amount d, effectively reducing the value of each incident token by d (note that such a node 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 node's height is increased. If the value of token (x, y) 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 nodes need not query each other for their heights, because a token is eventually sent to a node 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 node 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 nodes. There is no need to keep the node heights explicitly from then on.

We have left open the question of how a node'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 node's height beyond all its neighbors' heights whenever it is whitened; this amounts to sending all incident tokens to the neighbors when a node is whitened. Under this strategy, the token values are immaterial: a white node is blackened if it holds all incident tokens and upon being whitened, a node 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 by Chandy and Misra [1]; a black node is *eating* and a white node is *hungry*; constraint (S1) is the well-known requirement that neighboring philosophers do not eat simultaneously. Our current problem has no counterpart of the thinking state, which added a slight complication to the solution in [1]. The tokens are called *forks* in that solution.

6. SUMMARY

We have described the notion of maximality, which rules out implementations with insufficient non-determinism. A maximal program for a given specification has (upto stuttering) all the behaviors admitted by the specification. We showed several examples of maximal solutions, including a fair unordered buffer and a fair task scheduler. Notions similar to maximality have been studied elsewhere in the literature, e.g., the various flavors of *bisimulation* due to Milner and others [6]. However, unlike bisimulation, which relates two programs (i.e., agents of a process algebra), our notion of maximality relates a program written using guarded-commands with a specification written in a UNITY-like temporal logic. Although we have concerned ourselves here only with showing maximality, our proof method may be used with any given set of executions, to show that a given program admits all those executions.

7. ACKNOWLEDGMENTS

This paper has been enriched by comments and suggestions from the PSP research Group at the University of Texas at Austin, and the Distributed Systems Reading Group at the Technische Universität München, Germany.

8. REFERENCES

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

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

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

[4] R. Joshi and J. Misra. Maximally concurrent programs. *Formal Aspects of Computing*, (to appear).

[5] L. Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668, Paris, Sep 1983. IFIP, North-Holland.

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

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

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

[9] J. Misra. A discipline of multiprogramming, work in progress, ftp access at `ftp://ftp.cs.utexas.edu/pub/psp/seuss/discipline.ps.Z`, 1996.

APPENDIX

A. SUMMARY OF UNITY LOGIC

The UNITY logic, a fragment of linear temporal logic, has proof rules for reasoning about properties of programs. A short summary is given here; consult [7, 8] for details.

A.1 Safety

The fundamental safety operator of UNITY is *constrains*, or **co** for short. The property p **co** q asserts that in any execution a state satisfying p is always followed by a state satisfying q. In order to model stuttering steps p is required to imply q. The **co** operator and its derivative operators are defined as follows, where s is quantified over the actions of the program, and wp denotes weakest precondition [3]

$$p \textbf{ co } q \;\equiv\; (\forall\, s :: p \Rightarrow wp.s.q)$$
$$\textbf{stable } p \;\equiv\; p \textbf{ co } p$$
$$\textbf{invariant } p \;\equiv\; \textbf{initially } p \text{ and } \textbf{stable } p$$

A predicate is *stable* if it remains true once it becomes true. A predicate is *invariant* if it is stable and it holds in all initial program states. Observe that $p \wedge \neg q$ **co** $p \vee q$ is a property of a program if from any state where p holds it continues to hold until q holds; if q never holds then p holds for ever.

A.1.0.1 The Substitution Axiom

The operation of a program is over the reachable part of its state space. The UNITY proof rules, however, do not refer to the set of reachable states explicitly. Instead, the following *substitution axiom* is used to restrict attention to the reachable states: if **invariant** p is a property of a program then p may be replaced by *true* in any context.

A.2 Progress

The elementary progress operator, **en**, used in this paper has the following informal meaning. If p holds at any stage in the computation it will continue to hold as long as q does not hold, and q holds eventually. Further, there is one (atomic) action which guarantees to establish q starting in any p-state. Formally,

$$p \textbf{ en } q \;\triangleq\; (p \wedge \neg q \textbf{ co } p \vee q) \wedge (\exists\, s :: (p \wedge \neg q) \Rightarrow wp.s.q)$$

where s is quantified over all the actions of the program.

Given p **en** q, from the second conjunct in its definition, there is an action of the program that establishes q starting in any state in which $p \wedge \neg q$ holds; from the first conjunct, once p holds it continues to hold at least until q is established. Therefore, starting in a state in which p holds q will eventually be established.

Most of the progress properties of UNITY are expressed using the $\mapsto$ (leads-to) operator, a binary relation on state predicates. It is the transitive, disjunctive closure of the *ensures* relation, i.e., the strongest relation satisfying the following three conditions:

(basis)
$$\frac{p \textbf{ en } q}{p \;\mapsto\; q}$$

(transitivity)
$$\frac{p \;\mapsto\; q,\; q \;\mapsto\; r}{p \;\mapsto\; r}$$

(disjunction) In the following, S is any set of predicates.
$$\frac{(\forall\; p \;:\; p \in S \;:\; p \;\mapsto\; q)}{(\exists\; p \;:\; p \in S \;:\; p) \;\mapsto\; q}$$

Derived Rules for leads-to There are several derived rules for reasoning about the progress properties. Here, we mention only the ones used in this paper.

- implication

$$\frac{p \;\Rightarrow\; q}{p \;\mapsto\; q}$$

- lhs-strengthening, rhs-weakening

$$\frac{p \;\mapsto\; q}{\begin{array}{l} p' \;\wedge\; p \;\mapsto\; q\;, \\ p \;\mapsto\; q \;\vee\; q' \end{array}}$$

- cancellation

$$\frac{p \;\mapsto\; q \;\vee\; r\;,\; r \;\mapsto\; s}{p \;\mapsto\; q \;\vee\; s}$$

- PSP

$$\frac{p \;\mapsto\; q\;,\; r \textbf{ co } b}{p \;\wedge\; r \;\mapsto\; (q \wedge b) \;\vee\; (\neg r \wedge b)}$$

- Induction: In the following M is a total function mapping program states to a well-founded set $(W, \prec)$.

$$\frac{\langle \forall\; m \;:\; m \in W \;::\; p \;\wedge\; M = m \;\mapsto\; (p \wedge M \prec m) \;\vee\; q \rangle}{p \;\mapsto\; q}$$

In this paper we have used induction over natural numbers only.

B. PROOF OF PROGRESS, (S3), FOR THE TASK SCHEDULER

In section 5.3, we argued that (S3) holds. A proof using UNITY logic follows. It is required to prove that every node becomes black eventually, i.e., for all x, $\mathit{true} \mapsto x \in b$. Define

the *relative height* $x.rh$ of node x to be the sum of the height differences of x and all its neighbors of lower heights, i.e.,

$$x.rh = (+y : x \ neighbor \ y \ \wedge \ H[x] > H[y] : H[h] - H[y])$$

The following properties can be proven directly from the program text; each $\mapsto$ property is indeed an *ensures* property. For all x, y, n,

1. $x.low \mapsto x \in b$.
2. $x.rh = n \ \wedge \ (x \ neighbor \ y) \ \wedge \ y.low \mapsto (x.rh = n) \ \wedge \ (x \ neighbor \ y) \ \wedge \ y \in b$.
3. $x.rh = n \ \wedge \ (x \ neighbor \ y) \ \wedge \ y \in b \mapsto x.rh < n$.

We give an informal argument for the validity of these three properties. A node's height does not change as long as it remains white. Therefore, if x is low and white then it remains low (because its neighbors' heights can only increase) and white, until blackened. Eventually, x is considered for blackening and then blackened, establishing property (1). Proof of (2) is similar: the node y of the lowest height among the neighbors of x will eventually be black and until then $x.rh$ is unchanged. Property (3) says that that node y, as described above, will eventually become white and then $x.rh$ is decreased because the height of y is increased. The proof of $true \mapsto x \in b$ follows.

$x.rh = n \ \wedge \ (x \ neighbor \ y) \ \wedge \ y.low$
$\mapsto (x.rh = n) \wedge \ (x \ neighbor \ y) \ \wedge \ y \in b$
, From (2)

$x.rh = n \ \wedge \ (x \ neighbor \ y) \ \wedge \ y.low \mapsto x.rh < n$
, transitivity with (3)

$x.rh = n \ \wedge \ (\exists \, y :: (x \ neighbor \ y) \ \wedge \ y.low) \mapsto x.rh < n$
, disjunction over all y

$x.rh = n \ \wedge \ \neg x.low \mapsto x.rh < n$
, using Invariant P0 and the definition of *low*

$x.rh = n \ \wedge \ x.low \mapsto x \in b$, strengthening LHS of (1)

$x.rh = n \ \mapsto x.rh < n \ \vee \ x \in b$
, disjunction on above two

$true \mapsto x \in b$, induction on the above