Under consideration for publication in Formal Aspects of Computing

Maximally Concurrent Programs [1]

Rajeev Joshi[a] and Jayadev Misra[b]

[a]Compaq Systems Research Center, Palo Alto, CA
[b]The University of Texas, Austin, TX

Keywords: Concurrent Program Design, Safety, Progress, Maximal Solution

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.

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

[1] Partially supported by the NSF grant CCR–9803842

– 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 inter-related 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.

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 fair unordered channel of section 4.2 and 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 [CM88]. 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 $n = 0$
   $\alpha$:: $n := n + 1$
   $\beta$:: $x, n := n, 0$
end {FairNatural}
```

We claim that x is assigned only natural numbers as values, and x is infinitely often positive. We prove these claims in the section 2.2. We show in section 3.4 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. [2] □

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

Stuttering Each program contains *skip* as an action; this action will not be shown explicitly in the program. The effect of executing *skip* is to leave the program state unchanged; thus, the state is repeated in an execution. Note that, because of the fairness requirement on actions, there are at most finitely many consecutive occurrences of *skip* in any execution. (However, a program state may repeat forever; this happens when execution reaches a state in which no program action changes the state.) This use of stuttering in the context of refinement is due to Lamport [Lam83].

Interaction with an Environment An environment that interacts with the program *FairNatural* will, typically, "call" β to receive the next value of x. The programming model in this paper does not support procedure calls. A more general model, such as Seuss [Mis96], would allow β to be called as a procedure. Then, the output of the program is the sequence of values of x returned to the caller. In our current model, however, we can encode the interaction with the environment as follows: introduce a counter c that records the number of executions of (i.e., calls upon) β; that is, the action β increments c. A possible output sequence of this program is a sequence of states $(c_0, x_0), ..., (c_i, x_i), ..$, where $c_i = i + 1$. The goal of maximality is to show that any such sequence is a possible output of this program.

2.2. Specifications

A specification is a set of program *properties*. We use the following operators of UNITY to specify the properties; see appendix A for a short summary and [Mis95a, Mis95b] for details. In the following, p, q are predicates over the program states and s is quantified over the actions of the program.

2.2.1. Safety

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. (See section 3 for a formal definition.) 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. See the appendix for details.

Note: If p **co** q holds for a program [3], then $p \Rightarrow q$. To see this, recall that every program contains *skip* as an action. Since executing *skip* from a p-state results in a p-state, if the program satisfies p **co** q, it follows that $p \Rightarrow q$.

2.2.2. Progress

The elementary progress operator, **en**, or *ensures*, has the following informal meaning. If p holds at any point in the computation it will continue to hold as long as q does not hold, and eventually q holds. Further, there is one (atomic)

[3] To be precise, this implication holds only over the "reachable" states of the program. But that is a technicality that we will not be concerned with in this paper.

action which is guaranteed to establish q starting in any p-state; see the appendix for a formal definition.

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, as shown in the appendix.

Proofs in this paper appeal to a number of derived rules that are given in [Mis95a, Mis95b]; see [APP93, Pau99] for mechanical proofs of some of these rules.

Example: Program *FairNatural* of section 2.1 has the following properties.

stable $x \geq 0$, i.e., once x is nonnegative, it stays nonnegative
true $\mapsto$ $x > 0$, i.e., if *true* holds now, eventually x becomes positive

Note that since *true* holds for every state, the leads-to property above is equivalent to saying that x is positive infinitely often.

To prove that **stable** $x \geq 0$ is a property, we have to show that both actions α and β preserve $x \geq 0$; this follows from the program text. The progress property may be proven as follows. First, show that **invariant** $n \geq 0$. Then,

true **en** $n > 0$	, program text, and **invariant** $n \geq 0$	
true $\mapsto$ $n > 0$	, basis rule	(1)
$n > 0$ **en** $x > 0$	, program text	
$n > 0$ $\mapsto$ $x > 0$	, basis rule	
true $\mapsto$ $x > 0$	, transitivity on (1) and the above	

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 [Mis95b, Mis95a] (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 [Lam83].

For example, in the 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'.

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

stable $x \geq 0$,
true $\mapsto$ $x > 0$

choose an arbitrary 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 $n = 0 \wedge x = X_0 \wedge j = 1$
    $\alpha'$:: $n < X_j \rightarrow n := n + 1$
    $\beta'$:: $n = X_j \rightarrow x, n := n, 0;\ j := j + 1$
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*.

Remark on the Constrained Program The following example shows that a constrained program may not be executable.

Consider the following specification: output a sequence of integers where each element is one more than the preceding element with, possibly, one exception where the element is one *less* than the preceding element. Given below is a program that is maximal for this specification. The program has an integer variable x and a boolean b. Neither variable is initialized. If b is true then x only increases, and if b is false then eventually x decreases and then increases forever. The output is the sequence of values assigned to x.

```
Program choice
    var $x$: integer; $b$: boolean
    $\alpha$:: $x := x + 1$
    $\beta$:: $\neg b \rightarrow x := x - 1;\ b := true$
end {choice}
```

To prove maximality given a possible output sequence X, we have to construct a constrained program in which the initial value of b depends on X. However, no finite prefix of X can tell us how to initialize b: b has to be set true if and only if X is an increasing sequence.

initially $b = (\forall\, i : i \geq 0 : X_{i+1} = X_i + 1)$

Therefore, the constrained program is not executable.

3.2. 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 section 3.4 below.

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) $true \ \mapsto n < X_j$, $true \ \mapsto n = X_j$.

3.3. Justification for the Proof Rules

The chronicle correspondence rule establishes that the computation of the constrained program P' matches the given chronicle. The safety requirement guarantees the match at the current point and the progress requirement guarantees that successively longer prefixes of the chronicle will be computed.

Given that the execution correspondence conditions hold, we argue that for any fair execution τ of P', with $\tau = \tau_0 A_0 \tau_1 ... \tau_i A_i \tau_{i+1}..$, there is a fair execution γ of P, with $\gamma = \gamma_0 B_0 \gamma_1 ... \gamma_i B_i \gamma_{i+1}..$, such that τ reduces to the sequence of states $\gamma_0 \gamma_1 ... \gamma_i \gamma_{i+1}..$ over the variables of P.

We modify τ by removing certain actions and states from it, as follows. For each action A_i in τ that has an augmenting guard g, if $g(\tau_i)$ does not hold then ($\tau_i = \tau_{i+1}$ in this case) remove $\tau_i A_i$ from τ. We show that the resulting sequence, τ', is an infinite sequence, and hence, an execution.

From the progress condition of execution correspondence, the augmenting guard, g, of an augmented action α' is true infinitely often; from the safety condition of execution correspondence, g remains true as long as α' is not executed.

Each action α' is executed infinitely often in a fair execution of P'. Therefore, α' is infinitely often executed in a state where its augmenting guard, g, is true. Actions whose guards were false at the time of their execution were removed from τ. Therefore τ' contains every augmented action infinitely often, and the corresponding guard is then true. In a state where the augmenting guard g of α' holds, α' has the same effect on the original variables as the action α that it corresponds to. (The superposed actions do not modify the original variables.) Therefore, τ' is an execution of the constrained program and it corresponds to a fair execution, γ, of the original program such that the sequence of states for the original variables in τ, τ' and γ are identical (upto finite stuttering).

Not all computations of the constrained program, P', have counterparts in P, the original program. In particular, if X is a sequence of zeroes then *FairNatural'* computes X by executing the following sequence of actions, $(\alpha'\beta')^\omega$; in this execution, α' has no effect and β' computes the next value. However, the corresponding sequence, $(\alpha\beta)^\omega$ in *FairNatural*, does not compute X. The execution correspondence rule ensures that every fair execution of P' corresponds to a fair execution of P that computes the same sequence of states (in the original variables of P). In *FairNatural'* the guard of α', $n < X_i$, does not hold infinitely often if X is a sequence of zeroes, and, hence, the execution correspondence rule does not apply.

3.4. Proof Sketch of Maximality of *FairNatural*

We state certain properties of *FairNatural* that are required in the maximality proof; these properties follow from the program text.

P1. **invariant** $j > 0 \;\wedge\; n \le X_j \;\wedge\; x = X_{j-1}$.
P2. $j = J$ **co** $j = J \vee (j = J + 1 \wedge n = 0)$ for all natural J
P3. $X_j - n = K + 1$ **en** $X_j - n = K$ for all natural K
P4. $n = X_j \wedge j = J$ **en** $j = J + 1 \wedge n = 0$ for all natural J

We also have the following properties of X from the specification of *FairNatural*.

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

Property P5 below follows from the properties of X.

P5. There is a function, f, $f : naturals \rightarrow naturals$, such that
$f(i) > i$ and $X_{f(i)} > 0$, for all i.

Here, $f(i)$ denotes the next position beyond i where $X_{f(i)}$ is positive. Such a position exists because $(\forall\, i :: (\exists\, j : i \le j : X_j > 0))$.

Next, we show the proofs of chronicle correspondence and execution correspondence.

3.4.1. Proof of chronicle correspondence

1. (Safety) **invariant** $j > 0 \;\wedge\; x = X_{j-1}$ follows from P1.
2. (Progress) $j = J \mapsto j = J + 1$, for any natural J:

$true \mapsto\; n = X_j$, see (2) of section 3.4.2
$j = J$ **co** $j = J \vee (j = J + 1 \wedge n = 0)$

, P2
$j = J \;\mapsto\; (n = X_j \wedge j = J) \vee (j = J+1 \wedge n = 0)$
, PSP applied to the above two
$n = X_j \wedge j = J \;\mapsto j = J+1 \wedge n = 0$
, basis rule of $\mapsto$ on P4
$j = J \;\mapsto\; j = J+1 \wedge n = 0$, cancellation on the above two (*)
$j = J \;\mapsto\; j = J+1$, weakening the rhs

3.4.2. Proof of execution correspondence

1. (Safety) The guards, $n < X_j$ and $n = X_j$, are disjoint.
2. (Progress) *true* $\mapsto n = X_j$:

   $n \le X_j \wedge X_j - n = K \;\mapsto\; (n \le X_j \wedge X_j - n < K) \vee n = X_j$
   , basis rule of $\mapsto$ on P3
   $n \le X_j \mapsto\; X_j = n$, induction
   true $\mapsto n = X_j$, substitution axiom with **invariant** $n \le X_j$

3. (Progress) *true* $\mapsto n < X_j$:

   $j = J \;\mapsto\; j = f(J) - 1$, induction on (2) of section 3.4.1
   $j = f(J) - 1 \;\mapsto\; j = f(J) \wedge n = 0$
   , let J be $f(J) - 1$ in (*) of section 3.4.1
   $j = J \;\mapsto\; j = f(J) \wedge n = 0$, transitivity on the above two
   $j = J \;\mapsto\; n < X_j$, $j = f(J) \Rightarrow X_j > 0$
   true $\mapsto n < X_j$, disjunction over all J

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 that, essentially, assigns a random value to a variable; see section 9.4 of [AO97] for an axiomatic treatment of random assignment. We show the additional proof steps required to prove the constrained program when random assignments are replaced by specific assignments. As an example, we treat a fair unordered channel in which random assignments are essential in constructing the solution.

4.1. The Form of Random Assignment

A random assignment statement is of the form
$x :=?$
and execution of this statement assigns a random value of the appropriate type to x. There is no notion of fairness in this assignment; repeated execution of this statement may always assign the same value to x.

Random assignment is convenient for programming maximal solutions. However, it can be simulated using the existing features of our programming model.

For instance, the following program can be used to assign a random natural number to x. The program is similar to *FairNatural*; every execution of γ stores a random natural number in x. (However, there is no requirement that nonzero values be produced infinitely often.) The program is also maximal: any sequence of natural numbers may be assigned to x.

```
Program RandomNatural
    var $x, n$: integer
    initially $n = 0$
    $\alpha$:: $n := n + 1$
    $\beta$:: $n > 0 \rightarrow n := n - 1$
    $\gamma$:: $x := n$
end {RandomNatural}
```

4.1.1. Note on the Maximality of RandomNatural

The proof of maximality of *RandomNatural* is similar to that of *FairNatural*; so, we omit the proof. Note, however, that augmenting α, β, γ by the guards $n < X_j,\ n > X_j,\ n = X_j$, where X is a given sequence of natural numbers as in *FairNatural*, is not sufficient for the proof of maximality. If X is an increasing sequence, for instance, then $n > X_j$ will never hold, and execution correspondence cannot be proven. Create a constrained program in which the codes of the augmented actions α' and β' are executed at least once following each execution of γ'. This can be implemented by having another auxiliary variable c, $c \in \{0, 1, 2\}$, with the following meaning: $c = 1$ if the last executed action is γ', and then α' is executed and c is set to 2; if $c = 2$ then β' is executed and c is set to 0; when $c = 0$ any of α', β', γ' may be executed. The constrained program is shown below.

```
Program RandomNatural'
    var $x, n$: integer;
          $X$: sequence of integer;
          $j$: integer; $c$: {0,1,2}
    initially $n = 0 \wedge x = X_0 \wedge c = 0 \wedge j = 1$
    $\alpha'$:: $(c = 0 \wedge n < X_j) \vee c = 1 \rightarrow n := n + 1$; if $c = 1$ then $c := 2$
    $\beta'$:: $(c = 0 \wedge n > X_j) \vee c = 2 \rightarrow$
              $n > 0 \rightarrow n := n - 1;\ c := 0$
    $\gamma'$:: $c = 0 \wedge n = X_j \rightarrow x := n;\ c := 1;\ j := j + 1$
end {RandomNatural'}
```

Note: The augmenting guard of β' implies $n > 0$, since $n > X_j \Rightarrow n > 0$ and it can be shown that **invariant** $(c = 2 \Rightarrow n > 0)$.

4.1.2. General Form of Random Assignment

We use a more general form of random assignment
    $x :=?$ **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** $(\exists\, i :: x = 2 \times i)$

assigns any even number to x, and

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

4.1.3. Constraining Random Assignments

In constructing a constrained program a random assignment is replaced by a specific assignment. If

x :=? **st** p, is replaced by

$x := e$

it has to be shown that p holds after the assignment $x := e$.

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.

4.2. Fair Unordered Channel

In order to illustrate proofs with random assignments we take the example of a channel interposed between a sender and a receiver. A first-in-first-out (fifo) channel guarantees that the order of delivery of messages is the same as the order in which they were put into the channel. In this section we consider a fair unordered channel in which (1) the messages are delivered in random order, and (2) every message sent is eventually delivered. A fifo channel implements both requirements, but it is not maximal.

This problem, couched as a message transmission problem, has a number of other applications. In particular, the solution can be used to output all natural numbers in some order, and any order is possible. The solution can be used as a fair scheduler for programs that have an infinite number of actions, and it admits any fair schedule.

We consider the following simplification of the problem. A program has an infinite input sequence x and it has to generate a sequence y that is a permutation of x; any permutation is a possible output. We assume further that the items in x are distinct, which can be assured by appending a unique sequence number to each item of x. Then, every item in y corresponds to a unique item in x, and vice versa. The specification of the program is as follows: the safety conditions state that every item in y is from x and that the elements in y are unique; the progress condition states that every item of x appears in y eventually.

$(\forall\, j :: (\exists\, i :: x_i = y_j))$,
$(\forall\, i, j :: y_i = y_j \;\Rightarrow\; i = j)$, and
$(\forall\, i :: \mathit{true} \mapsto (\exists\, j :: x_i = y_j))$.

4.2.1. Maximal Solution for Fair Unordered Channel

Our solution consists of two actions, *read* and *write*. In the *read* action an item is removed from x and stored in a set z; in the *write* action an item from z is removed and appended to sequence y. It is not sufficient to remove a random item of z in *write*; then, the progress property may not hold. Therefore, we associate a *height*, a natural number, with each item that is placed into z and in the *write* action remove any item with the smallest height from z. An item is assigned any height greater than or equal to the value of variable t when it is added to z; we describe below how t is computed.

In the following program, heights of items are stored in array H. Variables i, j denote the number of items read from x and written to y, respectively.

```
**Program** $FUnCh$
  **var** $i, j, t$: integer; $c$: item; $x, y$: sequence of item;
        $z$: set of item;
        $H$: **array** item **of** natural
  **initially** $i = 0 \wedge j = 0 \wedge t = 0 \wedge y = \langle\rangle \wedge z = \emptyset$
  $read$:: $c := x_i$; $H[c] :=?$ **st** $H[c] \geq t$;
          $z := z \cup \{c\}$; $i := i + 1$
  $write$:: $z \neq \emptyset \rightarrow$
            $c :=?$ **st** $c \in z \wedge (\forall\, d : d \in z : H[c] \leq H[d])$;
            $t,\ y_j,\ z,\ j := H[c] + 1,\ c,\ z - \{c\},\ j + 1$
**end** $\{FUnCh\}$
```

The following properties hold for *FUnCh*.

$(\forall\, j :: (\exists\, i :: x_i = y_j))$, and
$(\forall\, i :: \mathit{true} \mapsto (\exists\, j :: x_i = y_j))$.

We leave it to the reader to prove these properties. For the progress property, it has to be shown that each item u in z is selected eventually, as c, in *write*. Let p be the number of items in z whose height is less than t. Show that in the pair $(H[u] + 1 - t, p)$ both components are non-negative, the pair is unaffected by the execution of *read*, and it decreases lexicographically whenever an item is removed from z. Therefore, eventually, u is removed.

4.2.2. The Constrained Program

Let Y be any sequence that is a permutation of x, i.e.,

$(\forall\, j :: (\exists\, i :: x_i = Y_j))$,
$(\forall\, i, j :: Y_i = Y_j \ \Rightarrow\ i = j)$, and
$(\forall\, i :: (\exists\, j :: x_i = Y_j))$.

We show that Y is a possible output of the program. A constrained program is shown below in which, in addition to the transformations described in section 3.1, the random assignments have been replaced by specific assignments.

```
**Program** $FUnCh'$
  **var** $i, j, t$: integer; $c$: item; $x, y, Y$: sequence of item;
        $z$: set of item;
        $H$: **array** item **of** natural
  **initially** $i = 0 \wedge j = 0 \wedge t = 0 \wedge y = \langle\rangle \wedge z = \emptyset$
  $read'$:: $Y_j \notin z \rightarrow$
            $c := x_i$; $H[c] := k$ **st** $c = Y_k$;
            $z := z \cup \{c\}$; $i := i + 1$
  $write'$:: $Y_j \in z \rightarrow$
```

```
        $z \neq \emptyset \rightarrow$
          $c := Y_j;$
          $t,\ y_j,\ z,\ j := H[c]+1,\ c,\ z-\{c\},\ j+1$
**end** $\{FUnCh'\}$
```

Notes: The assignment to $H[c]$ in *read'* is not a random assignment; there is a unique value Y_k that matches x_i. The augmenting guard of *write'*, $Y_j \in z$, implies the original guard, $z \neq \emptyset$.

4.2.3. Proof of Maximality: Invariants

We write $x_{0:i}$ to stand for the set $\{x_0, x_1, ..., x_{i-1}\}$; thus, $x_{0:0}$ is the empty set. The proofs of the following invariants are left to the reader.

P1. **invariant** $x_{0:i} = z \cup y_{0:j}$.
P2. **invariant** $y_{0:j} = Y_{0:j}$.
P3. **invariant** $(\forall\, d : d \in z : d = Y_{H[d]} \wedge j \leq H[d])$.
P4. $t = j$.

The proofs of P1, P2 are straightforward; these proofs use the fact that the items in z are distinct. Proof of P3 needs some explanation. The action *read'* adds c to z where $H[c] = k \wedge c = Y_k$; hence, $c = Y_{H[c]}$. To see that $j \leq H[c]$ in *read'*: it follows from P1 that $x_i \notin y_{0:j}$, hence, $c = x_i = Y_k$ where $j \leq k$, i.e., $j \leq H[c]$. The action *write'* removes c from z provided $H[c]$ is the smallest height. From P3, all heights are distinct because all items in Y are distinct; furthermore, each height is at least j. From the guard, $Y_j \in z$, the height of Y_j is the lowest and all other items in z have height exceeding j. Therefore, the incrementation of j in *write'* preserves $j \leq H[d]$ for each d in z. The proof of P4 is similar.

4.2.4. Correctness of Implementation of Random Assignments

We have to show

1. in *read'*: $H[c] := k$ **st** $c = Y_k$ implements $H[c] :=?$ **st** $H[c] \geq t$.
2. in *write'*: $c := Y_j$ implements $c :=?$ **st** $c \in z \wedge (\forall\, d : d \in z : H[c] \leq H[d])$.

Proof of (1) In *read'*, prior to the assignment we have, from the invariant P1,

$$
\begin{array}{ll}
 & x_{0:i} = z \cup y_{0:j} \\
\Rightarrow & \{\text{From P2, } y_{0:j} = Y_{0:j};\ x \text{ is a permutation of } Y\} \\
 & x_i \notin Y_{0:j} \wedge (\exists\, k :: x_i = Y_k) \\
\Rightarrow & \{\text{Predicate calculus}\} \\
 & (\exists\, k : k \geq j : x_i = Y_k) \\
\Rightarrow & \{k \text{ above is unique since items of } Y \text{ are distinct; } c = x_i\} \\
 & H[c] := k \ \mathbf{st}\ c = Y_k \text{ implements } H[c] :=?\ \mathbf{st}\ H[c] \geq j \\
\Rightarrow & \{\text{From P4, } j = t\} \\
 & H[c] := k \ \mathbf{st}\ c = Y_k \text{ implements } H[c] :=?\ \mathbf{st}\ H[c] \geq t
\end{array}
$$

Proof of (2) We have to show after the assignment $c := Y_j$ that $c \in z \wedge (\forall\, d : d \in z : H[c] \leq H[d])$. Applying the axiom of assignment, we have to show before the assignment that $Y_j \in z \wedge (\forall\, d : d \in z : H[Y_j] \leq H[d])$ holds. The first term in the consequent, $Y_j \in z$, follows from the guard of *write'*. For the remaining part,

$$\begin{array}{ll} & (\forall\, d : d \in z : H[Y_j] \leq H[d]) \\ \Leftarrow & \{H[Y_j] = j \text{ from P3}\} \\ & (\forall\, d : d \in z : j \leq H[d]) \\ \Leftarrow & \{\text{from P3}\} \\ & \mathit{true} \end{array}$$

4.2.5. Proof of Chronicle Correspondence

- (Safety) We have to show that $y_{0:j} = Y_{0:j}$, which follows from P2.
- (Progress) We have to show that $j = J \mapsto j = J + 1$, for any natural J. Each execution of write' increments j. From the progress proof of write' under execution correspondence the code of write' is executed infinitely often. Therefore, j increases without bound.

4.2.6. Proof of Execution Correspondence

- (Safety) The augmenting guards, $Y_j \notin z$ and $Y_j \in z$, are disjoint.
- (Progress of read') $\mathit{true} \mapsto Y_j \notin z$:

$$\begin{array}{ll} Y_j \in z \;\textbf{en}\; Y_j \notin z & \text{, from program text} \\ Y_j \in z \mapsto\; Y_j \notin z & \text{, basis rule of } \mapsto \\ Y_j \notin z \mapsto\; Y_j \notin z & \text{, implication rule of } \mapsto \\ \mathit{true} \mapsto Y_j \notin z & \text{, disjunction of the above two} \end{array}$$

- (Progress of write') $\mathit{true} \mapsto Y_j \in z$: There is a unique k such that $Y_j = x_k$. For any n,

$$\begin{array}{ll} Y_j \notin z \wedge k - j = n \;\textbf{en}\; k - j < n & \text{, from program text} \\ Y_j \notin z \wedge k - j = n \mapsto\; k - j < n & \text{, basis rule of } \mapsto \\ Y_j \notin z \mapsto\; Y_j \in z & \text{, induction} \\ \mathit{true} \mapsto Y_j \in z & \text{, similar to the proof for } \mathit{read}' \end{array}$$

4.3. Faulty Channel

We consider a faulty channel that may lose messages, duplicate any message an unbounded (though finite) number of times, and permute the order of messages. For any point in the computation, it is given that not all messages beyond this point will be lost; otherwise, there can be no guarantee of any message transmission at all. This is similar to the fault model of a channel assumed in the Alternating Bit Protocol [SBW69] (the difference being that in the latter, the channel does not reorder messages). Such a protocol can be studied (proved correct) by encoding the communication between the sender and the receiver using a maximal solution for the faulty channel. As we have remarked earlier, it is sometimes essential to have a maximal solution in this case, e.g., for use in verifying a communication protocol using model-checking. In this section, we sketch a maximal solution for faulty channel, but we leave the actual program, correctness and maximality proof to the reader. The maximality proof is similar to that for the *FUnCh*.

We simulate a faulty channel using a bag b, analogous to the set z in *FUnCh*.

The bag holds the messages that are to yet be delivered; it may hold several copies of the same message to simulate duplication, and the nature of a bag implements out-of-order delivery. To simulate message loss and duplication, we compute a count n whenever a message is added to b; the count is an arbitrary natural number, denoting the number of times that the message is to be delivered. If $n = 0$ for a message then it is immediately discarded (the message is lost), and for n exceeding 0 the message is added n times to b. In order to implement the requirement that not all messages are eventually lost, we require that n become non-zero periodically. Clearly, *FairNatural* can be used to compute n.

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 [Mis96]. 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. $(\forall\, u, v : u\ \mathit{neighbor}\ v : \neg(u \in b \wedge v \in b))$.
S2. $b = B$ **co** $b = B \vee (\exists\, v :: b = B + v\ \vee\ B = b + v)$, for any B.
S3. For all v, $\mathit{true}\ \mapsto v \in b$ and $\mathit{true}\ \mapsto v \notin b$.

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, $\mathit{true}\ \mapsto v \notin b$ is established by the tasks themselves (each task terminates, and, hence, becomes white, eventually), and the scheduler has to implement the remaining progress property, $\mathit{true}\ \mapsto v \in b$.

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

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 $b = \emptyset \ \wedge\ (\forall\, u, v : u\ \mathit{neighbor}\ v : H[u] \neq H[v])$
   $\langle \forall v ::$
      $add(v) :: v \notin b \wedge v.low \rightarrow b := b \cup \{v\}$
      $remove(v) :: v \in b\ \rightarrow b := b - \{v\};$
            $H[v] :=?$ st $H[v] > {}'H[v]\ \wedge\ (\forall\, u : u\ \mathit{neighbor}\ v : H[u] \neq H[v])$
   $\rangle$
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** $(\forall\, x, y : x\ \mathit{neighbor}\ y : H[x] \neq H[y])$.

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. Next, we present an alternative proof, based on the style of UNITY, that avoids arguments by contradiction.

Formal proof of S3 It is required to prove that every node becomes black eventually, i.e., for all x, $true \mapsto x \in b$. 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.,

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

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.

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

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

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** $b = \emptyset \ \wedge \ t = 0 \ \wedge \ (\forall \, v :: H[v] = v.next)$
  $\langle \forall v ::$
    $add'(v)$:: $z_{t+1} = z_t + v \rightarrow$
              $v \notin b \wedge v.low \rightarrow b := b \cup \{v\}$; $t := t + 1$
    $remove'(v)$:: $z_t = z_{t+1} + v \rightarrow$
              $v \in b \ \rightarrow b := b - \{v\}$; $H[v] := v.next$; $t := t + 1$
  $\rangle$
```

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. $(\forall\, u, v : u\ neighbor\ v : H[u] \neq H[v])$.
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

$$\begin{array}{ll} & z_{vh} = z_{vh-1} + v \ \wedge\ z_{uh} = z_{uh-1} + u \\ \Rightarrow & \{\text{By assumption, } uh = vh\} \\ & z_{uh} = z_{uh-1} + v \ \wedge\ z_{uh} = z_{uh-1} + u \\ \Rightarrow & \{\text{Set theory}\} \\ & u = v \end{array}$$

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 $H[v] \neq v.next.(t + 1)$.

$$\begin{array}{ll} & z_{t+1} = z_t + v \wedge v \notin b \\ \Rightarrow & \{\text{From the definition of } v.next,\ (z_{t+1} = z_t + v) \Rightarrow (v.next = t + 1)\} \\ & v.next = t + 1 \wedge v \notin b \\ \Rightarrow & \{\text{P5: } (H[v] = v.next) \equiv v \notin b\} \end{array}$$

$H[v] = t + 1$
$\Rightarrow$ {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, $H[v] < H[u]$.

$z_{t+1} = z_t + v$
$\Rightarrow$ {$b = z_t$ from P1}
$v \notin b \wedge v \notin z_t \wedge v \in z_{t+1}$
$\Rightarrow$ {Definition of $v.next$}
$v \notin b \wedge v.next = t + 1 \wedge v \notin z_t \wedge v \in z_{t+1}$
$\Rightarrow$ {From P5, $(H[v] = v.next) \equiv v \notin b$}
$H[v] = t + 1 \wedge v \notin z_t \wedge v \in z_{t+1}$
$\Rightarrow$ {Given u, v are neighbors, $v \in z_{t+1} \Rightarrow u \notin z_{t+1}$, from S1′}
$H[v] = t + 1 \wedge v \notin z_t \wedge v \in z_{t+1} \wedge u \notin z_{t+1}$
$\Rightarrow$ {Given $v \notin z_t \wedge v \in z_{t+1} \wedge u \notin z_{t+1}$ from S2′, $u \notin z_t$}
$H[v] = t + 1 \wedge v \notin z_t \wedge v \in z_{t+1} \wedge u \notin z_t \wedge u \notin z_{t+1}$
$\Rightarrow$ {using $b = z_t$ (P1), $(H[u] = u.next) \equiv u \notin b$ (P5), and $u.next > t$ (P4)}
$H[v] = t + 1 \wedge H[u] = u.next \wedge u.next > t$
$\Rightarrow$ {$H[v] = t + 1 \wedge H[u] > t$. Apply P3}
$H[v] < H[u]$

5.4.3. Correctness of the Implementation of Random Assignment

The random assignment
$H[v] :=?$ **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
   $\mathit{true} \mapsto z_{t+1} = z_t + v$, and
   $\mathit{true} \mapsto z_t = z_{t+1} + v$.
   We sketch a proof. From S3$'$ we can deduce that
   $(\forall\, i :: (\exists\, j : i \le j : z_{j+1} = z_j + v))$, and
   $(\forall\, i :: (\exists\, j : i \le j : z_j = z_{j+1} + v))$.

   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 $v.\mathit{low} \wedge v \notin b$ 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 $v \notin b\ \wedge v.\mathit{low}$ 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 [CM84]; 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 [CM84]. 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 [Mil89]. 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.

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

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 [Mis95a, Mis95b] 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 [Dij76]

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

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.

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,

$$p \;\mathbf{en}\; q \;\stackrel{\Delta}{=}\; (p \;\wedge\; \neg q \;\mathbf{co}\; p \;\vee\; q) \;\wedge\; (\exists\;\; s :: (p \;\wedge\; \neg q) \Rightarrow\; wp.s.q)$$

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) $$\frac{p \;\mathbf{en}\; q}{p \;\mapsto\; q}$$

(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

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

- 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.

References

[AO97] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs.* Springer–Verlag, 1997.

[APP93] Flemming Andersen, Kim Dam Petersen, and Jimmi S. Pettersson. Program Verification using HOL-UNITY. In *HUG'93: HOL User's Group Workshop*, volume 780 of *LNCS*, pages 1–17. Springer–Verlag, 1993.

[CM84] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, 1984.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison Wesley, 1988.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[Lam83] Leslie 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.

[Mil89] R. Milner. *Communication and Concurrency.* International Series in Computer Science, C. A. R. Hoare, Series Editor. Prentice-Hall International, London, 1989.

[Mis95a] Jayadev Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.

[Mis95b] Jayadev Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.

[Mis96] Jayadev Misra. A discipline of multiprogramming, work in progress, ftp access at `ftp://ftp.cs.utexas.edu/pub/psp/seuss/discipline.ps.Z`, 1996.

[Pau99] Lawrence C. Paulson. Mechanizing UNITY in Isabelle. Technical Report 467, Computer Laboratory, University of Cambridge, May 1999.

[SBW69] R. A. Scantlebury, K. A. Bartlett, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, May 1969.