On the Impossibility of Maximal Scheduling for Strong Fairness with Interleaving

Matthew Lang and Paolo A. G. Sivilotti
Computer Science & Engineering
The Ohio State University
Columbus, Ohio 43210
{*langma,paolo*}*@cse.ohio-state.edu*

Abstract

A strongly fair schedule is one in which tasks that are enabled infinitely often are also executed infinitely often. When tasks execute atomically, a strongly fair scheduler can be implemented in a maximal manner. That is, an algorithm exists that, for any valid schedule, is capable of generating that schedule. We show that this assumption of atomicity is necessary. That is, when task execution can be interleaved with other tasks, no algorithm is capable of generating all valid schedules. In other words, any algorithm that correctly generates some strongly fair schedules must also be incapable *of generating some other valid schedules.*

This impossibility result is the first example of an implementable UNITY specification for which no maximal solution exists.

1. Introduction

A maximal implementation is one that is capable of exhibiting *all* behaviors permitted by its specification [1]. In cases where a specification permits nondeterminism (as is common in distributed systems), maximality is a strictly stronger requirement than correctness. For example, consider a specification for a communication protocol that requires packet numbers to be assigned in increasing order, but permits any value to be used for the first packet number. A correct implementation of this protocol could always use 0 for the first packet number. But this implementation would not be maximal, since it would never exhibit the behavior where the sequence of packet numbers begins with a value different from 0. Nondeterminism is useful in specifications since it affords greater flexibility to implementations.

In some situations, maximal implementations are preferable to ones that are merely correct. For example, network fingerprinting is a security attack that exploits the non-maximality of implementations for common network protocols. Operating systems differ in how they implement TCP/IP, a nondeterministic specification. While each implementation is correct, none is maximal. Thus, an attacker can infer a remote machine's operating system vendor, version, and often patch level, simply by observing that machine's behavior with respect to TCP/IP [2], [3].

Maximality is also important for modular testing of component-based systems. In particular, a non-maximal implementation of one component can mask errors in a different component, and thus prevent the test-based detection of those errors. For example, consider a multithreaded system with a component responsible for scheduling which thread executes next. Testing such nondeterministic systems is notoriously difficult since an error in the multithreaded code might be exposed only by particular schedules. It is a well-known observation that testing can be used only to reveal the presence of errors, never their absence [4]. A non-maximal scheduler, however, makes things even worse. If the (correct but non-maximal) scheduler is incapable of producing any of the problematic schedules, then *no amount* of testing can be used to reveal the presence of this error.

In this paper, we examine the problem of implementing a strongly fair scheduler in a maximal manner. The context of this scheduling problem is a general one: tasks are modelled as actions that are either enabled or disabled. An enabled action, when executed, modifies the state of the system and thus can affect whether other actions are enabled or disabled. A strongly fair scheduler must guarantee that an action that is infinitely often enabled is infinitely often executed.

Informally, a tension exists when implementing a strongly fair scheduler in a maximal manner. On one hand, establishing the correctness of a scheduling algorithm is easiest when the algorithm permits little (or no) concurrency amongst tasks. On the other hand, a maximal algorithm must permit as much concurrency as possible.

A maximal algorithm for strongly fair scheduling does exist [5]. This algorithm, however, assumes that tasks execute atomically. That is, the algorithm is maximal only for systems in which a task affects the enabledness of other tasks in one atomic step. This assumption is not reasonable for distributed systems, where the effects of one task can be interleaved with the effects of another. We show, however, that this assumption is necessary. That is, we show that when task execution is interleaved, there does not exist a maximal implementation of a strongly fair scheduler. Correct implementations do exist, but any such implementation is necessarily non-maximal. This result has theoretical importance in that it establishes a new impossibility result regarding maximality. It has practical consequence as well; any system

scheduled using such a scheduler becomes unnecessarily deterministic.

The impossibility of a maximal distributed strongly fair scheduler is the first result of its kind: an implementable UNITY specification that has no maximal solution. In the next section, we illustrate two categories of specifications which do not have maximal implementations. We will also show that maximal implementability is non-monotonic with respect to refinement.

We then formalize the scheduling problem for strong fairness with interleaving, and prove that any solution to this problem is necessarily non-maximal.

2. Specifications without Maximal Solutions

Here, we present illustrative examples of two categories of specifications that are not maximally implementable. We use UNITY logic [6] for specifications and assume a model of computation based on the UNITY computational model. The results in this section, however, extend to any reasonable model of computation for parallel systems (such as the one described in [7]). The appendix provides a formal treatment of our model, including definitions of algorithms, executions, traces, and when a trace satisfies a UNITY specification.

2.1. The Fair Dice Problem

The *Fair Dice Problem* is as follows: A process observes dice being thrown. The process must eventually declare whether or not the dice are fair. The process is allowed to claim that the dice are fair when they may not be (*i.e.*, the process is allowed a false positive), but must not claim that they are unfair when in fact they are (*i.e.*, the process is not allowed a false negative).

The formal specification for the *observer* process with a single die valuing 0 or 1 is:

constant die	(O1)
invariant $decision \in \{undecided, fair, unfair\}$	(O2)
true $\leadsto$ $decision \neq undecided$	(O3)
stable $decision = k$ for $k \in \{fair, unfair\}$	(O4)
Hypothesis: $die = k \leadsto die \neq k$, **constant** $decision$	
Conclusion: **true** $\leadsto$ $decision = fair$	(O5)

Informally, this specification says that the *observer* is not allowed to change the die (O1) (*i.e.*, only the environment can roll the die), the *observer* must eventually decide on $fair$ or $unfair$ (O2, O3), and once a decision is made it cannot be changed (O4). The last property requires the *observer* to decide $fair$ if the die changes infinitely often.

The family of behaviors defined by the *observer* specification fall in two categories:

- Those where the value of die changes infinitely often, and hence eventually $decision = fair$.
- Those where the value of die changes a finite number of times, and hence eventually either $decision = fair$ or $decision = unfair$.

In order for an *observer* process to be maximal, it must be capable of generating any such behavior. However, there is no such observer process—every correct observer *must always declare the die to be fair*.

Theorem 2.1. *There is no maximal implementation of the fair dice problem.*

Proof: For a contradiction assume there is a maximal solution to the fair die problem and let $\mathcal{O}$ be a maximal observer. Now consider a sequence δ of die rolls such that die changes value only a finite number of times. In order for $\mathcal{O}$ to be maximal, there must be a execution of $\mathcal{O}$ in which $\mathcal{O}$ decides $unfair$ after observing a finite prefix of δ. Let n be the number of rolls that are observed.

Now let δ' be a sequence of die rolls identical to δ up to the $n + 1^{th}$ roll and for every subsequent roll, the value of the die is the opposite of the previous roll. Since $\mathcal{O}$ observes the same n rolls in δ' as in δ, the execution of $\mathcal{O}$ that decides $unfair$ after observing n rolls of δ must also decide $unfair$ after observing n rolls of δ'. However, δ' clearly meets the hypothesis of O5 and therefore since $\mathcal{O}$ is correct, it must decide $fair$. This is a contradiction; $\mathcal{O}$ cannot both decide $fair$ and $unfair$. □

Intuitively, the fair dice problem does not have a maximal solution because it requires a process to make an oracular decision about the future. Though it is not surprising that an observer process cannot tell the future, there are *correct* implementations of an observer process (namely, those that always eventually decide fair).

Note that if the specification is weakened by removing property O4, the resulting specification has a maximal solution. Furthermore, if the specification is strengthened by replacing O2 with **invariant** $decision \in \{undecided, fair\}$, the resulting specification has a maximal solution. This implies that maximal implementability is non-monotonic in the refinement hierarchy. It is possible that a specification becomes not maximally implementable by strengthening or weakening it (or, as in this case, either).

The fair dice problem is representative of specifications for which a subset of correct behaviors are unimplementable (*e.g.,* predicting the future). A similar problem would be requiring a process querying an eventually perfect ($\Diamond\mathcal{P}$) failure detector [7] to either make no claim about the completeness and accuracy of its detector or to correctly decide that its detector has entered its stable suffix. A maximal process must be able to decide that it has reached the stable suffix, which is not possible—such a process could be used to implement a perfect ($\mathcal{P}$) failure detector.

2.2. The Asynchronous Alarm Problem

The *Asynchronous Alarm Problem* is as follows: An alarm process is monitoring a door. The alarm process queries the door and receives a reply indicating whether or not the door was closed or open when the door processed the query. If door is ever open, the alarm process must raise an alarm. The alarm process may issue a false alarm (*i.e.*, it may raise an alarm if the door is never opened), but it must not fail to raise an alarm if the door is ever opened.

Formally, the specification for the *alarm* process is:

$$\textbf{constant } door \tag{A1}$$

$$\textbf{initially } \neg alarm \tag{A2}$$

$$\textbf{stable } alarm \tag{A3}$$

$$door = open \leadsto alarm \tag{A4}$$

The family of behaviors defined by the $alarm$ specification are those where

- Eventually $alarm$ is assigned **true**, or
- $alarm$ remains **false** and $door = closed$ holds continuously.

The $alarm$ specification is not maximally implementable; every correct $alarm$ process must always eventually assign **true** to $alarm$.

Theorem 2.2. *There is no maximal solution to the asynchronous alarm problem.*

Proof: To see why, assume there is a maximal implementation of the $alarm$ specification, say $\mathcal{A}$. Let ϵ be an execution of $\mathcal{A}$ where $door = closed$ holds continuously and $alarm$ is never assigned **true**.

Let ϵ' be an execution of $\mathcal{A}$ in which the door is initially open and subsequently closed before the first query by $\mathcal{A}$. The sequence of responses that $\mathcal{A}$ receives is then identical to that in ϵ. Since $\mathcal{A}$ never assigns **true** to $alarm$ in ϵ, it must also never assign **true** to $alarm$ in ϵ' given the same sequence of responses. However, since $\mathcal{A}$ is correct, by A4 $alarm$ must eventually be assigned **true**. This is a contradiction. □

The asynchronous alarm problem has no maximal solution because it requires an alarm process to be able to obtain a complete history of the state of the door. Due to the inherent asynchrony in the system and the fact that the door's response to a query only returns the current state of the door, it is impossible for the alarm process to build this history. In other words, a maximal alarm process must make a local decision based events that is never guaranteed to observe.

So, while the fair dice problem had no maximal solution because an observer process could not tell the future, the asynchronous alarm problem has no maximal solution because it cannot have a complete picture of the past.

It is important to again note that this problem has maximal solutions if the specification is either weakened or strengthened. The weaker specification comprised of A1, A2, and A3 has a maximal solution, as does the stronger specification in which A4 is replaced by

Hypothesis: **stable** $door = open$

Conclusion: $door = open \leadsto alarm$

The asynchronous alarm problem is characteristic of specifications where there are correct behaviors that are isomorphic [8] to incorrect behaviors. As we will show, the distributed strongly fair scheduling problem is a member of this category.

3. The Distributed Strong Fairness Scheduling Problem

The distributed strong fairness scheduling problem is a distributed resource allocation problem that models scenarios where a set of processes have conflicts over shared resources and a process's desire to access a shared resource depends on other processes' usage of the resource.

The problem was originally presented [9] in the context of scheduling actions in a distributed system. A distributed system is typically modelled as a set of actions, each of which is either enabled or disabled. An execution of such a system is a sequence of actions; a fairness criterion controls what sequences of actions are valid executions of a system. One of two notions of fairness is typically assumed: weak fairness or strong fairness. Weak fairness dictates that each action is infinitely often selected for execution, where strong fairness dictates that if an action is infinitely often enabled, it is infinitely often selected for execution.

Weak fairness is useful because of its minimal assumption—any individual action that is continually enabled is eventually selected for execution—and the ease of generating weakly fair schedules. Strong fairness is useful for simplifying algorithms since it ensures that actions that are repeatedly enabled are not starved. While weakly-fair scheduling is easy because of the independent nature of action selection, strongly-fair scheduling requires synchronization and coordination and therefore is non-trivial to implement.

Solutions to the strongly fairness scheduling problem can be used to generate schedules of atomic actions that satisfy the strong fairness criterion. However, the utility of the problem extends beyond the scheduling of atomic actions; solutions may be used to mediate access to shared resources (be it shared memory, radio links in a sensor network, etc.) in a strongly-fair fashion.

In the following, we formally define the distributed strong fairness scheduling problem.

3.1. Description of the System

The system is a set of processes, each of which is comprised of two components: a *client* and a *scheduler*. The scheduler designer is given the specification of both components and must design a refinement of the scheduler specification.

Each process has an associated *task* that is either *enabled* or *disabled*. When the task is enabled, the client may be granted a *lock*. The client, upon being granted a lock, executes the task. When the task completes the client releases the lock.

While a task is executing, it may either enable or disable the tasks belonging to other processes. This modification of the states of other tasks is subject to the constraint that a task may change the state of another task exactly once during its execution[1].

Two processes u and v are called *neighbors* if the execution of u's task may affect the enabledness of v's task or vice-versa. If the scheduler guarantees that no two neighboring processes are holding locks concurrently, tasks are guaranteed to eventually complete and the client must eventually release a lock.

The scheduling layer is responsible for granting locks to processes subject to the strong fairness criteria: if a task is infinitely often enabled, its corresponding client is infinitely often granted a lock. The system therefore generates strongly-fair schedules—if a task is infinitely often enabled it is infinitely often executed while it is enabled.

3.2. Formal Specification of the Distributed Strong Fairness Problem

The system is comprised of a set of processes Π with a symmetric *neighbor relation* $N \subseteq \mathcal{P}(\Pi)$. For two processes u and v, $(u, v) \in N$ if the execution of u or v's task may affect the enabledness of the other's. N is irreflexive; it is not the case that for any u, $(u, u) \in N$.

Each process $u \in \Pi$ has the associated local variables:

- $u.enabled$, a boolean representing the enabledness of u's task. If $u.enabled$ is **true** then u's task is enabled.
- $u.lock$, a boolean representing whether u holds a lock. If $u.lock$ is **true** then u may execute its task.

3.2.1. Client Specification. In our formalization of the problem, we will not formalize the notion of a task. We will subsume the effect that tasks have on the enabledness of processes in the specification of the client component.

In the formal specification of the client component, $u.\mathit{affect}_v$ is a specification variable indicating whether or not u's task has affected the enabledness of v's task. In the following, let v range over processes and a and b range over $\mathbb{B}$.

$$(\forall v : v \neq u : \textbf{constant } v.lock) \quad \text{(C1)}$$

$$(\forall v : \neg N(u, v) \wedge v \neq u : \textbf{constant } v.enabled) \quad \text{(C2)}$$

$$\textbf{stable } \neg u.lock \quad \text{(C3)}$$

$$\textbf{invariant } (\exists v :: u.\mathit{affect}_v \Rightarrow u.lock) \quad \text{(C4)}$$

$$(\forall v, b :: v.enabled = b \textbf{ unless } u.\mathit{affect}_v) \quad \text{(C5)}$$

$$(\forall v, b :: v.enabled = b \wedge u.\mathit{affect}_v \textbf{ next } v.enabled = b) \quad \text{(C6)}$$

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

Conclusion: $u.lock \rightsquigarrow \neg u.lock$ (C7)

The first two properties of the client component (C1 and C2) specify that a client cannot modify the lock variable of any other process and that it may only affect the enabledness of its neighbors. C3 restricts a client from granting itself a lock. C4 requires that the specification variable $u.\mathit{affect}_v$ is only **true** when $u.lock$ holds. C5 requires that u may only change the enabledness of a neighbor v when $u.\mathit{affect}_v$ becomes **true**. This property, along with C4 and C6 (which specifies that u may not change the enabledness of another process while $u.\mathit{affect}_v$ is **true**) allows u to change the enabledness of a neighbor at most once while $u.lock$ holds.

The progress property C7 states that u eventually releases a lock, provided: $u.lock$ is not falsified by any other process, u holds a lock only if $u.enabled$ holds, and u never holds a lock concurrently with a neighbor.

3.3. Scheduler Specification

The formal specification of the scheduler process is as follows, where v ranges over processes:

$$(\forall v :: \textbf{constant } v.enabled) \quad \text{(S1)}$$

$$(\forall v : v \neq u : \textbf{constant } v.lock) \quad \text{(S2)}$$

$$\textbf{stable } u.lock \quad \text{(S3)}$$

$$\textbf{invariant } u.lock \Rightarrow u.enabled \quad \text{(S4)}$$

$$\textbf{invariant } (\forall v : N(u, v) : \neg(u.lock \wedge v.lock)) \quad \text{(S5)}$$

Hypothesis: **true** $\rightsquigarrow u.enabled$, C1–C7
Conclusion: **true** $\rightsquigarrow u.lock$ (S6)

Properties S1 and S2 specify that the scheduler for u may not modify $u.enabled$ nor $v.lock$ for any other process v. S3 ensures that a scheduler may not revoke a lock once it has granted it and S4 ensures that a process only grants a

1. The problem where tasks are allowed to change the state of other tasks more than once has no solution.

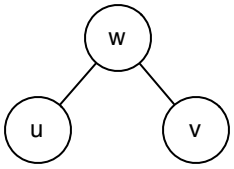

Figure 1. Example Instance of Strong Fairness Problem

lock if its client is enabled. S5 guarantees that a scheduler will not grant a lock if a neighboring process holds a lock.

The progress property S6 guarantees that a process is infinitely often granted a lock if it is infinitely often enabled and the client process is correct.

3.4. Composed Specification

Given a correct implementation $client$ of the client specification and a correct implementation $scheduler$ of the scheduler specification, the composed system satisfies the strong fairness property: if a task is infinitely often enabled, its client is infinitely often granted a lock (and therefore infinitely often executes its task). Formally, $client \parallel scheduler$ satisfies:

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

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

Conclusion: **true** $\leadsto u.lock$

4. The Impossibility of a Maximal Distributed Scheduler for Strong Fairness

The specification of the distributed strong fairness scheduler is implementable; there are schedulers that are capable of generating strongly-fair schedules. However, there is no maximal implementation of the scheduler specification. A correct scheduler must necessarily be incapable of generating *all* strongly-fair schedules.

Informally, the reason there does not exist a maximal strongly-fair scheduler is that a correct scheduler must necessarily limit the concurrency of locks being held by two non-neighboring processes with a shared neighbor. That is, in any schedule generated by a correct scheduler, there is a point in the schedule after which two processes with a shared neighbor never concurrently hold locks, even though they are not neighbors and therefore may be permitted to.

For example, in the system depicted by Figure 1 (where nodes represent processes and edges represent the neighbor relation), there is a point in every schedule generated by a correct scheduler after which $u.lock$ and $v.lock$ never hold concurrently. *i.e.*, after a certain point, the execution of tasks by u and v becomes serialized.

Consider the schedule s of the system in Figure 1 in which process w is never enabled and processes u and v are continually enabled and repeatedly are granted locks concurrently. This schedule is correct; both u and v are both infinitely often enabled and infinitely often hold locks. However, if a scheduler were to be capable of generating this schedule, it *must also be capable of generating incorrect schedules where w is infinitely often enabled and is never granted a lock.*

To see why, consider the schedule s' where w is never granted a lock and, as in s, u and w are both continually enabled and are repeatedly granted locks concurrently. However, in s' each time u is granted a lock it enables w and each time v is granted a lock it disables w. This schedule is clearly incorrect; w is infinitely often enabled but is never granted a lock. Now suppose a scheduler is capable of generating s. The same scheduler generates s' if the enabling and disabling of w occur without an action of the scheduler being executed in the interim. As the scheduler never observes w being enabled, the execution of the scheduler that produced s must also produce s'.

We begin a more formal treatment with the statement of the main theorem:

Theorem 4.1. *There is no maximal implementation of the scheduler specification.*

Proof: This follows directly from the following Lemma. □

Lemma 4.1. *Assume $\mathcal{S}$ is a correct implementation of the scheduler specification. There does not exist an implementation $\mathcal{C}$ of the client specification such that $\mathcal{S} \parallel \mathcal{C}$ is maximal with respect to the distributed strong fairness specification.*

Proof: By contradiction. Suppose $\mathcal{S}$ is a correct implementation of the scheduler specification and for a contradiction assume there does exist a client $\mathcal{C}$ such that $\mathcal{S} \parallel \mathcal{C}$ is a maximal solution to the distributed strong fairness specification.

Let $\Pi = \{u, v, w\}$ and $N = \{(u,w),(v,w)\}$ (*i.e.*, the system portrayed in Figure 1). Let γ be a trace of Π where w never is enabled and never holds a lock and u and v are continually enabled and are repeatedly granted locks. Figure 2 is a graphical representation of γ; the boxes depict the state of each process's $lock$ variable in each step of γ, where a black box indicates the process holds a lock in that state and a white box indicates the process does not hold a lock.

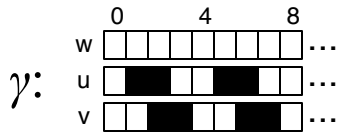

Figure 2. Trace of System

Formally, let γ be a trace where:

$$
\begin{aligned}
&(\forall p :: \neg\gamma_0(p.lock)\ )\\
&(\forall p : p \in \{u,v\} : (\forall i :: \gamma_i(p.enabled)\ )\ )\\
&(\forall i :: \neg\gamma_i(w.enabled) \wedge \neg\gamma_i(w.lock)\ )\\
&(\forall i : (i-1) \bmod 4 = 0 \vee (i-2) \bmod 4 = 0 : \gamma_i(u.lock)\ )\\
&(\forall i : (i-2) \bmod 4 = 0 \vee (i-3) \bmod 4 = 0 : \gamma_i(v.lock)\ )\\
&(\forall i : (i-1) \bmod 4 \neq 0 \vee (i-2) \bmod 4 \neq 0 : \neg\gamma_i(u.lock)\ )\\
&(\forall i : (i-2) \bmod 4 \neq 0 \vee (i-3) \bmod 4 \neq 0 : \neg\gamma_i(v.lock)\ )
\end{aligned}
$$

Let $\epsilon = \langle \sigma_0\alpha_0\sigma_1\alpha_1\sigma_2\ldots\rangle$ be an execution[2] of $\mathcal{S} \parallel \mathcal{C}$ producing γ (since, by assumption, $\mathcal{S} \parallel \mathcal{C}$ is maximal and γ is clearly a correct trace, there exists such an ϵ).

Let $\mathcal{C}'$ be the following program:

Program $\mathcal{C}'$
assign

$$
\begin{aligned}
\beta_p \quad & p.lock \wedge \neg p.set \longrightarrow\\
& \quad (\parallel q : N(p,q) : q.enabled := \neg q.enabled\ )\\
& \quad \parallel p.set := \textbf{true}\\
\delta_p \quad & p.lock \wedge p.set \longrightarrow\\
& \quad p.set, p.lock := \textbf{false}\ ,\ \textbf{false}
\end{aligned}
$$

To arrive at a contradiction, we will use ϵ to construct an execution ϵ' of $\mathcal{S} \parallel \mathcal{C}'$ that violates the strong-fairness progress property by repeatedly enabling and disabling w and never granting w a lock.

Let ϵ' be the sequence $\epsilon' = \langle \sigma'_0\alpha'_0\sigma'_1\alpha'_1\sigma'_2\ldots\rangle$. Define σ'_i for all i by:

$$
\begin{aligned}
&(\forall x : x \in V_{\mathcal{S}} :\\
&\quad (\forall i : i \bmod 3 = 0 : \sigma'_i(x) = \sigma_{i/3}(x)\\
&(\forall x : x \in V_{\mathcal{S}} \setminus \{w.enabled\} :\\
&\quad (\forall i : i-1 \bmod 3 = 0 : \sigma'_i(x) = \sigma_{(i-1)/3}(x)\\
&(\forall i : i-1 \bmod 3 = 0 : \sigma_{(i-1)/3}(u.lock) \wedge\\
&\quad \neg\sigma_{(i+2)/3}(b.lock) \Leftrightarrow \sigma'_i(w.enabled)\\
&(\forall x : x \in V_{\mathcal{S}} : (\forall i : i-2 \bmod 3 = 0 :\\
&\quad \sigma'_i(x) = \sigma_{(i-2)/3}(x)
\end{aligned}
$$

And define α'_i for all i by:

- if $i \bmod 3 = 0$ then:

  β_u if $\sigma'_{i/3}(u.lock) \wedge \neg\sigma'_{(i/3)+1}(u.lock)$
  β_w otherwise

- if $i-1 \bmod 3 = 0$ then:

  β_v if $\sigma'_{(i-1)/3}(u.lock) \wedge \neg\sigma'_{(i-1)/3+1}(u.lock)$
  δ_w otherwise

- if $i-2 \bmod 3 = 0$ then:

  $\alpha_{(i-2)/3}$ if $\alpha_{(i-2)/3} \in \mathcal{S}$
  δ_u if $\sigma'_{(i-2)/3}(u.lock) \wedge \neg\sigma'_{(i-2)/3+1}(u.lock)$
  δ_v if $\sigma'_{(i-2)/3}(v.lock) \wedge \neg\sigma'_{(i-2)/3+1}(v.lock)$
  δ_w otherwise

Informally, ϵ' is constructed by replacing every action in ϵ with three actions. If an action α_i in ϵ was an action of $\mathcal{S}$, α_i is replaced by $\beta_u\delta_u\alpha_i$ in ϵ'. If the action α_i in ϵ is an action of $\mathcal{C}$ then it is replaced by $\beta_u\beta_v\delta_u$ if the action corresponded to u or w releasing a lock or $\beta_w\delta_w\delta_v$ if it did not. The construction of σ' guarantees that each action of $\mathcal{S}$ occurs in the same state as it did in ϵ and for each α'_i in ϵ', $(\sigma'_i, \sigma'_{i+1}) \in \alpha'_i$.

By the construction of ϵ', each action of $\mathcal{S}$ and $\mathcal{C}'$ appear infinitely often. The proof that for all i, $(\sigma'_i, \sigma'_{i+1}) \in \alpha'_i$ follows from the construction of ϵ' and the definition of programs given in the appendix. It follows from these two properties that ϵ' is a valid execution of $\mathcal{S} \parallel \mathcal{C}'$.

By replacing actions of $\mathcal{C}$ with $\beta_u\beta_v\delta_u$ when u released a lock in ϵ, each time u released a lock in ϵ corresponds to a point in ϵ' where $w.enabled$ is assigned **true** and subsequently assigned **false**.

Because u infinitely often holds a lock in ϵ, w is infinitely often enabled in ϵ'. However, $w.lock$ never holds in ϵ', violating the strong-fairness progress property (S6). Since ϵ' is a valid execution of $\mathcal{S} \parallel \mathcal{C}'$, this contradicts the assumption that $\mathcal{S}$ is correct. □

2. See the appendix for a formal definition of executions, traces, and their relationship.

4.1. Discussion

The problem of scheduling actions in a strongly-fair manner under a weak fairness assumption was proposed along with a non-maximal algorithm in [9]; a formalization of the specification and a maximal solution was presented in [5]. This, on the surface, seems to be at odds with our result. However, the specification in [5] is strictly stronger than the specification presented here—it requires tasks to be atomic *and* be executed in a non-interleaved manner. This sacrifice in generality allows the specification to have a maximal solution. In fact, this highlights the observation that maximal implementability may be achieved by strengthening a specification.

The weakened specification presented here captures a more diverse set of resource allocation scenarios, including those where concurrency is a desired property. Furthermore, this specification makes a minimal set of requirements on a client process: client processes do not need to maintain any state nor do they need to have knowledge of a task's characteristics, they simply need to execute a task and release the lock when the task has completed.

Maximal general solutions to weakly-fair scheduling are common [10], [11], [12]. The impossibility result presented here means there are no such maximal algorithms for strong fairness. Any maximal strongly-fair scheduler must be with respect to a specialization of the problem.

5. Conclusions

This work establishes an impossibility result for maximal strongly fair scheduling. This result is the first of its kind for UNITY specifications and programs. We also note that the maximal-implementability of specifications is non-monotonic with respect to refinement. Both of these observations lead to an important question: what is a complete and precise characterization of specifications for which no maximal solution exists?

In this work, we gave two examples of categories of problems that have no maximal solution: those for which a subset of behaviors are impossible to implement (telling the future) and those for which there are correct behaviors that are isomorphic to incorrect behaviors. A goal of continuing work is to explore a precise and complete characterization of UNITY specifications for which no maximal solution exists.

References

[1] R. Joshi and J. Misra, "Toward a theory of maximally concurrent programs," in *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM Press, 2000, pp. 319–328.

[2] G. Lyon, *Nmap Network Scanning*. Nmap Project, 2009.

[3] M. Smart, G. R. Malan, and F. Jahanian, "Defeating TCP/IP stack fingerprinting," in *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2000, pp. 17–17.

[4] E. W. Dijkstra, "The humble programmer," *Communications of the ACM*, vol. 15, no. 10, pp. 859–866, 1972.

[5] M. Lang and P. A. G. Sivilotti, "A distributed maximal scheduler for strong fairness," in *DISC*, ser. Lecture Notes in Computer Science, A. Pelc, Ed., vol. 4731. Springer, 2007, pp. 358–372.

[6] K. M. Chandy, *Parallel program design: a foundation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1988.

[7] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[8] K. M. Chandy and J. Misra, "How processes learn," in *PODC '85: Proceedings of the fourth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1985, pp. 204–214.

[9] M. H. Karaata and P. Chaudhuri, "A self-stabilizing algorithm for strong fairness," *Computing*, vol. 60, no. 3, pp. 217–228, 1998.

[10] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, no. 2, pp. 115–138, 1971.

[11] K. M. Chandy and J. Misra, "The drinking philosophers problem," *ACM Trans. Program. Lang. Syst.*, vol. 6, no. 4, pp. 632–646, 1984.

[12] M. Lang and P. A. G. Sivilotti, "The maximality of unhygienic dining philosophers," The Ohio State University, Tech. Rep. OSU-CISRC-5/07-TR39, 5, 2007.

Appendix

We formally give the definition of a *program*, a *trace*, an *execution* of a program, and what it means for a trace to satisfy a set of UNITY properties.

A program $\mathcal{P}$ is a collection of guarded actions. A state σ is a function from program variables to values. Each action $\alpha \in \mathcal{P}$ is a relation on states; *i.e.*, if action α is selected for execution in a state σ and can result in the state σ', then $(\sigma, \sigma') \in \alpha$.

Actions are *dependent* on the program variables mentioned in the action. Let V_α be the set of variables mentioned by α. Then the following properties hold:

$$(\,\forall\, \sigma, \sigma', v \,:\, (\sigma, \sigma') \in \alpha \wedge v \notin V_\alpha \,:\, \sigma(v) = \sigma'(v)\,)$$

$$(\,\forall\, \sigma, \sigma', \sigma'', \sigma''' \,::\, (\sigma, \sigma') \in \alpha \wedge (\,\forall\, v \,:\, v \in V_\alpha \,:\, \sigma(v) = \sigma''(v) \wedge \sigma'(v) = \sigma'''(v)\,) \Rightarrow (\sigma'', \sigma''') \in \alpha\,)$$

Let $V_{\mathcal{P}}$ denote the set of variables mentioned by $\mathcal{P}$; $V_{\mathcal{P}} = \bigcup_{\alpha \in \mathcal{P}} V_\alpha$.

A trace τ is an infinite sequence of states $\tau = \langle \sigma_0, \sigma_1, \sigma_2, \ldots \rangle$. We say τ is *stutter-free* if $(\forall i :: \sigma_i \neq \sigma_{i+1} \vee (\forall j : j \geq i : \sigma_j = \sigma_{j+1}))$. We say a trace τ' *reduces to* another trace τ if τ can be produced by removing stuttering states from τ' (*i.e.*, replacing σ_i, σ_{i+1} with σ_i, σ_{i+2} if $\sigma_i = \sigma_{i+1}$).

An execution ϵ of a program $\mathcal{P}$ is an infinite sequence of the form $\epsilon = \langle \sigma_0, \alpha_0, \sigma_1, \alpha_1, \sigma_2, \ldots \rangle$, where each σ_i is a state, each α_i is an action of $\mathcal{P}$, and for all i, $(\sigma_i, \sigma_{i+1}) \in \alpha_i$. An execution ϵ is *fair* if each action of $\mathcal{P}$ appears in ϵ infinitely often.

The *projection* of a trace σ over a set of program variables V is the trace σ' where for all i and $v \in V$, $\sigma_i(v) = \sigma'_i(v)$ and for all $v \notin V$, $\sigma'_i(v)$ is undefined.

An execution $\epsilon = \langle \sigma_0, \alpha_0, \sigma_1, \alpha_1, \sigma_2, \ldots \rangle$ *produces* a trace $\sigma = \langle \sigma'_0, \sigma'_1, \sigma'_2, \ldots \rangle$ if (a) for all i, $\sigma_i = \sigma'_i$, (b) there exists a σ' such that ϵ produces σ' and σ' reduces to σ, or (c) ϵ produces a σ' and there exists a set of variables V such that σ is the projection of σ' over V.

The value of a predicate P in a state σ is denoted as $P(\sigma)$.

We now define when a trace σ satisfies UNITY properties. In the following, let P and Q be predicates, var be a program variable, ν be a value, and let $A_0, \ldots, A_n$ and $B_0, \ldots, B_m$ be sets of UNITY properties:

$\sigma \vdash$ **initially** P if $P(\sigma_0)$

$\sigma \vdash P$ **next** Q if $(\forall i :: P(\sigma_i) \Rightarrow Q(\sigma_{i+1}))$

$\sigma \vdash$ **constant** var if
$(\forall \nu :: \sigma \vdash (var = \nu)$ **next** $(var = \nu)$ and
$\sigma \vdash (var \neq \nu)$ **next** $(var \neq \nu))$

$\sigma \vdash$ **transient** P if $(\forall i :: (\exists j : j \geq i : \neg P(\sigma)))$

$\sigma \vdash P \leadsto Q$ if
$(\forall i :: P(\sigma_i) \Rightarrow (\exists j : j \geq i : Q(\sigma_j)))$

$\sigma \vdash A_0, \ldots, A_n$ if $\sigma \vdash A_0$ and $\ldots$ and $\sigma \vdash A_n$

$\sigma \vdash$ Hypothesis: $A_0, \ldots, A_n$ Conclusion: $B_0, \ldots, B_m$ if
$\sigma \vdash A_0$ and $\ldots$ and $\sigma \vdash A_n$
$\Rightarrow \sigma \vdash B_0$ and $\ldots$ and $\sigma \vdash B_n$

$\sigma \vdash A$ if the projection of σ over the variables mentioned by A, σ', is such that $\sigma' \vdash A$

It can be shown that for any program $\mathcal{P}$ proved correct with respect to a UNITY specification $A_0, \ldots, A_n$ the following holds: for all traces τ of $\mathcal{P}$, $\tau \vdash A_0, \ldots, A_n$ [3].

3. It should be noted that the converse is *not* true; proving a program satisfies a property **transient** P requires that the program have a single action that when executed in a state satisfying P results in a state satisfying $\neg P$.