# A Distributed Maximal Scheduler for Strong Fairness

Matthew Lang and Paolo A. G. Sivilotti

Department of Computer Science and Engineering
The Ohio State University, Columbus OH, USA, 43210-1277
{langma,paolo}@cse.ohio-state.edu

**Abstract.** Weak fairness guarantees that if an action is continuously enabled, it is executed infinitely often. Strong fairness, on the other hand, guarantees that actions that are enabled infinitely often (but not necessarily continuously), are executed infinitely often. In this paper, we present a distributed algorithm for scheduling actions for execution. Assuming weak fairness for the execution of this algorithm, the schedule it provides is strongly fair. Furthermore, this algorithm is maximal in that it is capable of generating *any* strongly fair schedule. This algorithm is the first strongly-fair scheduling algorithm that is both distributed and maximal.

## 1 Introduction

An action system models a distributed systems as a set of actions, each of which is either enabled or disabled. A fairness assumption controls the selection of actions from this set for execution. For example, weak fairness requires that an action that is enabled continuously be selected infinitely often while enabled. Strong fairness, on the other hand, requires that an action that is enabled infinitely often (but perhaps not continuously) be selected while enabled infinitely often.

The choice of fairness assumption is an important aspect of any model of behavior for a distributed system. Weak fairness is useful because of the minimal assumption it makes and the simple scheduling algorithm required to implement it: Select every action infinitely often. Strong fairness, on the other hand, is useful for simplifying the design of synchronization and communication protocols since it rules out the starvation of actions that are repeatedly enabled. While weak fairness reflects an asynchronous and independent scheduling of individual actions, strong fairness reflects some scheduling coordination to rule out certain pathological traces. The advantages of both models can be achieved by constructing a strongly-fair scheduler on top of an assumption of weak fairness.

A program is correct if it can exhibit *only* behaviors permitted by its specification. A correct program is maximal[3] if it can exhibit *all* behaviors permitted by its specification. Maximal programs are important for testing component-based systems because they prevent a component implementation from providing unnecessarily deterministic behavior and, in this way, masking errors in its clients. For example, if a scheduling algorithm is not maximal, it is incapable of generating some traces that are otherwise possible under the corresponding fairness assumption. These traces are no longer observable behaviors for the system built on top of such a limited scheduler.

In this paper, we present a strongly-fair scheduler, layered on top of a weak fairness assumption. This algorithm is distributed: it does not maintain a global set of enabled actions and it permits concurrent selection of independent actions. Furthermore, this algorithm is maximal: any trace that satisfies strong fairness is a possible behavior of the scheduler. To our knowledge, this is the first strongly-fair scheduler that is both distributed and maximal.

This paper is organized as follows. Section 2 describes maximality and fairness and illustrates how the properties one can derive about a system depend on the underlying assumption of fairness. Section 3 introduces the strong fairness scheduling problem and provides a formal specification of a strongly-fair scheduler. Section 4 contains our solution to the strong fairness problem; we prove its correctness in Section 5. Section 6 presents a proof of maximality of the algorithm. Section 7 examines related work and discusses the advantage of our maximal algorithm over existing strongly-fair scheduling algorithms, while Section 8 concludes the paper.

## 2 Maximality and Fairness

### 2.1 Maximality

A program is maximal if it is capable of generating any behavior permitted by a specification. This notion is similar to bisimulation [9, 11]. However, bisimulation involves relating artifacts with similar mathematical representations, while maximality relates a program text to a formal specification.

Proving the maximality of a program $P$ with respect to a specification $S$ is carried out in three stages. In the first stage, one defines a set of specification variables mentioned by $S$ and derives properties of traces of these variables from $S$. Next, one shows that an arbitrary trace $\sigma \in |S|$ satisfying these properties is a possible execution of an instrumented version $P'$ of $P$ (*Chronicle Correspondence*). Finally, one proves that every fair execution of $P'$ corresponds to a fair execution of $P$ (*Execution Correspondence*). Then, since $\sigma$ was a possible execution of $P'$ and every execution of $P'$ corresponds to an execution of $P$, $\sigma$ is a possible execution of $P$. Since $\sigma$ is an arbitrary trace, we can conclude that any trace satisfying $S$ is a possible execution of $P$.

Constructing $P'$ is carried out by adding new variables, assignments to new variables within existing actions, guards to existing actions, and actions that assign to only new variables. These additions ensure that safety properties of $P$ are safety properties of $P'$. The new variables typically include read-only *chronicle variables* that encode the trace $\sigma$ and auxiliary variables (e.g., variables that encode the current point in the computation).

In order to show *chronicle correspondence*, it suffices to show that (i) it is an invariant of $P'$ that the program state is identical to $\sigma$ and (ii) that the point in the computation eventually increases. Proving *execution correspondence* requires showing that (i) each additional guard in $P'$ is infinitely often true and (ii) the truth of each additional guard is preserved by the execution of every other action in $P'$. These properties ensure that each action is infinitely often executed in a state where the additional guard is true. Then, one can infer that every weakly-fair execution of $P'$ corresponds to a weakly-fair execution of $P$.

This proof technique was introduced in [4] and is used to carry out a proof of maximality for a variety of programs, including a task-scheduling problem, in [10]. The specification of the strongly-fair scheduling problem, however, is compositional in nature, so we follow the method for proving maximality in composed systems outlined in [8].

## 2.2 Fairness

In the rest of the section, we will illustrate the difference between weak and strong fairness using the following simple system.

```
Program    $fairness$
var        $b$ : boolean
           $x, y$ : int
initially  $b$
assign
  $A$ :      true $\longrightarrow$ $x := x + 1 \parallel b := \neg b$
  $B$ :      $b$ $\longrightarrow$ $y := x$
```

The program *fairness* has three program variables: two integers, $x$ and $y$, and a boolean variable, $b$. Action $A$ is always enabled and increments $x$ by one and sets $b$ to $\neg b$. Action $B$ is enabled in states where $b$ is true and assigns $y$ the value of $x$.

**Weak Fairness** Recall that weak fairness is the property that every action in a system is infinitely often selected to be executed. Under this assumption, one can prove the following properties of *fairness*: the safety property (i) $x$ increases by at most one in each step in the computation, and the progress properties (ii) $x$ eventually increases and (iii) $b$ eventually changes value. That is, in UNITY logic, $x = k$ **next** $|x - k| \leq 1$ (i), $x = k \leadsto x \neq k$ (ii), and $b = k \leadsto b \neq k$ (iii). Progress properties (ii) and (iii) follow from the fact that action $A$ is infinitely often executed in a state where it is enabled.

The only property we can derive about $y$ under weak fairness is the following safety property: at each step of the computation, $y$ either remains the same or changes to the value of $x$. We cannot derive any progress property about $y$; we cannot make the assumption that action $B$ will *ever* be selected for execution while it is enabled. To see why, observe that the following schedule $\sigma = \langle A, B, A, A, B, A, A, B, \ldots \rangle$ never executes $B$ in a state where $b$ is true. $\sigma$ is weakly-fair; both actions are infinitely often selected for execution.

**Strong Fairness** A strong fairness assumption allows us to prove stronger properties of *fairness*. Under strong fairness, any action that is infinitely often enabled must be infinitely often executed in a state in which it is enabled. Assuming strong fairness we can derive the same safety and progress properties about $x$ and $b$. However, we can now prove a stronger progress property about $y$. Since we know that $b$ infinitely often changes value, it follows that $b$ is infinitely often true ( $\mathbf{true} \leadsto b$ ) and action $B$ is infinitely often enabled.

Since $B$ is infinitely often enabled, the strong fairness assumption entails that it is infinitely often selected for execution in a state where $b$ holds. It follows that $x = y$ infinitely often ( $\mathbf{true} \leadsto x = y$ ).

### 2.3 Maximality and Scheduling

Maximal scheduling of systems is important. When reasoning about systems, we want our assertions to (i) be as strong as possible and (ii) hold in every possible execution of a system. A maximal scheduler ensures that the strongest properties we prove using the program text and a notion of fairness are the strongest properties of the actual system behavior. A non-maximal scheduler eliminates possible executions and therefore allows us to assert stronger properties that hold on only a subset of possible program executions.

To illustrate, consider a non-maximal strongly-fair scheduler that allows a process $u$ to disable any other action $v$ at most twice before $v$ is scheduled for execution in a state in which it is enabled. This scheduler is correct — actions which are infinitely often enabled are infinitely often executed in a state in which they are enabled. However, the scheduler clearly generates a small subset of possible correct schedules.

If we schedule the program *fairness* using this scheduler, we see action $A$ can execute at most four times before action $B$ must execute in a state in which it is enabled. This allows us to prove much stronger properties about $y$: $x - y \leq 4$ is a program invariant.

Although we can now assert a stronger program property, this is undesirable, for instance, in the case of testing. If one were to test the program *fairness* composed with such a non-maximal scheduler, one may be led to believe that $x - y \leq 4$ is indeed an invariant of the system. In fact, it would be impossible to design a test case to expose the fact that it is not.

Fortunately, maximal scheduling algorithms exist for weak fairness. The *unhygienic* algorithm presented in [8] is a maximal solution to the dining philosophers problem and can be used to generate maximal weakly-fair schedules.

In the rest of this paper, we create the first maximal strongly-fair scheduler. As a first step, we formalize the problem of strongly-fair scheduling.

## 3 Specification

In the following section we formally define the strong fairness scheduling problem. We formulate the problem as a distributed resource allocation problem, where each action corresponds to a process.

## 3.1 Description of the system

The system is comprised of a set of processes, each comprised of two components — a *client* layer and a *scheduler* layer. The system designer is given the specification of both layers and must design a refinement of the scheduler specification. Then, given any implementation meeting the client specification, the scheduler layer generates a strongly-fair schedule.

Each process has an associated *action* which modifies some state shared by all processes in the system. A process is *enabled* in some states. In these states, a client can be granted a *lock* to modify the shared state. The client must simultaneously increment a *count* when it changes the state, and in order to release a lock the client *must* execute its action and increase the count.

We say two processes $u$ and $v$ are neighbors if the execution of action $u$ or $v$ can affect the other's enabledness. If the scheduler guarantees that no two neighboring processes simultaneously hold a lock, the client layer guarantees that held locks are eventually relinquished.

It is the responsibility of the scheduler layer to manage locks. If a process is infinitely often enabled, the scheduler ensures that it is infinitely often granted a lock.

The composed system generates a strongly-fair schedule — if a process is infinitely often enabled, it infinitely often changes its count.

In order to make sure our specification is as general as possible, we neglect to formalize *state*, *action*, and *enabled*. Omitting formal definitions of these notions from the problem specification allows a solution to the strong fairness problem to be adapted to any domain in which processes intermittently desire access to shared resources (in this case, the shared program state) and desire to access the shared resource is predicated upon the behavior of other processes in the system (*neighbors*).

## 3.2 Formal Specification of the Strong Fairness Problem

The system is comprised of a set of processes $\mathcal{P}$. All processes have access to a symmetric *neighbor relation* $N \subseteq \mathcal{P}^2$. We define $N(u, v)$ if the execution of action $u$ or $v$ can affect the other's enabledness. This neighbor relation is irreflexive, it is never the case $N(u, u)$ [1].

In addition, each process $u \in \mathcal{P}$ has the following local variables:

- $u.count$, a count of the number of times action $u$ has executed. Since the execution of actions is atomic, there is no state in which an action is executing. Consequently, we require that when an action $u$ executes, $u.count$ is incremented.
- $u.enabled$, a boolean representing the enabledness of $u$.
- $u.lock$, a boolean. $u.lock$ is true if the process is free to execute its action.

[1] This is not to say that a process cannot enable/disable itself by executing its action; this is captured in the specification. The irreflexivity of $N$ only simplifies presentation.

## 3.3 Client Layer Specification

The client layer is responsible for execution of the action associated with a process. Intuitively, a client is "idle" until it is granted a lock. When granted a lock, the client eventually executes its action and increments its count, releasing the lock.

In UNITY logic, the specification is as follows:

Specification for client process $u$, where $v$ ranges over processes, $b$ and $a$ range over $\mathbb{B}$, and $k$ ranges over $\mathbb{N}$:

$$(\forall v : v \neq u : \textbf{constant } v.lock\ ) \tag{C0}$$

$$(\forall v : v \neq u : \textbf{constant } v.count\ ) \tag{C1}$$

$$(\forall v, b : \neg N(u,v) : \textbf{constant } v.enabled = b\ ) \tag{C2}$$

$$\begin{aligned}(\forall v, b, a, k : N(u,v) : &\ \textbf{stable } \neg u.lock \wedge u.count = k \\ &\wedge v.enabled = b \wedge u.enabled = a\ )\end{aligned} \tag{C3}$$

$$\begin{aligned}(\forall v, b, a, k : N(u,v) : &\ u.lock \wedge u.count = k \\ &\wedge v.enabled = b \wedge u.enabled = a \\ &\textbf{unless } \neg u.lock \wedge u.count = k+1\ )\end{aligned} \tag{C4}$$

Hypothesis: **invariant** $(\forall v : N(u,v) : \neg(u.lock \wedge v.lock)\ )$,
**invariant** $u.lock \Rightarrow u.enabled$

Conclusion: $u.lock \leadsto \neg u.lock$ (C5)

The first safety requirement on the client layer for a process $u$ is C0 which ensures $u$ does not assign to $v.lock$ for all other processes $v$. C2 ensures $u$ does not assign to $v.enabled$ for non-neighboring processes $v$. C3 dictates that when $u.lock$ does not hold, $u$ does not modify $u.count$ or $v.enabled$ for neighboring processes $v$. Finally, C4 dictates that when $u$ holds a lock, it continues to hold it until it increments $u.count$. C4 also ensures that $u$ may only change the enabledness of neighbors when it releases a lock and executes its action.

The progress property C5 is a conditional property; if the scheduling layer maintains the property that no neighboring processes hold a lock and holding a lock implies that a process is enabled, the client layer guarantees that a lock is eventually relinquished.

The mutual exclusion property and the invariant in the hypothesis of C5 are important; neighboring processes are permitted to modify the enabledness of their neighbors. If two neighboring processes $u$ and $v$ simultaneously hold locks, a process, say $u$, may execute its action and disable the other. Then $v$ is not guaranteed to become re-enabled and execute its action, releasing the lock.

This property also helps to further the goal of making our formulation of the strong fairness problem as general as possible; processes can perform computation requiring exclusive access to a shared resource while holding a lock.

### 3.4 Scheduler Layer Specification

The scheduler layer schedules actions for execution by granting client processes locks — when a client process holds a lock it is free to execute its associated action.

When a process becomes enabled, the scheduler should “negotiate” with neighboring scheduler processes in order to grant its client exclusive access to a lock.

The following are required properties of the scheduling layer for process $u$:

**constant** $u.count$ (S0)

**stable** $u.lock$ (S1)

**invariant** $u.lock \Rightarrow u.enabled$ (S2)

**invariant** $(\forall v : N(u,v) : \neg(u.lock \wedge v.lock))$ (S3)

Hypothesis: **true** $\rightsquigarrow u.enabled$, C0, C2, C3, C4, C5,

Conclusion: **true** $\rightsquigarrow u.lock$ (S4)

The first safety properties a scheduler must satisfy are S0 and S1, which dictate that the scheduler does not modify $u.count$ and once the scheduler grants a lock it cannot revoke it. The scheduler must also satisfy the hypothesis of C5 with S3 and S2, the latter of which dictates only enabled processes hold locks.

The progress property S4 is a conditional property which captures the notion of strong fairness. If a correct client process is infinitely often enabled the scheduler infinitely often grants the the process a lock.

### 3.5 Composed Specification

Given the *client* and *scheduler* specifications, the composed specification of the system satisfies the strong fairness property: if an action is infinitely often enabled, it is infinitely often executed in a state in which it is enabled. Or, if a process is infinitely often enabled, it infinitely often increases its execution count.

Formally, $client \parallel scheduler$ satisfies:

Hypothesis: **true** $\rightsquigarrow u.enabled$

Conclusion: $u.count = k \rightsquigarrow u.count = k+1$

## 4 Algorithm

Given the specification described in the previous section, we are tasked with creating a correct implementation of the *scheduler* component. However, we have an additional goal: our algorithm should be maximal with respect to the composed specification.

The challenge in designing a strongly fair scheduler lies in limiting concurrency — no correct scheduler *always* allows processes sharing a mutual neighbor to concurrently hold locks. As an illustration, consider the system with $\mathcal{P} = \{x, y, z\}$ and $\{\langle x, y\rangle, \langle x, z\rangle\}$ representing $N$. Let the behavior of action $y$ be to leave the system in a state where $y$ and $x$ are enabled and let the behavior of action $z$ be to leave the system in a state where $z$ is enabled and $x$ is disabled.

Now suppose the system is in a state where processes $y$ and $z$ are enabled and process $x$ is disabled. A scheduler that always allows processes sharing a mutual neighbor to concurrently hold locks will allow processes $y$ and $z$ to acquire locks. Now suppose process $y$ executes its action. Process $x$ is now enabled, but since $z$ still holds its lock and $N(x, z)$, $x$ may not acquire a lock. Now suppose process $z$ executes its action. The system is now in a state where $x$ is disabled and $y$ and $z$ are enabled. A scheduler which always allows processes with a mutual neighbor to concurrently hold locks allows this sequence of events to repeat continually, resulting in a schedule where $x$ is infinitely often enabled but never executed.

We will overcome this challenge by bounding the number of times a process allows its neighbors to hold locks concurrently. Although unintuitive, this will not affect the maximality of our solution: our scheduler will be capable of generating any schedule satisfying the strong fairness property. Furthermore, any correct algorithm satisfying the strong fairness scheduler specification can be viewed as a refinement of our algorithm.

### 4.1 Scheduler Design

In order to ensure the mutual exclusion property $S3$, we associate with each pair of neighboring processes $u, v$ a shared *lock token*, $tok(u, v)$. A process may only be granted a lock if it holds all of its shared tokens. A process $u$ also stores a read-only boolean array, $u.en$, storing the enabledness of its neighbors. A process $u$ notifies a neighbor $v$ of its enabledness by assigning to $v.en[u]$.

To ensure progress, each process $u$ is assigned a negative priority $u.ht$ which we call a process's height. A process is higher-priority than another if it has greater height. We require a process's height to be unique among its neighbors. Ties in priority between non-neighbors are broken by a static order on processes, say by process id. We will call lock tokens shared with higher-priority neighbors *high tokens* and lock tokens share with lower-priority neighbors *low tokens.*

A process only changes its priority after it has executed its action and released a held lock, at which point it lowers its height by a nondeterministically chosen finite but unbounded amount. A process which has released a lock holds all of its tokens until it lowers its height, at which point it gives up all its high tokens.

Processes always release tokens to higher-priority neighbors (*high neighbors*). An enabled process does not relinquish tokens to lower priority neighbors (*low neighbors*) and, in order to limit concurrency while still ensuring progress, a disabled process releases at most one low token.

In order to ensure there are no wait-cycles, a disabled process $u$ releases a low token only to its highest priority low neighbor, $v$. If $u.en[w]$ holds later for some higher-priority low neighbor $w$, $u$ retrieves the shared token from $v$ by assigning **true** to $v.en[u]$. It is guaranteed to eventually receive the token as processes always relinquish high tokens.

The variables of process $u$ include:

– $u.en$, an boolean array containing an entry for all neighbors $v$. A neighbor $v$ assigns true to $u.en[v]$ if it is either enabled or it needs to retrieve a low token from $v$.
– $u.gate$, a boolean variable. If $u.gate$ is true $u$ is free to exchange tokens with its neighbors or grant itself a lock. When $u$ grants itself a lock, it sets $u.gate$ to false. Upon releasing a lock, the process sets $u.gate$ to true, lowers its height, and releases its high tokens.

The following predicates are associated with a process $u$:

– $u.sendtok.v$ for all neighbors $v$ of $u$. $u.sendtok.v$ is true if a process $u$ should send its shared token to process $v$. $u.sendtok.v$ is true if $v$ is a high neighbor of $u$ and either $u.en[v]$ or $\neg u.enabled$. $u.sendtok.v$ is true when $v$ is a low neighbor of $u$ and $v$ is the highest-priority among all low neighbors of $u$, $w \neq v$, for which $u.en[w] = \textbf{true}$.

$$\begin{aligned}
&u.sendtok.v \equiv tok(u,v) = u \\
&\quad \wedge (\ (\ \ u.ht < v.ht \wedge (\neg u.enabled \vee u.en[v])) \\
&\qquad \vee (\ \ u.ht > v.ht \wedge u.en[v] \\
&\qquad\quad \wedge\ (\forall w\ :\ N(u,w) \wedge w.ht < u.ht\ :\ tok(u,w) = u\ ) \\
&\qquad\quad \wedge\ v.ht =\ (\textbf{Max}\, w\ :\ N(u,w) \wedge w.ht < u.ht \wedge u.en[w]\ :\ w.ht\ )))
\end{aligned}$$

– $u.maylock$. $u.maylock$ is true if $u$ holds all its shared tokens and is enabled.

$$u.maylock \equiv u.enabled \wedge\ (\forall v\ :\ N(u,v)\ :\ tok(u,v) = u\ )$$

– $u.retr.v$ for all neighbors $v$ of $u$. $u.retr.v$ is true if $u$ has granted a low token to $v$ and now some low neighbor $w$ of $u$ is enabled.

$$\begin{aligned}
&u.retr.v \equiv tok(u,v) = v \\
&\qquad\quad \wedge\ (\exists w\ :\ N(u,w) \wedge u.en[w]\ :\ v.ht < w.ht < u.ht\ )
\end{aligned}$$

The UNITY program in Figure 1 implements the *scheduler* layer of the strong fairness specification for a process $u$. Actions $U_{u,v}$ and $T_{u,v}$ should be understood as quantified across all neighbors $v$ of $u$.

Action $U_{u,v}$ updates $v.en[u]$ by assigning true if $u.enabled$ or $u.retr.v$ and assigns false otherwise. Action $T_{u,v}$ sends a token to $v$ if $u$ is free to exchange tokens and $u.sendtok.v$ is true. Action $L_u$ grants a lock to process $u$ and stops further communication by setting $u.gate$ to false. Finally, action $D_u$ frees $u$

```
Program  $SF_u$
var      $u.enabled, u.gate, u.lock$ : bool
         $u.ht$ : integer
         $u.en$ : array of bool
initially  $(\forall v \,:\, N(u,v) \,:\, u.ht \neq v.ht\,)$
         $\neg u.lock$
         $u.gate$
assign
  $U_{u,v}$    $\mathbf{true} \longrightarrow v.en[u] := u.enabled \,\vee\, u.retr.v$
  $T_{u,v}$    $u.sendtok.v \,\wedge\, u.gate \longrightarrow tok(u,v) := v$
  $L_u$      $u.maylock \,\wedge\, u.gate \longrightarrow u.lock := \mathbf{true}$;
               $u.gate := \mathbf{false}$
  $D_u$      $\neg u.lock \,\wedge\, \neg u.gate \longrightarrow u.gate := \mathbf{true}$;
               $u.ht$ :=? st $u.ht < u.ht' \,\wedge\, (\forall v \,:\, N(u,v) \,:\, u.ht \neq v.ht\,)$;
               $(\,\| \; v \,:\, N(u,v) \,\wedge\, u.ht < v.ht \,:\, tok(u,v) := v\,)$
```

**Fig. 1.** Maximal Strong Fairness Scheduling Algorithm

to exchange tokens with neighbors, lowers its height by a finite but unbounded amount, and releases $u$'s high tokens. $D_u$ is enabled only after a process has relinquished a lock and executed its action.

*Note:* In the algorithm $SF$, we assume that a process can read the height of its neighbors. In practice, this can be implemented as a refinement of this algorithm by encoding process height on shared tokens and associating with each process $u$ a new variable $u.lowtoken$ which is assigned the height of the process $u$ relinquishes a low token to. In this paper, we omit these refinements as they only serve to complicate reasoning about the behavior of $SF$.

## 5 Correctness of $SF_u$

In this section we present an argument for the correctness of program $SF_u$ with respect to the *scheduler* specification.

The safety properties S0, S1, and S2 follow directly from the program text. The invariant that no two neighbors hold locks simultaneously follows from the fact that a process must hold all of its shared tokens to grant itself a lock and that a process does not relinquish its tokens while it holds a lock. The progress property (that an infinitely often enabled process holds a lock infinitely often) requires a more thorough treatment.

In the interest of space and reader intuition we present an informal argument for the progress property. A detailed proof can be found in Appendix A and follows the same general structure as the argument presented below.

In order to prove S4, we show: (i) the system is free from deadlock, (ii) a process with no higher priority neighbors which becomes enabled eventually acquires a lock, (iii) a continually enabled process eventually is granted a lock, and finally (iv) an infinitely often enabled process eventually is granted a lock.

*(i) Freedom From Deadlock.* We argue by contradiction and assume the system is in a state of deadlock. If the system is deadlocked, no process changes enabledness, as processes are only enabled/disabled by the execution of an action.

The priority relation on processes forms a total order so if the system is in a state of deadlock, there must exist a highest-priority process $u$ that is continually enabled but never executed. Since there are no higher-priority enabled processes, $u$ is guaranteed to eventually acquire all its high and low tokens and acquire a lock.

*(ii) A process that becomes enabled at the top of the order eventually is granted a lock.* Let $u$ be a process such that $u$ has no higher priority neighbors which become enabled. It is easy to see from the program text that $u$ is missing at most one low token. Now, if $u$ holds all its tokens, it will continue to hold them as enabled processes do not release tokens to low neighbors. In this case $u$ eventually is granted a lock.

In the other case $u$ is missing a low token. Now, because $u$ is enabled and enabled processes do not give up low tokens, it must have been the case that $u$ was disabled when it gave up a low token. Since $u$ has no higher-priority neighbors, it must have been the case that its low neighbor executed its action and enabled $u$. After a process executes its action and releases a lock, the only enabled action is $D_u$, which releases all high tokens. So $u$ is guaranteed to eventually hold all its tokens in a state where it is enabled, which is the first case.

*A metric:* In order to show a continually enabled process is eventually granted a lock and an infinitely often enabled process is granted a lock infinitely often, we define a *metric* for each process $u$ called $u.M$.

$u.M$ is the sum of the difference in height between $u$ and all processes with higher priority than $u$ that are reachable from $u$ by following the neighbor relation through higher-priority processes. More formally we define the set $u.ab = \bigcup u.ab_n$ where $u.ab_n$ is defined by recursion:

$$u.ab_0 = \{\, v \mid u.ht < v.ht \wedge N(u,v) \,\}$$
$$u.ab_{i+1} = \{\, v \mid (\exists w \,:\, w \in u.ab_i \,:\, N(v,w) \wedge u.ht < w.ht \,) \,\}$$

Then $u.M = (\sum v \,:\, v \in u.ab \,:\, v.ht - u.ht \,)$.

By definition, $u.M$ is bounded below by zero when $u.ab = \emptyset$ and $u$ has no higher-priority neighbors. Furthermore, $u.M$ is non-increasing unless $u$ acquires a lock and lowers its height.

To show the progress property, we demonstrate that if $u.M = k$ and $u$ is infinitely often enabled, eventually either $u.M < k$ or $u.lock$. Since $u.M$ is bounded below and non-increasing unless $u$ acquires a lock, eventually $u$ acquires a lock.

*(iii) A process cannot remain continually enabled.* Again we argue by contradiction and assume there is some process $u$ that is continually enabled and

never executes. If $u$ is continually enabled, $u$ will eventually acquire all its low tokens so there must be processes in $u.ab$ that are infinitely often enabled which prevent $u$ from acquiring its high tokens.

Fix $v$ to be the highest-priority process in $u.ab$ that is infinitely often enabled. Since $v$ is the highest-priority such process, there is a point in the computation beyond which no process with higher priority than $v$ becomes enabled. By a similar argument as in the case of a process at the top of the order, $v$ eventually acquires a lock and eventually lowers its height, decreasing $u.M$.

Since we fixed $v$ to be the highest priority infinitely often enabled process in $u.ab$ and $v$ eventually lowers its height we can apply the same argument to the "new" highest priority infinitely often enabled process in $u.ab$. Then, eventually $u.ab = \emptyset$ and $u$ must be able to acquire its shared tokens and acquire a lock, which is a contradiction.

*(iv) An infinitely often enabled process eventually acquires a lock.* If a process $u$ is continually enabled, it eventually acquires a lock. Thus, we only have to consider the case where $u$ becomes disabled infinitely often.

So assume eventually $\neg u.enabled$. We show either $u.M$ decreases or $u$ acquires a lock.

There are two cases. In the first a higher priority neighbor $v$ executes its action and disables $u$. In this case, the metric decreases, as $v$ must eventually lower its height.

In the second case, a lower priority neighbor disables $u$. By our assumption $u$ must become enabled again. If a higher priority neighbor enables $u$, then the metric decreases. If a lower priority neighbor $v$ enables $u$, no other lower priority neighbor can disable $u$ as an enabled process does not release low tokens. Then either $u$ will eventually acquire its tokens and be granted a lock or a higher priority neighbor will disable $u$, decreasing the metric.

In either case, either $u.M$ is decreased or $u$ is eventually granted a lock.

Then a process that is infinitely often enabled and disabled eventually rises in the order.

It follows that if $u.enabled$ is infinitely often true and $u.enabled$ holds in a state where $u.M = k$, eventually either $u.lock$ or $u.M < k$. The interested reader should refer to the detailed proof in Appendix A for a thorough argument of this property.

It follows from this property and the previous properties described that if $u$ is infinitely often enabled, $u$ acquires a lock infinitely often.

## 6 The Maximality of *SF*

In this section we present an outline of the proof of maximality for $SF$. Our goal is to show that all schedules permitted by the strong-fairness specification are possible schedules of $SF$.

Since maximality is noncompositional, we use the rely-guarantee style proof outlined in [8] as a template. This method for proving the maximality of composed systems involves stipulating that other processes in the system satisfy certain properties beyond their formal specification and proving the maximality of the composed system using these properties. These additional properties entail that the client process our system is composed with is maximal and can be constrained in a way to establish its maximality.

In the interest of space and clarity, we only present the intuition behind the proof of maximality in this section. The interested reader should refer to Appendix B for a thorough proof of maximality of $SF$.

In this section we reverse the priority relation described in Section 4 to clarify presentation and allow the reader to maintain an intuition about the behavior of the constrained system. In Section 4 a process was higher priority if it had a greater height and processes lowered their priority by lowering their height. In this section, we will reverse this — a process has higher priority if it has a *lesser* height, thus a process *lowers* its priority by *increasing* its height. In our description of the behavior of the constrained program $SF'$, the motivation for this modification will become clear.

Formally $u.sendtok.v$, $u.retr.v$, and $u.maylock$ become:

$$\begin{aligned}
&u.sendtok.v \;\equiv\; tok(u,v)=u\\
&\quad\wedge\;(\quad(\quad u.ht > v.ht \wedge (\neg u.enabled \vee u.en[v]))\\
&\qquad\vee\;(\quad u.ht < v.ht \wedge u.en[v]\\
&\qquad\quad\wedge\;(\forall w : N(u,w)\wedge w.ht > u.ht : tok(u,w)=u\;)\\
&\qquad\quad\wedge\; v.ht = (\,\mathbf{Min}\, w : N(u,w)\wedge w.ht > u.ht \wedge u.en[w] : w.ht\;)))
\end{aligned}$$

$$u.maylock \;\equiv\; u.enabled \wedge\; (\forall v : N(u,v) : tok(u,v)=u\;)$$

$$\begin{aligned}
&u.retr.v \;\equiv\; tok(u,v)=v\\
&\qquad\wedge\;(\exists w : N(u,w)\wedge u.en[w] : v.ht > w.ht > u.ht\;)
\end{aligned}$$

## 6.1 Proving the Maximality of *SF*

In order to prove $SF$ is a maximal implementation of the strong-fairness specification, we need to show that any trace satisfying the strong-fairness specification is a possible trace of $SF$. In order to accomplish this, we create a constrained program $SF'$ from $SF$ that accepts as input any trace $\sigma$ satisfying the strong-fairness specification. We then show that at each point $i$ in the trace $\sigma$, the state of the system is exactly that of $\sigma_i$. This establishes $\sigma$ as a possible execution of $SF'$.

Next we need to show that any fair execution of $SF'$ corresponds to a fair execution of $SF$. Then, since any trace $\sigma$ satisfying the specification of the strong fairness problem is a possible execution of $SF'$, any trace satisfying the strong fairness problem is a possible execution of $SF$.

However, this simple view is not quite complete. Since we want to show that *any* schedule of action executions is a possible behavior of the composed system, we need to stipulate that the client process composed with $SF$ satisfies some additional requirements. Namely, we will require that this client process can be constrained to produce $\mathit{client}'$ which, when composed with $SF'$, can take the "steps" in the computation that $\sigma$ dictates. *i.e.,* if at some point $i$ in $\sigma$ some process $u$ is to execute and enable/disable itself or its neighbors, $\mathit{client}'$ can compute this step. The additional requirements are that the client process is maximal, $\mathit{client}'$ satisfies the safety properties of the $\mathit{client}$ specification, and that $\mathit{client}'$ is created in a way that ensures the correspondence between executions of $\mathit{client}'$ and the client process.

In order to compute $\sigma$, we will introduce a variable $p$ shared by $\mathit{client}'$ and $SF'$ that marks the current point in the trace. Then if we can prove (i) it is invariant that $u.\mathit{count} = C_p^u$ and $u.\mathit{enabled} = E_p^u$ for all $p$ and (ii) the point $p$ eventually increases, it follows that $\sigma$ is a possible execution of $SF' \parallel \mathit{client}'$.

## 6.2 A strong fairness trace

Let $\sigma$ be a stutter-free sequence of tuples $\sigma = \langle \sigma_0, \sigma_1 \ldots \rangle$ representing the state of processes in an execution satisfying the strong-fairness specification. $\sigma_i = \langle E, C \rangle_i$ is a tuple containing two arrays, $E_i$ and $C_i$, representing the enabledness and *count* of processes in state $\sigma_i$. That is, $E_i^u = \textbf{true}$ if $u.\mathit{enabled}$ in $\sigma_i$ and $C_i^u = k$ if $u.\mathit{count} = k$ in $\sigma_i$. $\sigma$ is stutter-free in that each tuple in the sequence is differs from the previous by at least one element, *unless* the execution is in a state of quiescence (each processes is disabled forever).

Since $\sigma$ is a correct trace of the strong fairness scheduling problem, it obeys certain properties. Namely, it satisfies the following: in subsequent states in $\sigma$, at most one process changes count (by incrementing it by one) and if a process changes enabledness, a process must change count. Also, if a process is infinitely often enabled in the trace, it infinitely often changes its count.

Given a trace $\sigma$, we create an isomorphic trace $\sigma'$ by inserting a stuttering-state in between every $\sigma_i$ and $\sigma_{i+1}$. That is, $\sigma'_0 = \sigma_0$ and $\sigma'_{i+1}$ is $\sigma'_i$ if $i$ is even and is $\sigma_{(i+1)/2}$ if $i$ is odd.

The motivation for introducing stuttering-states will become clear when we describe our approach for ensuring that $SF' \parallel \mathit{client}'$ computes $\sigma$.

## 6.3 Requirements of $\mathit{client}'_u$

We require that a client process $u$ can be constrained to produce $\mathit{client}'_u$. The requirements on $\mathit{client}'_u$ are as follows:

- $\mathit{client}'_u$ is produced from the client process by only adding new variables, assignments to new variables, and new guards referencing new and existing program variables. Furthermore, if random assignments in the client process are replaced with deterministic assignments, we require that the assigned value satisfy the predicate on the random assignment. These requirements ensure that $\mathit{client}'_u$ satisfies the safety properties of the client process.
- The additional guards of $\mathit{client}'_u$ are infinitely often true and the enabledness of each guard is preserved by the execution of any other action in the system.

- At each point $p$ in the computation, it is invariant that $u.enabled = E_p^u$ and $u.count = C_p^u$.
- $client'_u$ does not assign to $\sigma$ and only changes $p$ by at most one.
- If $SF'_u$ ensures $u$ holds a lock at a point $p = k$ in the trace where $C_k^u \neq C_{k+1}^u$ (*i.e.,* $u$ executes its action), $client'_u$ guarantees that $p$ is incremented and the lock is released.

These requirements on the client process ensure that $client'$ will compute the transitions dictated by $\sigma$. It is then the obligation of $SF'$ to ensure that processes hold locks when $\sigma$ dictates $u$ executes its action and increments its count.

## 6.4 The Constrained Program $SF'_u$

In the constrained program $SF'_u$ we introduce the following objects not found in $SF_u$: the input trace $\sigma$ and the point $p$, a function $u.next$ to compute the next point at which process $u$ executes its action and increments its count, a predicate $u.done$ to indicate whether or not $u$ increments its count again after the current point in the computation, and a predicate $u.quiet$ which indicates whether or not $u$ is enabled after the current point in the computation.

Formally, $u.quiet$, $u.done$, and $u.next$ are defined as the following.

$$u.quiet \equiv (\forall i : i \geq p : \neg E_i^u)$$

$$u.done \equiv (\forall i : i \geq p : C_i^u = C_{i+1}^u)$$

$$\begin{aligned} u.next = {} & (\mathbf{Min}\, i : i \geq p : C_i^u \neq C_{i+1}^u) \text{ if } \neg u.done \\ & (\mathbf{Min}\, i : i \geq p : (\forall j : j \geq i : \neg E_j^u \wedge \\ & \quad (\forall v : v \neq u : v.ht \neq j))) \text{ otherwise} \end{aligned}$$

Figure 2 shows the instrumented program.

The key property that follows from this instrumentation is that a process $u$'s height corresponds to the next point in the computation when $u$ increments its count. At that point, $u$ is the highest priority enabled process among its neighbors (*i.e., lowest height*). Any process with a higher priority (lower height) than $u$ at that point is in a state of quiescence.

If a process $u$ has executed for the last time, we set its height to be after the last point in the trace that it is enabled. This ensures that any process that executes and enables/disables $u$ will be higher priority than $u$ until $u$ is quiescent. Such a point is guaranteed to exist by the assumption that the process has executed for the last time; if no such point exists, the process must be infinitely often enabled (and therefore execute again).

The motivation for the introduction of stutter states in $\sigma$ is to ensure that a process that never executes again can be assigned a unique height. If $\sigma$ were stutter-free, it is not guaranteed that such a point exists.

```
Program $SF'_u$
var        $u.enabled, u.gate, u.lock$ : bool
           $u.ht$ : integer
           $u.en$ : array of bool
initially  $p = 0$
           $u.enabled = E^u_p$
           $\neg u.done \Rightarrow u.ht = u.next$
           $u.done \Rightarrow u.ht \geq \mathbf{min}\, i\, (\forall j : i \leq j : \neg E^u_j)$
           $(\forall v : N(u,v) : u.ht \neq v.ht)$
           $\neg u.lock$
           $u.gate$
assign
  $U'_{u,v}$ $\mathbf{true} \longrightarrow$
           $\mathbf{true} \longrightarrow v.en[u] := u.enabled \vee u.retr.v$
  $T'_{u,v}$ $\mathbf{true} \longrightarrow$
           $u.sendtok.v \wedge u.gate \longrightarrow tok(u,v) := v$
  $L'_u$   $(u.ht = p \wedge \neg u.done) \vee u.quiet \longrightarrow$
           $u.maylock \wedge u.gate \longrightarrow u.lock := \mathbf{true};$
               $u.gate := \mathbf{false}$
  $D'_u$   $\mathbf{true} \longrightarrow$
           $\neg u.lock \wedge \neg u.gate \longrightarrow u.gate := \mathbf{true};$
               $u.ht := u.next;$
               $(\parallel v : N(u,v) \wedge u.ht < v.ht : tok(u,v) := v)$
  $Q'$     $\sigma_i = \sigma_{i+1} \longrightarrow p := p + 1$
```

**Fig. 2.** Constrained Strong Fairness Scheduling Algorithm

A key invariant of $SF'_u$ is that if $\neg u.done$ and $u.gate$ hold, $u.ht = u.next$.

$SF'_u$ inherits the safety properties of $SF_u$ as guards are only strengthened and existing program variables are not assigned to, except for the replacement of the random assignment to $u.ht$ with a deterministic assignment. However, at the point of the assignment to $u.ht$, $u.next > u.ht$ and is unique by definition of $u.next$ and the properties of $\sigma$.

### 6.5 Proof Sketch of the Maximality of *SF*

There are two main obligations to dispatch: (i) $SF' \parallel client'$ computes $\sigma$ and (ii) every fair execution of $SF' \parallel client'$ corresponds to a fair execution of the original system.

(i) is proved by showing $u.enabled = E^u_p \wedge u.count = C^u_p$ is an invariant of the system and $p = k \leadsto p = k+1$. (ii) requires showing that the truth of each additional guard in the system is preserved by the execution of any other action and that each additional guard is infinitely often true. Then each additional guard is executed infinitely often in a state where it is true, corresponding to a fair execution of the original program.

*Proving the invariant:* The invariant in (i) is initially true by the initially predicates in $SF'_u$. Also, each action of $SF'$ maintains the invariant as no action assigns to the trace, $u.enabled$, or $u.count$ and the only action that assigns to $p$ only increments $p$ in a stuttering state. Thus, since the invariant is also a property of $client'_u$, it is an invariant of the composed system.

*Proving* $p = k \leadsto p = k + 1$ *:* There are two cases to consider — the case where the current point is a stuttering-state, in which action $Q'$ increments $p$, and a non-stuttering state. In a non-stuttering state, there exists some process $u$ such that $C^u_p = C^u_{p+1}$. It was a requirement on $client'_u$ that if $u$ holds a lock in such a state, $client'_u$ eventually increments $p$. It is the responsibility of $SF'$ to ensure that in such a state process $u$ eventually acquires a lock.

At that point in the computation $u.next = p$ and $\neg u.done$ holds. Without loss of generality, assume $u.gate$ holds as well, so by the invariant of $SF'_u$, $u.ht = p$. Also, by the way height is assigned $u$ is the highest priority process among all its neighbors that are enabled. So $u$ eventually acquires all its tokens and acquires a lock.

*Proving the stability of additional guards:* Since $client'_u$ is required to satisfy this property, it suffices to show that the guard of $L'_u$ is not falsified by any action of $SF'_u$. It is easy to see that the only actions which might affect the truth of the additional guard of $L'_u$ are $Q'$, which assigns to $p$, and $D'_u$, which assigns to $u.ht$.

Since $u.quiet$ is stable, neither $Q'$ nor $D'_u$ can falsify it. Now, if $u.ht = p \wedge \neg u.done$ hold, it is implied by the invariant that $\sigma_p \neq \sigma_{p+1}$, so $Q'$ is disabled in such a state. If action $D'_u$ is enabled, $\neg u.lock \wedge \neg u.gate$ holds. Then since $\neg u.lock \wedge \neg u.gate$ holds, $client'_u$ must have released a lock and incremented the point, which implies $u.ht < p$. So if $L'_u$ is enabled, $D'_u$ is not.

*Proving additional guards are infinitely often true:* Again, since this was a requirement of $client'_u$, we only need consider the guard of $L'_u$. Now, since $u.quiet$ is stable and if $u.done$ ever holds, eventually $u.quiet$ holds, it suffices to show that $u.ht = p \wedge \neg u.done$ is infinitely often true if $\neg u.quiet$ is an invariant of the trace. Assuming $\neg u.quiet$ is an invariant of the trace, $\neg u.done$ is an invariant of the trace as well.

Now, if $\neg u.gate$ holds at any point in the computation, it must be the case $\neg u.lock$ holds as well and both continue to hold until eventually $D'_u$ is executed. The execution of $D'_u$ in an enabled state ensures $u.gate$ holds. Then the invariant of $SF'_u$ dictates that $u.ht = u.next$ and, since $u.next \geq p$ and $p = k \leadsto p = k + 1$, eventually $u.ht = p$. Thus, the additional guard of $L'_u$ is infinitely often true.

*The Maximality of* $SF$ *:* The preceding arguments establish that any trace $\sigma$ satisfying the strong-fairness specification is a possible execution of $SF$ composed with a client process meeting the requirements described. It follows that $SF$ is a maximal strongly-fair scheduler.

## 7 Discussion

Fairness is a well-researched and developed notion in existing literature, both in terms of interaction fairness [1] and in terms of selection of actions in nondeterministic guarded command programs [7].

Although a large body of work surrounds fairness issues, our algorithm is unique in that it is the first solution for strongly-fair scheduling of atomic actions that is both maximal and distributed.

In [6] Karaata gives a distributed self-stabilizing algorithm for the strongly-fair scheduling of atomic actions under weak fairness. However, a key property of the algorithm is that an action $u$ can disable another action $v$ at most twice before action $v$ must execute. This corresponds exactly to the situation described in Section 2 and is clearly not maximal. In addition although there is no notion of a "lock," the algorithm precludes two processes with a shared neighbor from having the "right" to execute their actions. Although this does not affect the possible schedules the algorithm can generate, it in some sense limits the algorithm from being generalized to a situation where the mutual exclusion property of the strong-fairness specification can benefit processes (*e.g.,* processes perform some computation before releasing the lock and affecting their neighbors). Then the concurrency of non-neighboring processes holding locks is a valuable property.

We should note that Karaata's algorithm has the advantage of being self-stabilizing, whereas ours does not. Also, Karaata provides a brief message complexity analysis of the algorithm while we make no claims regarding the message complexity of our algorithm.

In [5], Joung develops a criterion for implementability of fairness notions for multiparty interactions. If a fairness notion fails to meet the criterion, then no deterministic scheduling algorithm can meet the fairness requirement in an asynchronous system. In the general case, both strong interaction fairness and strong process fairness fail to meet the criterion. Our problem is outside of this result.

The dining philosophers problem proposed by Dijkstra [2] seems superficially similar (as also pointed out in [6]) to the strong-fairness problem in that one can map the state $\neg u.enabled$ to *thinking*, $u.enabled \wedge \neg u.lock$ to *hungry*, and $u.enabled \wedge u.lock$ to *eating*. However, in the dining philosophers problem, a process becomes hungry autonomously, not as a result of the behavior of other processes in the system. Furthermore, processes remain hungry until the arbitration layer affects a change in state to eating.

The possibility for processes to affect the enabledness of neighboring processes adds complexity to the strong fairness scheduling problem. For example, a solution to the dining philosophers problem can maintain an invariant that if a process holds a request from a neighbor, that neighbor is hungry. No corresponding invariant can be shown for a solution to the strong-fairness problem without synchronization between a process and its neighbor's neighbors.

## 8 Conclusions

In this work we presented a formal specification of the distributed strong fairness scheduling problem and described a maximal solution $SF$ to the problem.

The importance of a maximal scheduling algorithm was discussed in detail in Section 2, making the maximality of the $SF$ algorithm a key contribution of the work. The maximality of $SF$ also implies that any correct implementation of the strong-fairness specification is a refinement of the $SF$ algorithm in that any correct algorithm's behavior is a subset of the behavior of $SF$.

## References

1. K. R. Apt, N. Francez, and S. Katz. Appraising fairness in distributed languages. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 189–198, New York, NY, USA, 1987. ACM Press.
2. Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.
3. Rajeev Joshi and Jayadev Misra. Maximally concurrent programs. *Formal Aspects of Computing*, 12(2):100–119, 2000.
4. Rajeev Joshi and Jayadev Misra. Toward a theory of maximally concurrent programs. In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 319–328, New York, NY, USA, 2000. ACM Press.
5. Yuh-Jzer Joung. On fairness notions in distributed systems, part I: A characterization of implementability. *Information and Computation*, 166:1–34, 2001.
6. Mehmet Hakan Karaata. Self-stabilizing strong fairness under weak fairness. *IEEE Trans. Parallel Distrib. Syst.*, 12(4):337–345, 2001.
7. Leslie Lamport. Fairness and hyperfairness. *Distrib. Comput.*, 13(4):239–245, 2000.
8. Matthew Lang and Paolo A. G. Sivilotti. The maximality of unhygienic dining philosophers. Technical Report OSU-CISRC-5/07-TR39, The Ohio State University, May 2007.
9. R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
10. Jayadev Misra. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications.* Sprinver-Verlag, New York, NY, USA, 2001.
11. David Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK, 1981. Springer-Verlag.

## A Full Proof of Correctness for $SF_u$

In this section we provide a full proof that $SF_u$ satisfies the progress property S4.

In order to carry out the proof, we first present some properties of $SF_u$, followed by an observation and a few lemmas from which S4 follows.

First, we show the system is free from deadlock. Next, we prove that an enabled process can make individual progress by showing that the highest priority process among all is neighbors can acquire a lock and that a continuously enabled process eventually is granted a lock. Then we develop a *metric* for $u$ that is bounded below and decreases each time $u$ is enabled and then later disabled, following the intuition presented Section 5.

The following properties follow directly from the text of $SF_u$ and are presented without proof.

$$\textbf{invariant } ( \sum v : tok(u,v) = v \wedge v.ht < u.ht : 1 ) \leq 1 \qquad \text{(SF0)}$$

$$\textbf{invariant } u.lock \Rightarrow \neg u.gate \qquad \text{(SF1)}$$

$$\textbf{invariant } \neg u.gate \Rightarrow ( \forall v : N(u,v) : tok(u,v) = u ) \qquad \text{(SF2)}$$

$$\textbf{invariant } ( \exists v : N(u,v) : tok(u,v) = v) ) \Rightarrow u.gate \qquad \text{(SF3)}$$

$$u.gate \textbf{ unless } u.lock \qquad \text{(SF4)}$$

$$\textbf{stable } u.ht = k \wedge u.gate \qquad \text{(SF5)}$$

$$\textbf{transient } (u.enabled \vee u.retr.v) \neq v.en[u] \qquad \text{(SF6)}$$

$$\textbf{transient } \neg u.lock \wedge \neg u.gate \qquad \text{(SF7)}$$

Next, we present the following observation:

**Observation A1** *If $\neg u.gate$ and $u.ht = k$, eventually $u.ht < k$ and $( \forall v : N(u,v) \wedge u.ht < v.ht : tok(u,v) = u )$.*

*Proof.* Assume $\neg u.gate \wedge u.ht = k$. We can assume $\neg u.lock$ as if $u.lock$ holds, by S2 and S3, the client progress property C5 holds and eventually $\neg u.lock$. Then, eventually $\neg u.lock \wedge \neg u.gate \wedge u.ht = k$. Then action $D_u$ is the only enabled action and by weak fairness $D_u$ it is eventually executed. As a result, $u.gate \wedge u.ht < k \wedge ( \forall v : N(u,v) \wedge u.ht < v.ht : tok(u,v) = v )$.

We begin the proof of progress for $SF_u$ by first showing that if a process is enabled, some process eventually holds a lock.

**Lemma A1** *(Weak progress) If a process is enabled, some process eventually holds a lock.*

*Proof.* By contradiction. Assume at least one processes is enabled and does not hold a lock. For a contradiction, assume the system is in a state of deadlock. That is, there is no process that eventually acquires a lock.

First notice that by property C3, if no process in the system holds a lock, no process in the system will change enabledness.

As the only action by which processes acquire locks is action $L$, no process $u$ ever executes $L_u$ in a state where $u.\mathit{maylock} \wedge u.\mathit{gate}$ hold. For a contradiction it suffices to show there exists a process $u$ such that $u.\mathit{maylock} \wedge u.\mathit{gate}$ eventually and continuously holds.

Without loss of generality we may assume that $u.\mathit{gate}$ is true for all processes $u$. If $\neg u.\mathit{gate}$ holds for any process $u$, $\neg u.\mathit{lock} \wedge \neg u.\mathit{lock}$ holds by assumption $\neg u.\mathit{lock}$ for all processes $u$. Then eventually $u.\mathit{gate}$ holds by SF7. Furthermore, $u.\mathit{gate}$ continuously holds, by assumption the system is in deadlock and SF4.

Then there must be processes that are enabled for which $\mathit{maylock}$ does not hold.

Consider the highest-priority enabled process, say $u$. There are two cases:

*Case (a):* $u$ has no higher-priority neighbors. In this case, by SF6, eventually $v.\mathit{en}[u] = \mathbf{true}$ for all $v$ such that $N(u, v)$. Furthermore, since $u$ is enabled and the system is in deadlock, $v.\mathit{en}[u] = \mathbf{true}$ continuously. Since by assumption $v.\mathit{ht} < u.\mathit{ht}$, $v.\mathit{sendtok}.u$ is true and eventually $u$ will hold all its tokens. Notice that in this case, $u.\mathit{sendtok}.v$ is false as $u$ is enabled and higher-priority than all its neighbors. Then $u$ will hold all its tokens and $u.\mathit{maylock} \wedge u.\mathit{gate}$ will hold continuously. Contradiction.

*Case(b):* $u$ has at least one higher-priority neighbor. By the same argument as in case (a), $u$ eventually holds all its low tokens. Then there exists some $v$ will not relinquish a token to $u$ such that $N(u, v)$ and $u.\mathit{ht} < v.\mathit{ht}$. By SF6, eventually $v.\mathit{en}[u] = \mathbf{true}$ and $u.\mathit{en}[v] = \mathbf{false}$. Furthermore, can assume by SF6 and the fact that no process changes state that $v.\mathit{en}[w] = w.\mathit{enabled} \vee w.\mathit{retr}.v$ for all $w$ such that $N(w, v)$.

$v$ is not enabled by the assumption $u$ is the highest-priority enabled process, so in order for $v.\mathit{sendtok}.u$ to not hold, $v$ must have relinquished a low token to some other low neighbor, say $w$. $w$ is either enabled or disabled — we show that in either case, eventually $v$ holds all its tokens.

If $w$ is disabled, $w.\mathit{sendtok}.v$ holds and eventually $v$ holds all its tokens. If $w$ is enabled, $v.\mathit{retr}.w$ is true by the assumption $u$ is the highest priority enabled process and $v.\mathit{en}[u] = \mathbf{true}$. So by SF6, eventually $w.\mathit{en}[v] = \mathbf{true}$. Then $w.\mathit{sendtok}.v$ holds and eventually $v$ holds all its tokens.

Now, since $u$ is the highest-priority enabled process, $u.\mathit{ht} > w.\mathit{ht}$ for all $w \neq u$ such that $N(v, w)$ and $v.\mathit{en}[w] = \mathbf{true}$. Then $v.\mathit{sendtok}.u$ will hold and eventually $\mathit{tok}(u, v) = u$. Furthermore, since no process changes enabledness $u.\mathit{en}[v]$ will continue to be false.

Since $v$ was an arbitrary higher-priority process such that $\mathit{tok}(u, v) \neq u$ and $u$ holds its low tokens, eventually $u.\mathit{maylock} \wedge u.\mathit{gate}$ continuously holds. Contradiction.

In both cases we arrive at a contradiction. It follows that if a process is enabled, some process eventually holds a lock.

In the following, we use the metric described in Section 5 as part of the proof of progress.

**Lemma A2** *If $u.ab = \emptyset \wedge u.gate$ and $u$ becomes enabled, eventually $u$ acquires a lock.*

*Proof.* Assume $u$ is a process which becomes enabled and $u.ab = \emptyset \wedge u.gate$. We will show eventually $u.lock$

$u.ab = \emptyset$ implies $u$ has no higher-priority neighbors. Then, by SF0, $u$ is missing at most one low token.

Now, by C3 and C4 there must be a process $v$ such that $N(u, v)$ that released a lock and enabled $u$. Since $u$ is enabled, $u.sendtok.w$ is disabled for all $w$ such that $N(u, w)$.

It follows from SF1 and SF2 that at the point $v$ releases its lock enabling $u$, $tok(u, v) = v$ and $\neg v.lock \wedge \neg v.gate$ hold. By Observation A1, eventually $tok(u, v) = u$ holds. Furthermore, for all $w \neq v$ such that $N(u, w)$, $tok(u, w) = u$. Then $u.maylock$ holds and by SF3 $u.gate$ holds.

Since $u.sendtok.w$ is disabled for all $w$ such that $N(u, w)$, $u.maylock \wedge u.gate$ holds continuously. By weak-fairness, $L_u$ eventually executes and $u.lock$ holds.

Then a process with no higher-priority neighbors that becomes enabled eventually is granted a lock.

**Lemma A3** *If a process is continuously enabled, it eventually is granted a lock.*

*Proof.* By contradiction. Let $u$ be a process such that $u.enabled$ and let $u.M = k$. For a contradiction assume that $u.enabled \wedge \neg u.lock$ hold continuously.

It follows from the assumption that $u.enabled \wedge \neg u.lock$ hold continuously and by a similar argument as in Lemma A2 that $u.ab \neq \emptyset$ continuously. We obtain a contradiction by showing that it must be the case that eventually $u.ab = \emptyset$.

Without loss of generality we may assume (by SF6 and SF7) that $v.en[u] =$ **true** for all neighbors $v$ of $u$ and $u.gate$.

First notice that by SF5, $u.ht$ never changes value. It follows from the definition of $u.M$ that $u.M$ never increases.

In addition, we can assume that $u$ holds all its low tokens: If $u$ is missing low tokens, by SF0, $u$ is missing at most one to a low neighbor, say $v$. As $v.en[u] =$ **true**, $v.sendtok.u$ holds. If $\neg v.gate$ then by Observation A1 eventually $tok(u, v) = u$. If $v.gate$ then eventually either $\neg v.gate$ (if it is the case that $v$ holds all its tokens) or $v$ executes $T_{v,u}$. In either case eventually $tok(u, v) = u$.

In order for $u.enabled \wedge \neg u.lock$ to continuously hold, it must be the case that $u.maylock$ is infinitely often false. Since $u$ holds all its low tokens, infinitely often $u$ does not hold all its high tokens.

Then infinitely often there is some process $v \in u.ab$ such that $v.enabled$. Since $u.ab$ is finite, there must be some $v \in u.ab$ such that $v.enabled$ infinitely often. Fix $v$ to be the highest-priority such process and let $v.ht = j$ and $u.M = k$. We show eventually $u.M < k$.

Since $v$ is the highest-priority process which is infinitely often enabled, eventually all higher-priority processes with respect to $v$ remain disabled. Then, by C4 if $v.enabled$ becomes true, it must be the case that some lower-priority neighbor, say $w$ of $v$ executed its action. By Observation A1 eventually $tok(v, w) = v$ while $v.enabled$ (since no higher-priority neighbor is enabled). Then $v$ holds all its low tokens. Furthermore, by SF6, eventually $w.en[v] = \textbf{true}$ for all higher priority neighbors $w$ of $v$. Again by the assumption no process of higher priority than $v$ is enabled, $w.sendtok.v$ and $w.en[v] = \textbf{false}$ will at some point hold continuously for all such $w$. Then eventually $v$ acquires all its shared tokens and $v.maylock$ holds continuously.

Then, by weak fairness, $v.lock$ eventually holds and by C5 and Observation A1, eventually $v.ht < j$.

It follows that eventually $u.M < k$. Since $v$ was arbitrary as the highest priority process in $u.ab$ such that $v.enabled$ infinitely often, $u.M$ decreases infinitely often.

Since $u.M$ never increases and is bounded below, it must be the case that eventually $u.ab = \emptyset$, which is a contradiction.

**Lemma A4** *If a process is infinitely often enabled, then either eventually it is granted a lock or its metric decreases.*

*Proof.* Assume process $u$ is infinitely often enabled and $u.M = k$. We will show that eventually either $u.lock$ or $u.M < k$.

Since $u$ is infinitely often enabled, we can assume without loss of generality that $u.enabled$ holds.

By A3 $u$ cannot remain enabled indefinitely. Then either eventually $u.lock$ or $\neg u.enabled$. If $u.lock$ then we are done so assume eventually $\neg u.enabled$.

There are two cases.

*Case (a):* Some higher priority neighbor of $u$, say $v$, executed its action and disabled $u$. Then $\neg v.lock \wedge \neg v.gate$. Let $v.ht = j$. By Observation A1, eventually $v.ht < j$. Then, since $u.M$ is the sum of the difference in height between $u$ and all processes above it and $v \in u.ab$, $u.M < k$.

*Case (b):* Some lower priority neighbor of $u$, say $w$, executed its action and disabled $u$. Now, since $u$ is infinitely often enabled, eventually $u.enabled$ will hold. By C4, some process $v$ such that $N(u, v)$ must execute its action, enabling $u$. If $v$ is higher-priority than $u$, by the same argument as in the previous case, eventually $u.M < k$. If $v$ is lower-priority than $u$, no lower-priority neighbor of $u$ can execute once $u.enabled$ holds, since $v$ will return its token and $u$ will not release low tokens while $u.enabled$ holds. Then by A3, either $u.lock$ or $u.disabled$ eventually holds. If $u.lock$ then we are done. Similarly we are done if $u.disabled$, as it must have been a higher-priority process which disabled $u$, which falls under case (a).

In both cases, eventually $u.lock$ or $u.M < k$. Then, if a process that is infinitely often enabled, eventually it is granted a lock or its metric decreases.

**Theorem A1** *(Progress of $SF_u$) If a process is infinitely often enabled, it infinitely often is granted a lock.*

*Proof.* This follows directly from Lemmas A2 and A4.

Assume $u$ is infinitely often enabled. We show $u.lock$ infinitely often.

First we show that if $u.enabled$, eventually $u.lock$.

Assume $u.enabled$ and $u.M = k$. By the assumption and Lemma A4, eventually either $u.lock$ or $u.M < k \wedge u.enabled$. By induction on $k$ and the fact that $u.M$ is bounded below, either $u.lock$ or $u.M = 0$. It follows from Lemma A2 that eventually $u.lock$.

It follows that since $u.enabled$ infinitely often, infinitely often $u.lock$.

## B Proof of Maximality for *SF*

In this section, we provide a full proof of the maximality of $SF$.

### B.1 A Maximal Trace

Let $\sigma$ be a stutter-free sequence of tuples $\sigma = \langle \sigma_0, \sigma_1 \ldots \rangle$ as described in Section 6.

The following properties characterize any trace $\sigma$ that is constructed in the way previously described from a trace satisfying the strong-fairness specification.

These properties are quantified over all processes $u$ and all points $i$ in the trace.

– Every other state is a non-stuttering state, except in the event of quiescence.

$$\sigma_i = \sigma_{i+2} \Rightarrow (\forall j : i \leq j : \sigma_j = \sigma_{j+1}) \tag{T0}$$

$$i \bmod 2 = 0 \Rightarrow \sigma_i = \sigma_{i+1} \tag{T1}$$

– At each point in the computation, $u.count$ increases by at most one for all $u$ and at most one process changes $count$.

$$C_i^u \neq C_{i+1}^u \Rightarrow C_{i+1}^u = C_i^u + 1 \tag{T2}$$

$$C_i^u \neq C_{i+1}^u \Rightarrow (\forall v : u \neq v : C_i^v = C_{i+1}^v) \tag{T3}$$

– If a process changes its count, it must have be enabled.

$$C_i^u \neq C_{i+1}^u \Rightarrow E_i^u \tag{T4}$$

– If a process's enabledness changes, either the process or a neighbor must have increased its count.

$$\begin{aligned} &E_i^u \neq E_{i+1}^u \Rightarrow C_i^u \neq C_{i+1}^u \\ &\vee (\exists v : N(u,v) : C_i^v \neq C_{i+1}^v) \end{aligned}$$

– An infinitely often enabled process infinitely often changes its count.

$$\begin{aligned} &(\forall i :: (\exists j : i < j : E_j)) \Rightarrow \\ &(\forall i :: (\exists j : i < j : C_j^u \neq C_{j+1}^u)) \end{aligned}$$

## B.2 The Constrained Program $SF'$

The constrained program $SF'$ is as it appears in Figure 2 in Section 6.

First, observe that $u.next$ is odd if $u.done$ and even if $\neg u.done$ by the construction of the trace (processes only change count and state in even, non-stuttering states). Then $u.next$ is a function as by definition of $u.done$, $\neg u.done \Rightarrow (\exists i : i \geq p : C_i^u \neq C_{i+1}^u)$. For the other case, by the definition of $\sigma$, $u.done \Rightarrow (\exists i : i \geq p : (\forall j : i < j : \neg E_j^u))$. Also, it is possible to find a unique such $i$: by the construction of the trace, there are an infinite number of stuttering states and a finite number of quiescent processes which assign heights equal to such states.

The following properties follow directly from the text of $SF'$.

- $u.next$ is greater than or equal to the current point in the computation.

$$\textbf{invariant } u.next \geq p \tag{SF$'$0}$$

- If $u.gate$ holds for a process that will change its count again, $u.ht = u.next$.

$$\textbf{invariant } \neg u.done \wedge u.gate \Rightarrow u.ht = u.next \tag{SF$'$1}$$

- If a process will change its count again, $u.next$ is equal to the next point a process changes its count.

$$\textbf{invariant } \neg u.done \Rightarrow C_{u.next+1}^u = C_{u.next}^u + 1 \tag{SF$'$2}$$

- If a process does not change its count again, $u.ht$ is at least the last point $u$ was enabled.

$$\begin{aligned}&\textbf{invariant } u.done \wedge u.gate \Rightarrow u.ht \geq \\ &\qquad (\textbf{Min}\, i :: (\forall j : i \leq j : \neg E_j^u))\end{aligned} \tag{SF$'$3}$$

- The trace is not changed and the current point in the computation increases by at most one.

$$\textbf{constant } \sigma \tag{SF$'$4}$$

$$p = k \textbf{ unless } p = k + 1 \tag{SF$'$5}$$

- It is transient that the current point is a stuttering state and the point is stable in a non-stuttering state.

$$\textbf{transient } \sigma_k = \sigma_{k+1} \wedge p = k \tag{SF$'$6}$$

$$\textbf{stable } \sigma_k \neq \sigma_{k+1} \wedge p = k \tag{SF$'$7}$$

In addition, the following are easily shown:

- If a process increases its count again, it is enabled in the at the point $u.ht$.

$$\textbf{invariant } \neg u.done \wedge u.gate \Rightarrow E_{u.ht}^u \tag{SF$'$8}$$

*Proof.* This follows directly from SF ′ 1, SF ′ 2, and T4.

– *u.done* is stable.

$$\textbf{stable}\ u.done \tag{SF$'$9}$$

*Proof.* If *u.done* holds, it can only be invalidated by $\sigma$ changing or $p$ decreasing. However, by SF ′ 4 and SF ′ 5, the trace is constant and $p$ is nondecreasing.

– *u.quiet* is stable.

$$\textbf{stable}\ u.quiet \tag{SF$'$10}$$

*Proof.* As above.

– If a process will increases its count again, its height is equal to a point in the trace where it increments its count.

$$\neg u.done \Rightarrow C^u_{u.ht} \neq C^u_{u.ht+1} \tag{SF$'$11}$$

*Proof.* The invariant initially holds and since *u.done* is stable, if $\neg u.done$ holds, it held at the point at which $u$ assigned to *u.ht* . This assignment would have assigned to *u.ht* the next point in the computation where $u$ changed its height.

## B.3 The Constrained Client Program

Now we formally state our assumptions about the processes satisfying the client specification. Specifically, we assume that a client process $u$ can be constrained to produce $client'_u$ which satisfies the following:

– The trace is not changed.

$$\textbf{constant}\ \sigma \tag{C$'$0}$$

– The state corresponds with the current point in the trace.

$$\textbf{invariant}\ u.count = C^u_p \tag{C$'$1}$$

$$\textbf{invariant}\ u.enabled = E^u_p \tag{C$'$2}$$

– The client process does not change the point in the computation in a stuttering state.

$$\textbf{stable}\ p = k \wedge \sigma_k = \sigma_{k+1} \tag{C$'$3}$$

– The current point in the computation advances by at most one.

$$p = k\ \textbf{unless}\ p = k + 1 \tag{C$'$4}$$

– Given that no other process changes the point when $C_p^u \neq C_{p+1}^u$, the point is eventually changed.

Hypothesis: **stable** $p = k \wedge C_k^u \neq C_{k+1}^u$
**invariant** $u.enabled = E_p^u \wedge u.count = C_k^u$
**stable** $u.lock$
Conclusion: $p = k \wedge C_k^u \neq C_{k+1}^u \wedge u.lock \leadsto \neg u.lock$ (C′5)

– Guards added to the constrained program are infinitely often true and the truth of an additional guard is preserved by the execution of any action in *client′* and *SF′*.

The following is a property follows from C′1 and C4.

$u.lock \wedge p = k$ **unless** $p \neq k$ (C′6)

Furthermore, we require (as in our constrained *SF′*) that *client′* is produced from a correct client process by only adding new variables, assignments to new variables, and new guards referencing new and program variables. Also, if *client′* replaces random assignments with deterministic assignments referencing new and existing variables, the assigned value satisfies the predicate on the random assignment.

These additional constraints ensure that *client′* satisfies the safety properties in the *client* specification.

## B.4 Proof of Maximality

There are two obligations to prove the maximality of *SF*; we must show (i) an arbitrary trace $\sigma$ satisfying the strong-fairness specification is a possible execution of $SF' \parallel client'$ and (ii) any fair execution of $SF' \parallel client'$ is a fair execution of *SF* composed with a process satisfying the client specification.

**Lemma B1** *At each step in the computation, for each process $u$, $u.count$ and $u.enabled$ are equal to $C_p^u$ and $E_p^u$.*

***invariant*** $u.count = C_p^u \wedge u.enabled = E_p^u$

*Proof.* Initially the invariant holds: **initially** $p = 0$, **initially** $u.enabled = E_0^u$, and by C′1 **initially** $u.count = C_0^u$.

By SF′4, the only way the invariant can be invalidated in *SF′* is if $p$, *u.enabled*, or *u.count* is assigned to. However, no action in *SF′* assigns to $p$, *u.count*, or *u.enabled*, so *SF′* maintains the invariant.

*client′* maintains the invariant by C′1 and C′2.

Then the invariant holds in the composed system.

Next, we introduce a few obligations which we will use in the proofs of the remaining properties.

**Observation B1** ***invariant*** $u.lock \Rightarrow \quad (u.ht = p \wedge \neg u.done)$

*Proof.* The invariant holds initially as no process holds a lock. The only action of $SF'$ which assigns to $u.lock$ is $L'_u$ . It follows from the guards of $L'_u$ that if the action is enabled, $u.ht = p \wedge \neg u.done$ .

To show the invariant holds in the composed program, it suffices to show the predicate is stable in $client'$ . Since releasing a lock maintains the invariant and by C3 and $\neg u.lock$ is stable in a correct client process, the only way for $client'$ to invalidate $u.lock \Rightarrow (u.ht = p \wedge \neg u.done)$ is by changing $p$ while $u.lock$ holds (as both $u.ht$ and the trace are constant in $client'$ ). However, by C$'$1, C$'$4, and C4, in order to change the point, the client must release a lock. Then the predicate is stable in $client'$ and it is an invariant of the composed program.

**Observation B2** ***invariant*** $(p = k \wedge C^u_k = C^u_{k+1}) \Rightarrow \quad \neg u.lock$

*Proof.* Assume $p = k$ and $C^u_k = C^u_{k+1}$ . For a contradiction assume $u.lock$ . By the Observation B1, $u.ht = p \wedge \neg u.done$ . It follows from $\neg u.done$ and SF$'$11 that $C^u_{u.ht} \neq C^u_{u.ht+1}$ . However, $u.ht = p = k$ and $C^u_k = C^u_{k+1}$ . Contradiction.

**Observation B3** $(\neg u.done \wedge u.gate \wedge u.ht = p) \Rightarrow (\forall v : N(u,v) \wedge v.ht < u.ht \wedge v.gate : v.quiet\ )$ .

*Proof.* By contradiction. Assume $\neg u.done \wedge u.ht = p$ and $v$ is a neighbor of $u$ such that $v.gate$ , $v.ht < u.ht$ , and $\neg v.quiet$ . There are two cases, $\neg v.done$ or $v.done$ .

In the case of $\neg v.done$ , by SF$'$1 and SF$'$0, $v.ht \geq p$ . So $v.ht \geq u.ht$ , which is a contradiction.

In the case of $v.done$ and $\neg v.quiet$ , there exists some point in the computation $i \geq p$ such that $E^v_i$ but no point $j \geq p$ such that $C^v_i \neq C^v_{i+1}$ . By SF$'$3, $v.ht \geq i$ . Then $v.ht \geq p$ and $v.ht \geq u.ht$ , which is a contradiction.

In either case we have a contradiction.

**Lemma B2** *The current point in the computation eventually changes.*

$$p = k \leadsto p \neq k$$

*Proof.* Assume $p = k$ . It follows from T1, T0, and T3 that either $\sigma_k = \sigma_{k+1}$ or there exists a process $u$ such that $C^u_k \neq C^u_{k+1}$ . Then there are two cases.

*Case (a):* $p = k \wedge \sigma_k = \sigma_{k+1}$ . We show $\sigma_k = \sigma_{k+1} \wedge p = k$ **ensures** $p \neq k$ in the composed system. By C$'$3, **stable** $\sigma_k = \sigma_{k+1} \wedge p = k$ in $client'$ . It suffices to show $\sigma_k = \sigma_{k+1} \wedge p = k$ **ensures** $p \neq k$ in $SF'$ .

By SF$'$6, $\sigma_k = \sigma_{k+1} \wedge p = k$ is transient in $SF'$ . Since $\sigma$ is constant, the only way for the predicate to be transient is for $p$ to change. The ensures-property follows from this.

*Case (b):* $C^u_k \neq C^u_{k+1} \wedge p = k$ . C $'$ 5 and C $'$ 6 entails $client'$ will eventually change the count provided $p = k \wedge C^u_k \neq C^u_{k+1} \wedge u.lock$ . Then, in order to show eventually $p \neq k$ it suffices to show that the hypothesis of C $'$ 5 is satisfied by the rest of the processes in the union and eventually $u$ holds a lock.

The hypothesis is satisfied by Lemma B1, S1, SF $'$ 7, C $'$ 3, and T3.

Assume $p = k$ and $C^u_k \neq C^u_{k+1}$ . By the definition of *done* and *next* it is the case that $\neg u.done$ and $u.next = p$ . Without loss of generality we may assume $u.gate$ holds as if $\neg u.lock \wedge \neg u.gate$ , eventually $u.gate$ and if $u.lock \wedge \neg u.gate$ , we are done.

Then by SF $'$ 1, SF $'$ 8, and Lemma B1, $u.ht = p$ and $u.enabled$ .

Furthermore, by T3, it is the case for all $v \neq u$ that $C^v_k = C^v_{k+1}$ . Then, by C $'$ 3, no other process may change the point in the computation. It then follows from Lemma B1 that $u.enabled$ is stable unless the point is incremented by process $u$ .

Now, for all $v \neq u$ , it is the case that $\neg v.lock$ by Observation B2 and $C^v_k = C^v_{k+1}$ . We may then assume without loss of generality $v.gate$ . Since $u.ht = p$ , by Observation B3, $u$ is the highest-priority among all processes that may ever become enabled.

Also notice that for all $v \neq u$ the additional guard of action $L'_v$ is false since heights are unique and $u.ht = p$ .

Then eventually $u$ will collect its shared tokens and $u.lock$ will hold.

**Lemma B3** *The current point in the computation eventually advances by one.*

$$p = k \leadsto p = k + 1$$

*Proof.* This follows directly from Lemma B3 and $p = k$ **unless** $p = k + 1$ is a property of the composed system.

The following observation is used in the proofs of the remaining Lemmas.

**Observation B4** ***invariant*** $(\neg u.lock \wedge \neg u.gate) \Rightarrow \quad u.ht < p$

*Proof.* Assume $\neg u.lock \wedge \neg u.gate$ . By SF4, in order to have $\neg u.gate$ , it must have been the case that action $L'_u$ was executed in a state where its guard was true, granting $u$ a lock. Let this point in the computation be $p'$ . Then $u.ht = p'$ at the point $L'_u$ was executed in a state where its guard was true. Since $u.lock$ is stable in $SF'$ (S1), it must have been falsified by an action of $client'$ . By C $'$ 6, $client'$ releasing a lock necessarily changes the point, so by C $'$ 4 $p' < p$ . Furthermore, since $u.ht = k$ is stable in $client'$ and $u.ht = k \wedge \neg u.gate$ is stable in $SF$ , it must be the case $u.ht = p'$ if $\neg u.gate$ . Then $u.ht = p' < p$ .

**Lemma B4** *Any guard added to an existing action in the composed system can only be falsified by the action it guards.*

*Proof.* The requirements on constraining a client process to create $client'$ stipulate that this property holds.

It remains to be show that the Lemma holds for additional guards of $SF'$ . It holds trivially for actions $U'_{u,v}$ , $T'_{u,v}$ and $D'_u$ . Then it suffices to show $(u.ht = p \ \wedge \ \neg u.done) \ \vee \ u.quiet$ is preserved by all actions of $client' \parallel SF'$ .

Again, by the requirements on $client'$ , we only need to consider actions of $SF'$ . Since $u.quiet$ is stable and the trace constant in $SF'$ , the only actions which can falsify the predicate are $D'_u$ , which assigns to $u.ht$ , and $Q'$ , which assigns to $p$ .

To show $D'_u$ cannot falsify the guard of $L'_u$ , assume $u.ht = p$ and $\neg u.done$ . We will show $u.lock \ \vee \ u.gate$ by contradiction. So assume $\neg u.lock \ \wedge \ \neg u.gate$ . By B4, $u.ht < p$ . Contradiction.

To show $Q'$ cannot falsify $L'_u$ , assume $u.ht = p \ \wedge \ \neg u.done$ . By SF $'$ 9, at the point $u.ht$ was assigned to, $\neg u.done$ held. Then by the definition of $u.next$ , $C^u_{u.ht} \neq C^u_{u.ht+1}$ . So $C^u_p \neq C^u_{p+1}$ and therefore $\sigma_p \neq \sigma_{p+1}$ . Then $Q'$ is disabled in a state where $u.ht = p \ \wedge \ \neg u.done$ holds.

It follows that $(u.ht = p \ \wedge \ \neg u.done) \ \vee \ u.quiet$ can be falsified only by executing $L'_u$ .

**Lemma B5** *Any guard added to an existing action in the composed system is infinitely often true.*

*Proof.* The Lemma trivially holds for actions $U'_{u,v}$ , $T'_{u,v}$ , and $D'_u$ and holds for all actions in $client'$ by the requirements of $client'$ . Then it suffices to show $(u.ht = p \ \wedge \ \neg u.done) \ \vee \ u.quiet$ is infinitely often true.

If $u.quiet$ ever becomes true, it will remain true forever by SF $'$ 10. We show that if $\neg u.quiet$ is an invariant of the trace, $u.ht = p \ \wedge \ \neg u.done$ infinitely often. From the definition of $u.quiet$ and $u.done$ , if $\neg u.quiet$ is an invariant of the trace, then $\neg u.done$ is an invariant of the trace as well.

Since the only action which assigns to height is $D'_u$ , $u.ht = k$ unless $\neg u.done \ \wedge \ \neg u.gate$ . By B4, $(\neg u.done \ \wedge \ \neg u.gate) \ \Rightarrow \ u.ht < p$ . So $u.ht = k$ **unless** $u.ht < p$ .

To prove the Lemma, we will show that if $\neg u.done$ is an invariant of the trace, then $u.ht = p$ infinitely often. There are two cases.

*Case (a):* $u.gate$ holds. Then by SF $'$ 3, $u.ht = u.next$ and by SF $'$ 0, $u.ht \geq p$ . If $u.ht = p$ then we are done so assume $u.ht = k$ where $k > p$ . By the unless property demonstrated in the penultimate paragraph and induction on $p$ , eventually $u.ht = p$ .

*Case (b):* $\neg u.gate$ holds. If $\neg u.gate$ then as in the proof of B4, $\neg u.lock \ \wedge \ \neg u.gate$ hold. It follows eventually $u.gate$ , which is case (a).

In either case, eventually $u.ht = p$ . Then $u.ht = p$ infinitely often and the guard of $L'_u$ is infinitely often true.

**Lemma B6** *The assignment* $u.ht := u.next$ *in* $D'_u$ *implements the random assignment in* $D^u$ .

*Proof.* In order for $D'_u$ to be enabled, it must be the case $\neg u.lock \wedge \neg u.gate$. By B4, $u.ht < p$. By SF$'$0, $u.next \geq p$. So $u.ht$ is increased. Now we show $u.ht$ is unique. There are two cases. In the first case, $u.done$ at the point of the assignment. By the definition of $u.next$ if $u.done$ holds, then $u.next$ is odd and $u.next \neq v.ht$ for all $v$. If $\neg u.done$ holds, then $u.next$ is even and equal to the next point $u$ increments count, which is unique by T3.

Then $u.ht := u.next$ implements the random assignment in $D^u$.

Finally, we prove the main results — $\sigma$ is a possible computation of $SF' \parallel client'$ and $SF$ maximally implements the *scheduler* specification.

**Theorem B1** *Any fair execution of $SF' \parallel client'$ is a fair execution of $SF$ composed with the client process.*

*Proof.* Lemmas B4 and B5 establish that each action of $SF' \parallel client'$ is infinitely often executed in a state where its additional guard is true. Then weakly-fair execution of the composed constrained system corresponds to a weakly-fair execution of the original composed system.

Lemma B6 establishes that $u.ht := u.next$ correctly implements the random assignment in $SF$.

**Theorem B2** *The $SF$ algorithm composed with a client process meeting the additional requirements is a maximal with respect to the strong fairness specification.*

*Proof.* Let $\sigma$ be an arbitrary trace satisfying the strong-fairness specification. We show $\sigma$ is a possible execution of $SF$ composed with a client process that is able to satisfy the requirements stated.

Lemmas B1 and B3 establish $\sigma$ as a fair execution of $SF' \parallel client'$. Theorem B1 proves that any execution of $SF' \parallel client'$ is a fair execution of the unconstrained system. Then $\sigma$ is a possible execution of the unconstrained system.

Since $\sigma$ is arbitrary, any trace satisfying the strong-fairness specification is a possible execution of $SF$ composed with a client process meeting the requirements stated. Then the $SF$ algorithm is a maximal solution to the strong-fairness problem.