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 continuously enabled actions are 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 while enabled infinitely often. 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.

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 [4] 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

A. Pelc (Ed.): DISC 2007, LNCS 4731, pp. 358–372, 2007.

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.

2 Maximality and Fairness

2.1 Maximality

A program is maximal if it is capable of generating any behavior permitted by a specification [5,12]. This notion is similar to bisimulation [11,13]. 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. Firstly, 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*). Since σ is a possible execution of P' and every execution of P' corresponds to a possible execution of P, σ is a possible execution of P. Hence, any trace in 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 σ and auxiliary variables (e.g., variables that encode the current point in the computation).

Proving *chronicle correspondence*, requires showing that the execution of P' follows a given trace σ. Proving *execution correspondence* requires showing that (i) each added guard in P' is infinitely often true and (ii) the truth of each added 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. Thus, every weakly-fair execution of P' corresponds to a weakly-fair execution of P.

2.2 Fairness

Consider the following UNITY [2] program:

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

Action A is always enabled. It increments x and sets b to $\neg b$. Action B is enabled in states where b is true and assigns to y the value of x.

Weak fairness requires that every action be selected infinitely often. Under this assumption, the *fairness* program satisfies: the safety property (i) x increases by at most one in each step, and the progress properties (ii) x eventually increases and (iii) b eventually changes value. More formally: (i) $x = k$ **next** $|x-k| \leq 1$, (ii) $x = k \leadsto x \neq k$, and (iii) $b = k \leadsto b \neq k$. 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 involving y is one of safety: at each step of the computation, y either remains the same or changes to the value of x. Since action B may never be selected while enabled, no progress properties for y can be proven. For example, consider the sequence of actions: $\langle A, B, A, A, B, A, A, B, \ldots \rangle$. This schedule is weakly fair since all actions are selected infinitely often, but B never executes from an enabled state and so y never changes value.

Strong fairness, on the other hand, requires that any action that is infinitely often enabled be selected while enabled infinitely often. Under strong fairness, the *fairness* program satisfies the same properties as it did under weak fairness. In addition, the program also satisfies new properties, including the progress property y increases eventually

2.3 Maximality and Scheduling

Assertions should be as strong as possible and must hold in every possible program execution. 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 an action to be disabled at most twice before being 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, for example: $x - y \leq 4$ 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.

3 Specification

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. Clients can be *enabled*, and an enabled client can be granted a *lock*. Holding a lock allows the client to access some resource, perform some action(s), or otherwise modify the system state, including enabling or disabling other clients. When a client modifies system state, it simultaneously increments its own *count* and releases the lock it holds.

The scheduler layer manages locks. If a process is infinitely often enabled, the scheduler ensures that it is infinitely often granted a lock. We say two processes u and v are neighbors if u or v's client 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.

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

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 u or v can affect the other's enabledness.[1]. Each process $u \in \mathcal{P}$ has boolean variables $u.enabled$ and $u.lock$ representing that process being enabled and holding a lock, respectively. A third variable, $u.count$, is 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.

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. The specification for client u is:

[1] This neighbor relation is irreflexive, it is never the case $N(u, u)$. This is not to say that a process cannot enable/disable itself by executing its action; this is captured in the specification. The irreflexitivity of N only simplifies presentation.

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

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

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

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

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

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

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

Properties (C0)–(C2) ensure that clients can modify only the enabledness of neighbors. Property (C3) ensures that a lock is necessary for a client to act. Propery (C4) ensures that count is incremented and enabledness of neighbors affected only with the release of a lock. Property (C5) is a conditional property; if the scheduling layer maintains the properties that neighbors do not hold locks simultaneously and that only enabled clients hold locks, 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.

3.4 Scheduler Layer Specification

This 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.

$$\textbf{constant}\ u.count \tag{S0}$$

$$\textbf{stable}\ u.lock \tag{S1}$$

$$\textbf{invariant}\ u.lock \Rightarrow u.enabled \tag{S2}$$

$$\textbf{invariant}\ (\forall\, v \,:\, N(u,v) \,:\, \neg(u.lock \wedge v.lock)\ ) \tag{S3}$$

Hypothesis: $\textbf{true} \leadsto u.enabled$, C0, C2, C3, C4, C5,

Conclusion: $\textbf{true} \leadsto u.lock$ (S4)

Properties (S0) and (S1) ensure that the scheduling layer does not modify the count, nor revoke a lock once granted. Property (S2) ensures that locks are granted only to enabled processes, while property (S3) ensures that neighbors do not hold locks simultaneously. Property (S4) is a conditional property that captures the notion of strong fairness. If a correct client process is infinitely often enabled, the scheduler infinitely often grants 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 a process is infinitely often enabled, it infinitely often increases its execution count.

Formally, *client* $\|$ *scheduler* satisfies:

Hypothesis: $\mathbf{true} \rightsquigarrow u.enabled$

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

4 Algorithm for Scheduling Layer

Solving the strong fairness scheduling problem entails providing an algorithm that satisfies the specification of the scheduling layer from the previous section. In addition, our goal is for this algorithm to be maximal with respect to the composed specification.

The challenge in designing a strongly fair scheduler lies in limiting concurrency—no correct scheduler can *always* allow 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. Suppose y and z are enabled while x is disabled. A scheduler that always allows processes sharing a mutual neighbor to concurrently hold locks permits both y and z to acquire locks. Now suppose y executes its action, leaving x and y both enabled. Since z still holds its lock and $N(x, z)$, x may not acquire a lock. Now suppose z executes its action, disabling x but leaving z enabled. The system is now in back in the state where y and z are enabled while x is disabled. A scheduler that 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 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 v notifies a neighbor u of its enabledness by assigning to $u.en[v]$.

To ensure progress, each process u is has a height, $u.ht$, representing its priority. 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.

In addition, process u includes a boolean variable $u.gate$. 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 $u.en[w] = \mathbf{true}$.

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

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

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

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

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

Figure 1 shows this implementation of u's *scheduler* layer. Actions $U_{u,v}$ and $T_{u,v}$ are understood to be quantified across all neighbors v of 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
$U_{u,v}$ **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 $u.maylock \wedge u.gate \longrightarrow u.lock :=$ **true**;
$u.gate :=$ **false**
D_u $\neg u.lock \wedge \neg u.gate \longrightarrow u.gate :=$ **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

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 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 information can be encoded on shared tokens as differences in height, and by storing locally the height of the (unique) low neighbor holding a token.

5 Correctness of SF_u

Properties (S0), (S1), and (S2) follow directly from the program text. Property (S3) is satisfied since a process must hold all its shared tokens to grant itself a lock and a process does not relinquish its tokens while it holds a lock.

The progress property (S4) (that an infinitely often enabled process holds a lock infinitely often) requires a more thorough treatment. In the interest of space, however, we only sketch the key proof ideas here. The complete proof is available in [9].

In order to prove (S4), we show: (i) the system is free from deadlock, (ii) a process with no higher priority neighbors that 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.

Part (i) follows from the acyclicity of the partial order of priorities. The remaining parts rely on the identification of a metric. We define $u.M$ to be 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.

6 The Maximality of *SF*

Since maximality is noncompositional, we use the rely-guarantee style proof outlined in [10] 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 [9] 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.

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 σ satisfying the strong-fairness specification. We then show that at each point i in the trace σ, the state of the system is exactly that of σ_i. This establishes σ 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 σ 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 require that this client process can be constrained to produce $client'$ which, when composed with SF', can take the "steps" in the computation that σ dictates. *i.e.,* if at some point i in σ some process u is to execute and enable/disable itself or its neighbors, $client'$ can compute this step. The additional requirements are that the client process is maximal, $client'$ satisfies the safety properties of the $client$ specification, and that $client'$ is created in a way that ensures the correspondence between executions of $client'$ and the client process.

In order to compute σ, we introduce a variable p shared by $client'$ and SF' that marks the current point in the trace (i.e., σ_p). We then prove (i) it is invariant that the current state is σ_p and (ii) the point p eventually increases. It follows that σ is a possible execution of $SF' \parallel client'$.

6.2 A Strong Fairness Trace

Let σ 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 σ_i. That is, $E_i^u = \mathbf{true}$ if $u.enabled$ in σ_i and $C_i^u = k$ if $u.count = k$ in σ_i. σ is stutter-free in that each tuple in the sequence differs from the previous by at least one element, *unless* the execution is in a state of quiescence (each processes is disabled forever).

Since σ is a correct trace of the strong fairness scheduling problem, it obeys certain properties. Namely, it satisfies the following: in subsequent states in σ, 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 σ, we create an isomorphic trace σ' by inserting a stuttering-state in between every σ_i and σ_{i+1}. That is, $\sigma'_0 = \sigma_0$ and σ'_{i+1} is σ'_i if i is even and is $\sigma_{(i+1)/2}$ if i is odd.

6.3 Requirements of $client'_u$

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

- $client'_u$ 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 $client'_u$ satisfies the safety properties of the client process.
- The additional guards of $client'_u$ 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 $u.enabled = E^u_p$ and $u.count = C^u_p$.
- $client'_u$ does not assign to σ 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 $C^u_k \neq C^u_{k+1}$ (*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 σ. It is then the obligation of SF' to ensure that processes hold locks when σ 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 σ 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.

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

$$u.done \equiv (\forall i : i \geq p : C^u_i = C^u_{i+1})$$

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

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 σ is to ensure that a process that never executes again can be assigned a unique height. If σ 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
$U'_{u,v}$ **true** $\longrightarrow$
true $\longrightarrow$ $v.en[u] := u.enabled \;\vee\; u.retr.v$
$T'_{u,v}$ **true** $\longrightarrow$
$u.sendtok.v \;\wedge\; u.gate \;\longrightarrow\; tok(u,v) := v$
L'_u $(u.ht = p \;\wedge\; \neg u.done) \;\vee\; u.quiet \;\longrightarrow$
$u.maylock \;\wedge\; u.gate \;\longrightarrow\; u.lock :=$ **true**;
$u.gate :=$ **false**
D'_u **true** $\longrightarrow$
$\neg u.lock \;\wedge\; \neg u.gate \;\longrightarrow\; u.gate :=$ **true**;
$u.ht := u.next$;
$(\,\|\; v \,:\, N(u,v) \;\wedge\; u.ht < v.ht \,:\, tok(u,v) := v\,)$
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 σ.

6.5 Proof Sketch of the Maximality of *SF*

There are two main obligations to dispatch: (i) $SF' \parallel client'$ computes σ 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 σ 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 [8]. 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 [7], Karaata gives a distributed self-stabilizing algorithm for the strongly-fair scheduling of atomic actions under weak fairness. A key property of the algorithm is that an action u can disable another action v at most twice before action v must execute, and therefore this algorithm is not maximal. In addition, although there is no notion of a "lock," the algorithm precludes two processes with a shared neighbor from both having the "right" to execute their actions. Although this does not affect the possible schedules the algorithm can generate, it does limit 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. 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 [6], 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.

The dining philosophers problem proposed by Dijkstra [3] is superficially similar (as also pointed out in [7]) 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. Apt, K.R., Francez, N., Katz, S.: Appraising fairness in distributed languages. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 189–198. ACM Press, New York (1987)
2. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Reading, Massachusetts (1988)
3. Dijkstra, E.W.: Hierarchical ordering of sequential processes. Acta Informatica 1(2), 115–138 (1971)
4. Joshi, R., Misra, J.: Maximally concurrent programs. Formal Aspects of Computing 12(2), 100–119 (2000)
5. Joshi, R., Misra, J.: Toward a theory of maximally concurrent programs. In: Proceedings of PODC '00, pp. 319–328 (2000)
6. Joung, Y.-J.: On fairness notions in distributed systems, part I: A characterization of implementability. Information and Computation 166, 1–34 (2001)
7. Karaata, M.H.: Self-stabilizing strong fairness under weak fairness. IEEE Trans. Parallel Distrib. Syst. 12(4), 337–345 (2001)
8. Lamport, L.: Fairness and hyperfairness. Distrib. Comput. 13(4), 239–245 (2000)
9. Lang, M., Sivilotti, P.A.G.: A distributed maximal scheduler for strong fairness. Technical Report OSU-CISRC-7/07-TR61, The Ohio State University (July 2007)
10. Lang, M., Sivilotti, P.A.G.: The maximality of unhygienic dining philosophers. Technical Report OSU-CISRC-5/07-TR39, The Ohio State University (May 2007)
11. Milner, R.: Communication and concurrency. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
12. Misra, J.: A Discipline of Multiprogramming: Programming Theory for Distributed Applications. Springer, New York (2001)
13. Park, D.: Concurrency and automata on infinite sequences. In: Proceedings of the 5th GI-Conference on Theoretical Computer Science, London, UK, pp. 167–183. Springer, Heidelberg (1981)