# How to reason with Strong-fairness and No-fairness

Notes on UNITY: 31–92

Jayadev Misra*
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
(512) 471-9547
misra@cs.utexas.edu

7/2/92

## 1 The Problem

Some feel that the UNITY-logic is deficient because it embeds a fixed notion of fairness, namely that each statement be executed infinitely often in any execution. Other notions of fairness, such as every statement that is enabled infinitely often is executed infinitely often (known as "strong fairness"), or, as long as there is an enabled statement some (enabled) statement is executed (known as "minimal progress condition," or "no-fairness" in this note) might be more appropriate for certain problems. In fact, [3] and [4] allow different forms of fairness to be specified for different statements of a program.

I believe that the weakest notion of fairness that is adequate to establish "absence of individual starvation" is the only useful type of fairness. Strong-fairness can be added as an axiom if desired. The advantage of embedding a particularly weak form of fairness into the logic is that the appropriate logical operators and the corresponding inference rules can be postulated once for all, and these can be exploited to reduce proof lengths. As an example, we prove a program from [4] that assumes strong-fairness for some statements, using UNITY-logic. Later, we show that the proof is still valid if the weak-fairness of UNITY is replaced by no-fairness (and the same strong-fairness assumption is retained).

**Program** (reproduced from [4], p. 3)

```
var       integer x, y = 0 ;
  semaphore     sem = 1 ;
cobegin loop    α1 : ⟨P(sem)⟩ ;
                β1 : ⟨x := x + 1⟩ ;
                γ1 : ⟨V(sem)⟩
        endloop
   ||
        loop
                α2 : ⟨P(sem)⟩
                β2 : ⟨y := y + 1⟩
                γ2 : ⟨V(sem)⟩
coend   endloop
```

*This material is based in part upon work supported by the Texas Advanced Research Program under Grant No. 003658–219, by the Office of Naval Research Contract N00014-90-J-1640 and by the National Science Foundation Award CCR–9111912.

It is required to show that assuming strong-fairness for the action at $\alpha_1$—i.e., if $sem > 0$ holds infinitely often then the action at $\alpha_1$ executes infinitely often—$x$ grows arbitrarily large.

**Notes**

1. Lamport assumes strong-fairness for actions at $\alpha_1$ and $\alpha_2$. Strong-fairness of the latter action is unnecessary for this particular proof.
2. Initial values of $x, y$ are irrelevant for the proof.
3. Weak-fairness for the other statements, I believe, is assumed by Lamport in constructing his proof. This is unnecessary; we return to this issue in Section 4.

# 2 The Mathematical Model

I will model this program—this program is a mathematical object, no doubt, but it is poorly suited for rigorous analysis—by a UNITY program. In the following, $pc_1, pc_2$ denote the program counters of process 1 and process 2. I have replaced *sem* by $s$ (since shorter identifiers are preferable for formal manipulations by hand).

**var** $pc_1, pc_2$ : $(\alpha, \beta, \gamma)$ ;
    $s, x, y$ : integer
**initially** $s, x, y, pc_1, pc_2 = 1, 0, 0, \alpha, \alpha$
**assign**
    {code for process 1}
    $s, pc_1 := s-1, \beta$ if $s > 0 \wedge pc_1 = \alpha$
  ▯ $x, pc_1 := x+1, \gamma$ if $pc_1 = \beta$
  ▯ $s, pc_1 := s+1, \alpha$ if $pc_1 = \gamma$
  ▯ {code for process 2}
    $s, pc_2 := s-1, \beta$ if $s > 0 \wedge pc_2 = \alpha$
  ▯ $y, pc_2 := y+1, \gamma$ if $pc_2 = \beta$
  ▯ $s, pc_2 := s+1, \alpha$ if $pc_2 = \gamma$
**end**

The strong-fairness constraint for the first action in the first process is modelled by the axiom

(SF)
$$\frac{true \;\mapsto\; s > 0 \;\wedge\; pc_1 = \alpha}{true \;\mapsto\; pc_1 = \beta}$$

Note that $true \mapsto p$ is one way of representing that $p$ holds infinitely often.

# 3 The Proof

We will establish

$(P)$
$$(\forall\, n \;::\; x = n \;\mapsto\; x = n+1)$$

Note: A better representation of "$x$ grows arbitrarily large" is

$$(\forall\, n \;::\; true \;\mapsto\; x > n)$$

This property follows from (P) by applying the induction rule of $\mapsto$ .

**Proof Outline**

1. $true \ \mapsto \ s > 0 \ \wedge \ pc_1 = \alpha$ , see below
   $true \ \mapsto \ pc_1 = \beta$ , using SF
   $x = n \ unless \ x = n + 1$ , code inspection
   $x = n \ \mapsto \ (pc_1 = \beta \ \wedge \ x = n) \ \vee \ x = n + 1$ , PSP on the above two
   $pc_1 = \beta \ \wedge \ x = n \ \mapsto \ x = n + 1$ , code inspection for *ensures*
   $x = n \ \mapsto \ x = n + 1$ , cancellation on the above two

Proof of (1): $true \ \mapsto \ s > 0 \ \wedge \ pc_1 = \alpha$

By code inspection prove the following invariant.

$$0 \leq s \leq 1 \ \wedge \ (pc_1 = \alpha \ \vee \ pc_2 = \alpha) \ \wedge \ [s = 1 \ \equiv \ (pc_1 = \alpha \ \wedge \ pc_2 = \alpha)]$$

Now,

$s = 0$
$\Rightarrow$ {from the invariant}
$pc_1 \neq \alpha \ \vee \ pc_2 \neq \alpha$
$\Rightarrow$ $\{pc_1 \ \in \ \{\alpha, \beta, \gamma\} \ , \ pc_2 \ \in \ \{\alpha, \beta, \gamma\} \ \}$
$pc_1 = \beta \ \vee \ pc_1 = \gamma \ \vee \ pc_2 = \beta \ \vee \ pc_2 = \gamma$
$\mapsto$ $\{pc_1 = \beta \ \mapsto \ pc_1 = \gamma$, using code inspection for *ensures*. Similarly, for $pc_2$. Apply cancellation$\}$
$pc_1 = \gamma \ \vee \ pc_2 = \gamma$
$\Rightarrow$ $\{pc_1 = \gamma \ \Rightarrow \ s = 0$ , from the invariant. Similarly, for $pc_2$.$\}$
$(pc_1 = \gamma \ \wedge \ s = 0) \ \vee \ (pc_2 = \gamma \ \wedge \ s = 0)$
$\mapsto$ $\{pc_1 = \gamma \ \wedge \ s = 0 \ \mapsto \ s > 0$ , using code inspection for *ensures*. Similarly, for $pc_2$. Apply cancellation$\}$
$s > 0$

Hence,

The given invariant
$\Rightarrow$ $s = 0 \ \vee \ s = 1$
$\mapsto$ $s > 0 \ \vee \ s > 0$
$\equiv$ $s > 0$

Applying the substitution axiom,

$true \ \mapsto \ s > 0$ □

**Properties requiring code inspection** The following properties have to be proven by code inspection.

2. $x = n \ unless \ x = n + 1$
3. $pc_1 = \beta \ \wedge \ x = n \ ensures \ x = n + 1$
4. invariant $0 \leq s \leq 1 \ \wedge \ (pc_1 = \alpha \ \vee \ pc_2 = \alpha) \ \wedge \ [s = 1 \ \equiv \ (pc_1 = \alpha \ \wedge \ pc_2 = \alpha)]$
5. $pc_1 = \beta \ ensures \ pc_1 = \gamma$
6. $pc_1 = \gamma \ \wedge \ s = 0 \ ensures \ s > 0$

Properties (3,5) can be replaced by

(7) $$pc_1 = \beta \ \wedge \ x = n \ \textit{ensures} \ pc_1 = \gamma \ \wedge \ x = n + 1$$

Note that (3) follows from (7) by weakening the latter's rhs to $x = n + 1$. Property (5) does not follow from (7); we, however, need only

$$pc_1 = \beta \ \mapsto \ pc_1 = \gamma$$

in the proof. This follows from (7) by

$$\begin{array}{lll} pc_1 = \beta \wedge x = n \ \textit{ensures} \ pc_1 = \gamma & & \text{, weakening the rhs of (7)} \\ pc_1 = \beta \wedge x = n \quad \mapsto \quad pc_1 = \gamma & & \text{, definition of } \mapsto \\ pc_1 = \beta \quad \mapsto \quad pc_1 = \gamma & & \text{, disjunction over all } n \end{array}$$ □

**Compositional Proof** The proof can be shortened further by employing the union theorem, in particular to prove the propeties (2,6,7). Since $pc_1, x$ are local to process 1, it suffices to prove (2,7) in process 1 only. For property 6, from the invariant (4) we have,

$$s = 0 \ \textit{unless} \ s > 0 \qquad \text{in process 2}$$

Then,

$$pc_1 = \beta \ \wedge \ x = 0 \ \textit{unless} \ s > 0 \quad \text{in process 2 , locality of } pc_1.$$

Therefore, it suffices—using the union theorem—to prove property (6) in process 1 alone. Property (4), the invariant, does require a proof over both processes; it has to follow from the initial conditions and be shown *stable* over both processes.

# 4 No-fairness

The following treatment is inspired by [2]. The progress property, (P), holds even when no fairness is assumed for any statement of the program besides the strong fairness property, (SF). Fortunately, exactly the same proof applies; only, the *ensures* properties have different definitions and they have to be reverified. It is extremely lucky that all inference rules of UNITY—except the union theorem—continue to hold for the new *ensures*. Hence, the proof in Section 3 continues to hold as long as the *ensures* properties are not proven compositionally. (As a matter of fact, there is a weaker union theorem for the new *ensures* that is adequate for this particular problem.)

For a loop of the form

$$\text{do } [i \ : \ 0 \leq i < N \ : \ g_i \ \mapsto \ A_i]$$

where $g_i$ is the guard of command $A_i$, define

$$p \ \textit{ensures}' \ q \ \equiv \quad p \ \wedge \ \neg q \ \Rightarrow \langle \exists \ i \ :: \ g_i \rangle \ \wedge \ \langle \forall \ i \ :: \ \{p \ \wedge \ \neg q \ \wedge \ g_i\} \ A_i \ \{q\} \rangle$$

Thus, $p$ *ensures*$'$ $q$ says that (1) some statement is enabled whenever $p \ \wedge \ \neg q$ holds, and (2) execution of any enabled statement in a state satisfying $p \ \wedge \ \neg q$ establishes $q$. It is easy to show that the properties of *ensures* given in Sec. 3.6.2 of [1] hold for *ensures*$'$, as well. (*Unless* will be defined as before.) Hence, other derived rules, such as PSP, continue to hold with *ensures*$'$.

We have the following weaker union theorem with *ensures′*.

$$\frac{p \textit{ ensures}' q \quad \text{in } F \ , \ p \textit{ ensures}' q \quad \text{in } F \parallel G}{p \textit{ ensures}' q \quad \text{in } F \parallel G}$$

Note that $p$ *ensures′* $q$ in $F$ and $p$ *unless* $q$ in $G$ does *not* establish $p$ *ensures′* $q$ in $F \parallel G$. In particular, $G$ could contain the single statement

$$\textit{true} \ \mapsto \ \textit{skip}$$

which could be executed forever.

Also, the converse of this rule does not hold; $p$ *ensures′* $q$ may hold in $F \parallel G$ though it holds in neither $F$ nor $G$. To see this, consider

$$\begin{array}{rcl} F & :: & x = 0 \ \mapsto \ x \ := \ 2 \qquad \{\text{if } x \neq 0 \text{ this statement execution is } \textit{skip}\} \\ G & :: & x = 1 \ \mapsto \ x \ := \ 2 \end{array}$$

$0 \leq x$ *ensures* $x > 1$ holds in $F \parallel G$ , though in neither $F$ nor $G$.

Finally, a disjunction rule, similar to the disjunction rule for $\mapsto$, holds for *ensures′* but not *ensures*.

# References


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

[2] Charanjit S. Jutla and Josyula R. Rao. On the Design of Proof Rules for Fair Parallel Programs. *Formal Aspects of Computing* (submitted).

[3] Simon S. Lam and A. Udaya Shankar. Refinement and Projection of Relational Specifications. Proc. REX Workshop on Stepwise Refinement of Distributed Systems, Plasmolen-Mook, The Netherlands, May 1989, LNCS series, Springer-Verlag.

[4] Leslie Lamport. A Temporal Logic of Actions. DEC, Systems Research Center, Palo Alto, CA, April 1990.