Equivalence Is In The Eye Of The Beholder

Yuri Gurevich*and James K. Huggins*

Abstract

This is a reaction to Lamport's "Processes are in the Eye of the Beholder." To illustrate the "insubstantiality of processes," Lamport represents a 2-process algorithm and an N-process algorithm by temporal formulas and proves the equivalence of the two formulas. We analyze in what sense the two algorithms are and are not equivalent, and give a more direct equivalence proof.

1 Introduction

In "Processes are in the Eye of the Beholder" [7], Leslie Lamport writes:

> A concurrent algorithm is traditionally represented as the composition of processes. We show by an example that processes are an artifact of how an algorithm is represented. The difference between a two-process representation and a four-process representation of the same algorithm is no more fundamental than the difference between $2 + 2$ and $1 + 1 + 1 + 1$.

To demonstrate his thesis, Lamport uses two different programs for a first-in, first-out ring buffer of size N, both written in a CSP-like language [4]. The first, shown in Figure 1, operates the buffer using 2-processes; the second, shown in Figure 2, uses N-processes. We call these two programs $\mathcal{R}_{csp}$ and $\mathcal{C}_{csp}$, respectively (for reasons which shall become apparent). After presenting the programs, Lamport represents them as formulas in TLA, the Temporal Logic of Actions [6], and proves the equivalence of the two formulas in TLA.

There are three issues where we disagree with Lamport.

Issue 1: The Notion of Equivalence. What does it mean that two programs are equivalent? The answer to the question depends on which abstraction you find appropriate. Here are some possible definitions of equivalence.

1. The two programs produce the same output when given the same input.
2. The two programs produce the same output when given the same input, and the two programs are of the same time complexity (with respect to your favorite definition of time complexity).
3. The two programs produce the same output when given the same input, and the two programs take *precisely* the same amount of time.
4. No external observer of the two programs in execution can detect any difference.

*CSE Technical Report CSE-TR-240-95. EECS Department, University of Michigan–Ann Arbor, Ann Arbor, MI, 48109-2122, USA. {gurevich, huggins}@umich.edu. Partially supported by ONR grant N00014-94-1-1182 and NSF grant CCR-92-04742.

```
$in, out$ : **channel of** $Value$
$buf$ : **array** $0..N-1$ **of** $Value$
$p, g$ : **internal** $Natural$ **initially** $0$
$Receiver$ :: $*$[ $p - g \neq N$  $\rightarrow$  $in\ ?\ buf[p\ mod\ N]$;
                                   $p := p + 1$ ]
    ||
$Sender$ ::   $*$[ $p - g \neq 0$  $\rightarrow$  $out\ !\ buf[g\ mod\ N]$;
                                   $g := g + 1$ ]
```

Figure 1: A 2-process ring buffer L_2, in a CSP-like language.

```
$in, out$ : **channel of** $Value$
$buf$ : **array** $0..N-1$ **of** $Value$
$pp, gg$ : **internal array** $0..N-1$ **of** $\{0,1\}$ **initially** $0$
$Buffer(i: 0..N-1)$ ::
      [ $empty$ : $IsNext(pp, i)$  $\rightarrow$  $in\ ?\ buf[i]$;
                                      $pp[i] := (pp[i] + 1)\ mod\ 2$;
    *     $full$ : $IsNext(gg, i)$  $\rightarrow$  $out\ !\ buf[i]$;
                                      $gg[i] := (gg[i] + 1)\ mod\ 2$; ]

  $IsNext(r, i)$  $\triangleq$  **if** $i = 0$  **then**  $r[0] = r[N-1]$
                                   **else**  $r[i] \neq r[i-1]$
```

Figure 2: An N-process ring buffer L_N, in a CSP-like language.

The reader will be able to suggest numerous other reasonable definitions for equivalence. For example, one could substitute space for time in conditions 2 and 3 above. In particular, the nature of an "external observer" in condition 4 is admittedly vague and has several plausible interpretations.

Which notion of equivalence is the best one? Naturally, it depends upon the application. We do not promote any particular notion of equivalence, but note that there are many reasonable definitions. There is no one definition of equivalence that is best for all situations [1].

$\mathcal{R}_{csp}$ and $\mathcal{C}_{csp}$ are indeed "strongly equivalent"; in particular, they are equivalent in the sense of definition 3 above. However, they are not equivalent in the sense of definition 4 for certain external observers, or in the sense of some space-complexity versions of definitions 2 and 3.

Issue 2: Representing Programs as Formulas. Again, we quote Lamport [7]:

> We will not attempt to give a rigorous meaning to the program text. Programming languages evolved as a method of describing algorithms to compilers, not as a method for reasoning about them. We do not know how to write a completely formal proof that two programming language representations of the ring buffer are equivalent. In Section 2, we represent the program formally in TLA, the Temporal Logic of Actions [6].

We believe that it is not only possible but also beneficial to prove the desired equivalence of programs directly. We cannot, however, prove the desired equivalence of $\mathcal{R}_{csp}$ and $\mathcal{C}_{csp}$ because the CSP-like language used lacks a formal semantics. (Actually, the CSP-like language seems to us to be a method for describing algorithms for humans, not compilers.) Instead, we formalize Lamport's interpretations of $\mathcal{R}_{csp}$ and $\mathcal{C}_{csp}$ as evolving algebra programs $\mathcal{R}_{ea}$ and $\mathcal{C}_{ea}$. We then define an appropriate notion of equivalence and show that $\mathcal{R}_{ea}$ and $\mathcal{C}_{ea}$ are equivalent in this sense.

Issue 3: The Formality of Proofs. Continuing, Lamport writes [7]:

> We now give a hierarchically structured proof that Π_2 and Π_N [the TLA translations of $\mathcal{R}_{csp}$ and $\mathcal{C}_{csp}$ – GH] are equivalent [5]. The proof is completely formal, meaning that each step is a mathematical formula. English is used only to explain the low-level reasoning. The entire proof could be carried down to a level at which each step follows from the simple application of formal rules, but such a detailed proof is more suitable for machine checking than human reading. Our complete proof, with "Q.E.D." steps and low-level reasoning omitted, appears in Appendix A.

We prefer to separate the process of explaining a proof to people from the process of computer-aided verification of the same proof [2]. An important benefit of this separation of concerns is that a human-oriented exposition is much easier for humans to read and understand than expositions attempting to satisfy both concerns at once. Writing a good human-oriented proof is the art of creating the correct images in the mind of the reader. Such a proof is amenable to the traditional social process of debugging mathematical proofs. Granted, mathematicians make mistakes and computer-aided verification may be desirable, especially in safety-critical applications. But the two concerns can and should be treated separately.

These disagreements do not mean that our position on "the insubstantiality of processes" is the direct opposite of Lamport's. We simply point out that "the insubstantiality of processes" may itself be in the eye of the beholder. The same two programs can be equivalent with respect to some reasonable definitions of equivalence and inequivalent with respect to others.

This paper is self-contained. In Section 2, we describe Lamport's first-in, first-out ring buffer example. Section 3 contains a brief introduction to evolving algebras, culminating with the evolving algebra programs $\mathcal{R}_{ea}$ and $\mathcal{C}_{ea}$ to be compared. In Section 4, we define a strong version of lock-step equivalence and prove that $\mathcal{R}_{ea}$ and $\mathcal{C}_{ea}$ are equivalent in that sense. Finally, we discuss the inequivalence of $\mathcal{R}_{ea}$ and $\mathcal{C}_{ea}$ in section 5.

2 The Ring Buffer Example

The problem we consider is a first-in, first-out ring buffer of size N. The buffer is implemented as an array of N elements. Input number i (starting with $i = 0$) is stored in position i *mod* N until it is sent out as output number i. Input number i and output number j may occur concurrently only if $i \neq j$ *mod* N. Items may be placed in the buffer if the buffer is not full; of course, items may be sent from the buffer if the buffer is not empty.

Each input or output action is dependent upon other actions within the system. Input number i cannot occur until all previous inputs have occurred and either $i < N$ or output number $i - N$ has occurred. Output number i cannot occur until all previous outputs and input number i have occurred. These dependencies are illustrated pictorially in Figure 3, where circles represent the actions to be taken and arrows represent dependency relationships between actions.

$\mathcal{R}_{csp}$ decomposes this graph into two rows, each row representing one of the processes of the algorithm. Similarly, $\mathcal{C}_{csp}$ decomposes this graph into columns. Figures 4 and 5 show this decomposition.

3 Evolving Algebras

We recall only as much of evolving algebra theory [3] as needed in this paper. The term evolving algebra is often abbreviated *ealgebra* or *EA*. Those already familiar with ealgebras may wish to skip ahead to section 3.6, in which the ealgebras for $\mathcal{R}_{ea}$ and $\mathcal{C}_{ea}$ are given.

3.1 States

A *vocabulary* is a finite collection of function names, each of fixed arity. Every vocabulary contains the nullary function names *true* and *false* and (the names of) the usual Boolean operations.

A *state* s of vocabulary Υ is a nonempty set X, the *superuniverse* of S, together with interpretations of the function names in Υ on X. An r-ary function name is interpreted as a function from X^r to X. The interpretations of function names *true* and *false* are always distinct and are operated upon in the usual way by the Boolean operations. We denote the value of a term t in state s by t_s.

A Boolean-valued function f may be viewed as the set of tuples where it evaluates to *true*. If f is unary it can be viewed as a special *universe*. For example, we may have a universe *nodes* and declare a binary relation *Edge* over the universe of *nodes*; *Edge(x,y)* will hold only if both x and y belong to *nodes*. Such universes allow us to view states as many-sorted structures.

Certain function names in Υ are called *external*; the idea is that the values of external functions will be determined outside of the ealgebra. External functions can be used, for example, to model input provided by the external world. Non-external function names are called *internal*.

If Υ is a vocabulary, Υ^- is the set of internal function names of Υ. If s is a state of vocabulary Υ, s^- is the state of vocabulary Υ^- which agrees with s over all functions in Υ^-.

3.2 Updates

A *location* of state S is a pair $\ell = (f, \bar{x})$, where f is a function name and $\bar{x}$ is a tuple of elements of S whose length equals the arity of f. An *update* of a state S is a pair $\alpha = (\ell, y)$, where ℓ is a location of S and y is an element of S. To *fire* α at S, put y into the location ℓ; that is, if $\ell = (f, \bar{x})$, redefine S to interpret $f(\bar{x})$ as y; nothing else is changed.

An *update set* over a state S is a set of updates of S. A set of updates is *consistent* at S if no two updates in the set have the same location but different values. To fire a consistent set at S, fire all its members simultaneously; to fire an inconsistent set at S, do nothing.

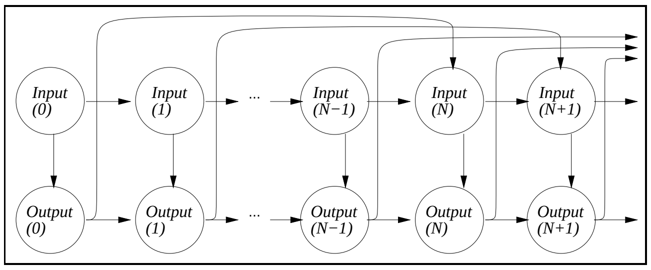

Figure 3: Moves of the ring-buffer algorithm.

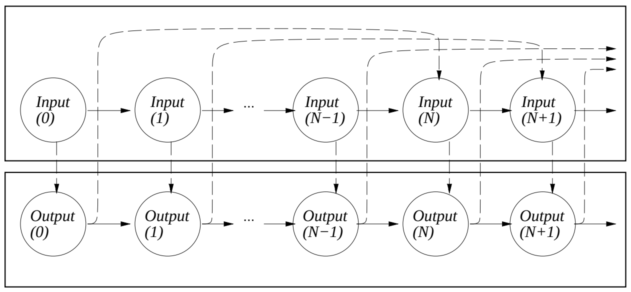

Figure 4: Moves of $\mathcal{R}_{csp}$ and $\mathcal{R}_{ea}$.

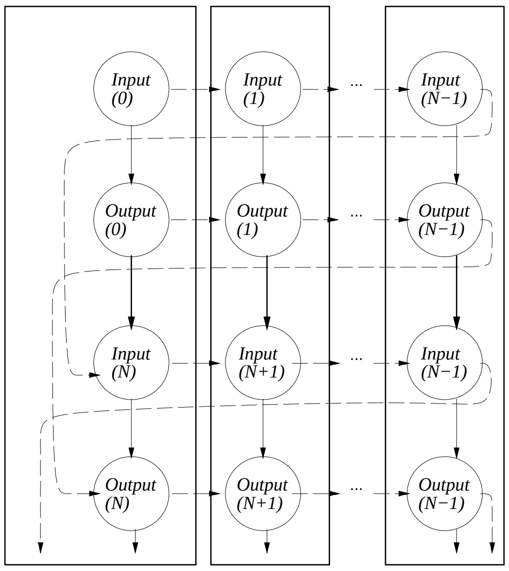

Figure 5: Moves of $\mathcal{C}_{csp}$ and $\mathcal{C}_{ea}$.

3.3 Transition Rules

Executing a rule R at state S is equivalent to executing an appropriate update set. There are three types of basic transition rules.

An *update instruction* R is an expression

$$f(t_1, \ldots, t_r) := t_0$$

where f is a non-static function name of arity r and each t_i is a term. The update set of R at state S has a single element (ℓ, y), where $y = (t_0)_S$ and $\ell = (f, (x_1, \ldots, x_r))$ with $x_i = (t_i)_S$.

A *block rule* R is a sequence of transition rules. The update set of R at S is the union of the update sets of the rules in the sequence at S.

A *conditional rule* R is an expression

```
**if** $g_0$ **then** $R_0$
**elseif** $g_1$ **then** $R_1$
    ⋮
**elseif** $g_k$ **then** $R_k$
**endif**
```

where the g_i are terms and the R_i are rules. (If $g_k \equiv$ *true*, we usually write "**else** R_k".) The update set of R at S is the update set of R_i at S, where g_i is true in S but every g_j with $j < i$ is false in S. If no such i exists, the update set is empty.

3.4 Distributed Evolving Algebras

A *distributed evolving algebra* $\mathcal{A}$ consists of the following:

- A finite indexed set of *modules*, each of which is a transition rule. Intuitively, a module is the program to be executed by one or more agents. Each module is assigned a unique nullary function name.
- A vocabulary Υ containing all functions used in the modules of $\mathcal{A}$, except for the nullary function *Me*. (*Me* will be interpreted differently by each agent and thus must be treated separately.) Υ also contains an additional unary function name *Mod*.
- A collection of Υ-states, called *initial states*.

Structure S of signature Υ is a *state* of $\mathcal{A}$ if module names are interpreted as different elements of S and only finitely many elements a (called *agents*) exist such that *Mod(a)* equals some module name.

$View_a(S)$ is the reduct of S to the functions mentioned in the module *Mod(a)*, expanded by interpreting *Me* as a. Think about $View_a(S)$ as the local state of agent a corresponding to the global state S. To *fire a* at S, fire *Mod(a)* at $View_a(S)$.

3.5 Runs

A *run* ρ of a distributed ealgebra $\mathcal{A}$ of vocabulary Υ from initial state S_0 can be defined as a triple (M, A, σ) satisfying the following conditions.

1. M, the set of *moves* of ρ, is a partially ordered set where every $\{y : y \leq x\}$ is finite.

   Intuitively, $x < y$ means move x completes before move y begins.

2. A assigns elements of S_0 to moves in such a way that every non-empty set $\{x : A(x) = a\}$ is linearly ordered.

   Intuitively, $A(x)$ is the agent performing move x; every agent acts sequentially.

3. σ maps finite initial segments of M (including $\emptyset$) to states of $\mathcal{A}$.

   Intuitively, $\sigma(X)$ is the result of performing all moves of X; $\sigma(\emptyset)$ is an initial state.

4. *Coherence.* If μ is a maximal element of a finite initial segment Y of M, and $X = Y - \{\mu\}$, then $A(\mu)$ is an agent in $\sigma(X)$ and $\sigma(Y)^-$ is obtained by firing $A(\mu)$ at $\sigma(X)$.

Given a finite initial segment Y with a maximal element μ and $X = Y - \{\mu\}$, one may want to associate the state $\sigma(X)$ with μ. One particular choice of X is $\{\nu : \nu < \mu\}$. We define $\Lambda(\mu) = \Lambda_{\mathcal{A}}(\mu) = \sigma(\{\nu : \nu < \mu\})$.

3.6 Evolving Algebras for the Ring Buffer Problem

Here we present evolving algebras $\mathcal{R} = \mathcal{R}_{ea}$ and $\mathcal{C} = \mathcal{C}_{ea}$, corresponding to $\mathcal{R}_{csp}$ and $\mathcal{C}_{csp}$, respectively.

Each ealgebra has universes of *data*, *integers*, int_N (integers modulo N), and int_2. As usual, we identify elements of int_N with their representatives $0, 1, \ldots N-1$ (and similarly for int_2). The buffer is represented by a unary function *Buffer:* $int_N \rightarrow$ *data.*

Following Lamport (but using different names), we model the input and output channels with signaling bits *InBit, OutBit:* int_2 and functions *InData, OutData: data. InBit* and *InData* are external functions; when the external world places a new datum into *InDatum*, *InBit* is also flipped. *MyBit:* int_2 reflects the most recent value of *InBit* that has been seen by the algorithm.

Algorithms perform arithmetic over *integers*, int_N and int_2; we use the standard arithmetic functions with their conventional notations.

3.7 $\mathcal{R}$: The Rows Evolving Algebra

Ealgebra $\mathcal{R}$ describes the 2-process algorithm for the ring buffer. One agent (called I) reads data into the buffer; the other (called O) sends data from the buffer. (I and O are different elements of $\mathcal{R}$, *e.g.* 0 and 1.)

$\mathcal{R}$ uses counters *First, Last: integers* to indicate the current buffer slot to be used for input or output; initially, *First = Last = 0.* The two modules are given in Figure 6.

```
Module: Input
if $MyBit \neq InBit$ and $First - Last \neq N$ then
    $Buffer(First \bmod N) := InData,\ MyBit := 1 - MyBit,\ First := First + 1$
endif

Module: Output
if $Last \neq First$ then
    $OutData := Buffer(Last \bmod N),\ OutBit := 1 - OutBit,\ Last := Last + 1$
endif
```

Figure 6: $\mathcal{R}$: A 2-process ring buffer algorithm.

For any run (M, A, σ) of $\mathcal{R}$, we call a move μ an *input move* (respectively, an *output move*) if $A(\mu) = I$ (resp. O). We call a move μ a *k-slot move* if either μ is an input move and *First* $= k$ in state $\Lambda(\mu)$ or μ is an output move and *Last* $= k$ in state $\Lambda(\mu)$.

Lemma 3.1 *Let Y be a finite initial segment of M, μ a maximal element of Y, and $X = Y - \{\mu\}$. If μ is a k-slot input (respectively, output) move, then we have modulo N:*

1. *$First_{\sigma(X)} = k$ (resp., $Last_{\sigma(X)} = k$)*

2. $First_{\sigma(Y)} = k+1$ (resp., $Last_{\sigma(Y)} = k+1$)

Proof: We prove the property for input moves; the proof for output moves is similar. Condition 2 follows from condition 1 by definition of μ.

Obviously $\Lambda(\mu) \subseteq X$. Since μ is a k-slot input move, $First_{\sigma(\Lambda(\mu))} = k \bmod N$. Any $\nu \in X - \{\mu' : \mu' < \mu\}$ is incomparable with μ and therefore is an output move. Let $s_0 = \Lambda(\mu), s_1, \ldots, s_n = \sigma(X)$ be a sequence of states of $\mathcal{R}$ such that every s^-_{m+1} is obtained from s_m by executing a move of $X - \{\mu' : \mu' < \mu\}$. Since output moves do not alter *First*, $First_{\sigma(X)} = First_{\Lambda(\mu)}$. □

3.8 $\mathcal{C}$: The Columns Evolving Algebra

Ealgebra $\mathcal{C}$ describes the N-process algorithm for our ring buffer. The universe of agents is simply int_N. It uses two nullary functions *get* and *put* with different values (*e.g.* 0 and 1).

Functions *InCount, OutCount*: $int_N \rightarrow int_2$ are used to count (modulo 2) the number of input and output operations performed; initially, both have the value 0 for each agent. Function *Mode*: $int_N \rightarrow modes$ notes whether each agent is receiving input (when *Mode(x) = get*) or producing output (when *Mode(x) = put*). Initially, *Mode(x) = get* for all agents x.

We present in Figure 7 the module used by all agents, which contains a block of two rules.

```
Rule: Get
if MyBit ≠ InBit and Mode(Me) = get and InputTurn(Me) then
    Buffer(Me) := InData, MyBit := 1 - MyBit
    InCount(Me) := 1 - InCount(Me), Mode(Me) := put
endif

Rule: Put
if Mode(Me) = put and OutputTurn(Me) then
    OutData := Buffer(Me), OutBit := 1 - OutBit
    OutCount(Me) := 1 - OutCount(Me), Mode(Me) := get
endif

Abbreviations:
InputTurn(Me) ≡ (Me = 0 and InCount(0) = InCount(N-1))
             or (Me ≠ 0 and InCount(Me) ≠ InCount(Me-1))
OutputTurn(Me) ≡ (Me = 0 and OutCount(0) = OutCount(N-1))
              or (Me ≠ 0 and OutCount(Me) ≠ OutCount(Me-1))
```

Figure 7: $\mathcal{C}$: An N-process ring buffer algorithm.

3.9 Remarks

One may argue that *InBit* and *OutBit* are not used in $\mathcal{R}_{csp}$ and $\mathcal{C}_{csp}$ and should not be used in $\mathcal{R}_{ea}$ and $\mathcal{C}_{ea}$. It is easy to avoid using *InBit* and *OutBit* and make the input and output channels implicit, as in $\mathcal{R}_{csp}$ and $\mathcal{C}_{csp}$. Lamport uses a trick with bits to model his input and output channels, and we followed him (though we altered the trick insignificantly for elegance). Note that both algorithms use identical input and output mechanisms; thus, different presentations of this mechanism do not affect the equivalence of the two algorithms.

One the other hand, one may wish to make the actions of the external world explicit in $\mathcal{R}_{ea}$ and $\mathcal{C}_{ea}$. It is easy to do this, but is not needed for our purposes here.

4 Equivalence

Here we define a strong version of lock-step equivalence for ealgebras which, for brevity, we call simply *lock-step equivalence.* We also define an even stronger version of lock-step equivalence which we call *strict lock-step equivalence.*

4.1 Strict Lock-Step Equivalence

Let $\mathcal{A}$ and $\mathcal{B}$ be two ealgebras with the same superuniverse.

Suppose h is a one-to-one mapping from states of $\mathcal{A}$ onto states of $\mathcal{B}$, such that if $h(s) = t$, then s and t coincide on function names common to $\mathcal{A}$ and $\mathcal{B}$. A sequential run $\langle s_i : i < \alpha \rangle$ of $\mathcal{A}$ is *strictly h-similar* to a sequential run $\langle s'_j : j < \beta \rangle$ if $\alpha = \beta$ and $s'_i = h(s_i)$ for every $i < \alpha$. A partially ordered run (M, A, σ) of $\mathcal{A}$ is *strictly h-similar* to a partially ordered run (M', A', σ') of $\mathcal{B}$ if there is an isomorphism $\eta : M \rightarrow M'$ such that for every finite initial segment X of M, $h(\sigma(X)) = \sigma'(Y)$, where $Y = \{\eta(\mu) : \mu \in X\}$. (Note that we permit $X = \emptyset$, in which case $Y = \emptyset$ as well.)

$\mathcal{A}$ and $\mathcal{B}$ are *strictly h-similar* if every run of $\mathcal{A}$ is strictly h-similar to a run of $\mathcal{B}$, and every run of $\mathcal{B}$ is h^{-1}-similar to a run of $\mathcal{A}$. We say $\mathcal{A}$ and $\mathcal{B}$ are *strictly lock-step equivalent* if $\mathcal{A}$ and $\mathcal{B}$ are h-similar for some h.

Ideally, this is the type of equivalence we would like to use. However, this is not possible. $\mathcal{R}$ uses functions (*First, Last*) which have infinitely many possible values while their counterparts in $\mathcal{C}$ (*InCount, OutCount*) have only finitely many values. Imagine that the universe of *data* is finite. Then $\mathcal{R}$ has infinitely many states while $\mathcal{C}$ has only finitely many states. A one-to-one mapping h will not be possible in this case.

We can easily re-write either $\mathcal{R}$ or $\mathcal{C}$ to make them strictly lock-step equivalent. (For example, it suffices to modify $\mathcal{C}$ to perform math on *InCount* and *OutCount* over *integers* instead of int_2.) However, a slightly weaker version of this equivalence will do.

4.2 Lock-Step Equivalence

Let $\cong$ be an equivalence relation over states of an ealgebra $\mathcal{A}$. Let *Result(a,s)* denote the state resulting from agent a firing at state s. We call $\cong$ a *congruence* if for any states s_1 and s_2 and any agent a, $s_1 \cong s_2 \rightarrow Result(a, s_1) \cong Result(a, s_2)$. We call the equivalence classes of $\cong$ *configurations* (with respect to $\cong$), and denote by $[s]$ the configuration to which state s belongs.

Let $\mathcal{A}$ and $\mathcal{B}$ be ealgebras with congruences $\cong_{\mathcal{A}}$ and $\cong_{\mathcal{B}}$, respectively. (We will drop the subscripts on $\cong$ when no confusion arises.) We assume that $\mathcal{A}$ and $\mathcal{B}$ have the same superuniverse and $\cong_{\mathcal{A}}$ and $\cong_{\mathcal{B}}$ preserve the interpretation of function names common to $\mathcal{A}$ and $\mathcal{B}$. In other words, if f is a common function name, p and q are both states of $\mathcal{A}$ (or both states of $\mathcal{B}$), and $p \cong q$, then $f_p = f_q$; that is, $f(\bar{x})_p = f(\bar{x})_q$ for all $\bar{x}$.

Let h be a one-to-one mapping from the set of configurations of $\mathcal{A}$ onto the set of configurations of $\mathcal{B}$ such that for every function name f common to $\mathcal{A}$ and $\mathcal{B}$, if $h([p]) = [s]$, then $f_p = f_s$. A partially ordered run (M, A, σ) of $\mathcal{A}$ is *h-similar* to a partially ordered run (M', A', σ') of $\mathcal{B}$ if there is an isomorphism $\eta : M \rightarrow M'$ such that for every finite initial segment X of M, $h([\sigma(X)]) = [\sigma'(Y)]$, where $Y = \{\eta(\mu) : \mu \in X\}$.

$\mathcal{A}$ and $\mathcal{B}$ are *h-similar* if every run of $\mathcal{A}$ is h-similar to a run of $\mathcal{B}$, and every run of $\mathcal{B}$ is h^{-1}-similar to a run of $\mathcal{A}$. $\mathcal{A}$ and $\mathcal{B}$ are *lock-step equivalent* (with respect to $\cong_{\mathcal{A}}$ and $\cong_{\mathcal{B}}$) if $\mathcal{A}$ and $\mathcal{B}$ are h-similar for some h.

Note that strict lock-step equivalence is a special case of lock-step equivalence, where $\cong_{\mathcal{A}}$ and $\cong_{\mathcal{B}}$ are both the identity relation.

We will show that $\mathcal{R}$ is lock-step equivalent to $\mathcal{C}$ with respect to the congruences defined below. We assume that $\mathcal{R}$ and $\mathcal{C}$ have the same superuniverse.

Definition 1 *For states s, t of $\mathcal{C}$, $s \cong_{\mathcal{C}} t$ if $s = t$.*

Since each configuration of $\mathcal{C}$ has only one element, we identify a state of $\mathcal{C}$ with its configuration.

Definition 2 *For states p, q of $\mathcal{R}$, $p \cong_{\mathcal{R}} q$ if:*

- $\mathit{Last}_p = \mathit{Last}_q \bmod 2N$
- $(\mathit{First} - \mathit{Last})_p = (\mathit{First} - \mathit{Last})_q$
- *$f_p = f_q$ for all other function names f.*

Let *div* represent integer division: $i \; \mathit{div} \; j = \lfloor i/j \rfloor$.

Lemma 4.1 *If $p \cong_{\mathcal{R}} q$ then we have the following modulo 2:*

- $\mathit{First}_p \; \mathit{div} \; N = \mathit{First}_q \; \mathit{div} \; N$
- $\mathit{Last}_p \; \mathit{div} \; N = \mathit{Last}_q \; \mathit{div} \; N$

Proof: We prove the desired property for *First*; the proof for *Last* is similar.

By the definition of $\cong_{\mathcal{R}}$, we have modulo $2N$ that $\mathit{First}_p = \mathit{Last}_p + (\mathit{First} - \mathit{Last})_p = \mathit{Last}_q + (\mathit{First} - \mathit{Last})_q = \mathit{First}_q$. Thus, there are non-negative integers x_1, x_2, x_3 such that $\mathit{First}_p = 2Nx_1 + Nx_2 + x_3$ and $x_2 \leq 1$ and $x_3 < N$. Since $\mathit{First}_p = \mathit{First}_q \bmod 2N$, there exists a non-negative integer y_1 such that $\mathit{First}_q = 2Ny_1 + Nx_2 + x_3$. Hence $\mathit{First}_p \; \mathit{div} \; N = 2x_1 + x_2$ and $\mathit{First}_q \; \mathit{div} \; N = 2y_1 + x_2$, which are equal modulo 2. □

We define a mapping h from configurations of $\mathcal{R}$ onto configurations of $\mathcal{C}$.

Definition 3 *If $h([p]) = s$, then*

$$\mathit{InCount}(i)_s = \begin{cases} (\mathit{First}_p \; \mathit{div} \; N) \bmod 2 & \text{if } i \geq \mathit{First}_p \bmod N \\ 1 - (\mathit{First}_p \; \mathit{div} \; N) \bmod 2 & \text{otherwise} \end{cases}$$

$$\mathit{OutCount}(i)_s = \begin{cases} (\mathit{Last}_p \; \mathit{div} \; N) \bmod 2 & \text{if } i \geq \mathit{Last}_p \bmod N \\ 1 - (\mathit{Last}_p \; \mathit{div} \; N) \bmod 2 & \text{otherwise} \end{cases}$$

and for all other common function names f, $f_p = f_s$.

Thus, h relates the counters *First* and *Last* used in $\mathcal{R}$ and the counters *InCount* and *OutCount* used in $\mathcal{C}$. We have not said anything about *Mode* because *Mode* is uniquely defined by the rest of the state (see Lemma 4.7 in section 4.3) and is redundant. We only include *Mode* in our specification for $\mathcal{C}$ because we wish the translation of $\mathcal{C}_{csp}$ to $\mathcal{C}$ to be more faithful.

We now prove that $\mathcal{R}$ and $\mathcal{C}$ are h-similar.

4.3 Properties of $\mathcal{R}$

Lemma 4.2 *For any state q of any run of $\mathcal{R}$, $0 \leq (\mathit{First}_q - \mathit{Last}_q) \leq N$.*

Proof: By induction. Initially, $\mathit{First} = \mathit{Last} = 0$.

Let X be a finite initial segment of a run with maximal element x, such that $0 \leq \mathit{First} - \mathit{Last} \leq N$ holds in $p = \sigma(X - \{x\})$. Let $q = \sigma(X)$.

- If x is an execution of rule Input, $(\mathit{First} - \mathit{Last})_p < N$. Input increments *First* and does not alter *Last*; thus, $0 < (\mathit{First} - \mathit{Last})_q \leq N$.
- If x is an execution of rule Output, $\mathit{Last}_p < \mathit{First}_p$ (*i.e.* $0 < (\mathit{First} - \mathit{Last})_p$). Output increments *Last* and does not alter *First*; thus, $0 \leq (\mathit{First} - \mathit{Last})_q < N$. □

Lemma 4.3 *Fix a non-negative integer $k < N$. For any run (M, A, σ) of $\mathcal{R}$, the k-slot moves of M are linearly ordered.*

Proof: By contradiction, suppose that two k-slot moves μ, ν are incomparable. Since all input moves are comparable and all output moves are comparable, one of these moves (say μ) is an input move while the other (that is, ν) is an output move.

Let $X = \{\xi : \xi < \mu \text{ or } \xi < \nu\}$, $p = \sigma(X)$, $Y = X \cup \{\mu\}$, and $Z = X \cup \{\nu\}$. By the coherence condition applied to Y and μ, $A(\mu) = I$ is enabled in X; consequently, $First_p = k \bmod N$ and $First_p$ - $Last_p \neq N$. By the coherence condition applied to Z and ν, $A(\nu) = O$ is enabled in X; consequently, $Last_p = k \bmod N$ and $First_p$ - $Last_p \neq 0$. Combining with Lemma 4.2, we have $0 < First_p - Last_p < N$ and $First_p = Last_p \bmod N$, which is impossible. □

4.4 Properties of $\mathcal{C}$

Lemma 4.4 *For any state t of any run of $\mathcal{C}$, there is an agent $k = In(t) < N$ such that:*

- InputTurn*(Me) is true for agent k and for no other agent.*
- *For all $i < k$, $InCount(i)$ = 1 - $InCount(k)$.*
- *For all $k \leq i < N$, $InCount(i) = InCount(k)$.*

Proof: By induction. Initially, agent 0 (and no other) satisfies InputTurn*(Me)* and $InCount(Me) = 0$ for all agents. Thus, if t is an initial state, $In(t) = 0$.

Let Y be a finite initial segment of a run of $\mathcal{C}$ with maximal element y, such that the requirements hold in $s = \sigma(Y - \{y\})$. Let $t = \sigma(Y)$. Since rule Put does not modify *InCount*, it cannot affect the three requirements.

The desired $In(t) = In(s) + 1 \bmod N$. This is obvious in the case that $In(s) < N-1$. If $In(s) = N-1$, then all values of *InCount* are equal in t, so $In(t) = 0$ satisfies the three requirements. □

Lemma 4.5 *For any state t of any run of $\mathcal{C}$, there is exactly one agent $k = Out(t) < N$ such that:*

- OutputTurn*(Me) is true for agent k and no other agent.*
- *For all $i < k$, $OutCount(i)$ = 1 - $OutCount(k)$.*
- *For all $k \leq i < N$, $OutCount(i) = OutCount(k)$.*

Proof: Parallel to that of the last lemma. □

Lemma 4.6 *For any run (M, A, σ) of $\mathcal{C}$, the set $\{\mu : A(\mu)$ executes rule Get in $\Lambda(\mu)\}$ is linearly ordered, as is the set $\{\mu : A(\mu)$ executes rule Put in $\Lambda(\mu)\}$.*

Proof: We prove the claim for Get; the proof for Put is similar. By contradiction, suppose that are two incomparable moves μ, ν such that Get is enabled in both $\Lambda(\mu)$ and $\Lambda(\nu)$.

Let $X = \{\xi : \xi < \mu \text{ or } \xi < \nu\}$, $p = \sigma(X)$, $Y = X \cup \{\mu\}$, and $Z = X \cup \{\nu\}$. By the coherence condition applied to Y and μ, $A(\mu)$ fires at p; by Lemma 4.4, $In(p) = A(\mu)$. Similarly, $A(\nu)$ fires at p and $In(p) = A(\nu)$. Thus, μ and ν must be moves of the same agent k. But all moves of the same agent are ordered, so μ and ν cannot be incomparable. □

Lemma 4.7 *In any state t of any run of $\mathcal{C}$, for any agent k,*

$$Mode(Me) = \begin{cases} get & \text{if } InCount(Me) = OutCount(Me) \\ put & \text{if } InCount(Me) = 1 - OutCount(Me) \end{cases}$$

Proof: We fix a k and perform an induction over runs. Initially, *Mode(Me)* = *get* and *InCount(Me)* = *OutCount(Me)* = *0* for every agent.

Let Y be a finite initial segment of a run with maximal element y. Thus, the required condition holds in $s = \sigma(Y - \{y\})$.

If move Y is executed by any agent other than k, no functions named in the required condition are affected. If agent k does execute move y, there are two cases.

- If y is an execution of rule Get, the guard ensures that *Mode(Me)* = *get*. Get sets *Mode(Me)* to *put* and executes *InCount(Me) := 1 - InCount(Me)*, maintaining the requirement.
- If y is an execution of rule Put, the guard ensures that *Mode(Me)* = *put*. Put sets *Mode(Me)* to *get* and executes *OutCount(Me) := 1 - OutCount(Me)*, maintaining the requirement. □

Remark. This lemma shows that function *Mode* is indeed redundant. The only substantial difference between the signatures of $\mathcal{R}$ and $\mathcal{C}$ is the presence of *First* and *Last* in $\mathcal{R}$ and the presence of *InCount* and *OutCount* in $\mathcal{C}$.

4.5 Proof of Equivalence

Lemma 4.8 *If $h([p]) = s$, then In(s) = $First_p$ mod N and Out(s) = $Last_p$ mod N.*

Proof: Recall that *In(s)* is the agent k for which INPUTTURN$(k)_s$ holds. Lemma 4.4 asserts that $InCount_s$ has one value everywhere below k and another value everywhere else. By definition of h, this "switch-point" in *InCount* occurs at $First_p$ *mod N*. The proof for *Out(s)* is similar. □

Lemma 4.9 *Rule Input is enabled in state p of $\mathcal{R}$ iff rule Get is enabled in state $s = h([p])$ of $\mathcal{C}$ for agent In(s).*

Proof: Let $k = In(s)$, so that INPUTTURN$(k)_s$ holds. Both Input and Get have $MyBit \neq InBit$ in their guards. It thus suffices to show that $(First - Last)_p \neq N$ iff $Mode(k)_s = get$. By Lemma 4.7, it suffices to show that $(First - Last)_p \neq N$ iff $InCount(k)_s = OutCount(k)_s$.

Suppose $(First - Last)_p \neq N$. There exist non-negative integers a_1, a_2, a_3, a_4 such that $First_p = a_1N + a_3$, $Last_p = a_2N + a_4$, and $a_3, a_4 < N$. (Note that by 4.8, $k = First_p \text{ mod } N = a_3$.)

By Lemma 4.2, $0 \leq (First - Last)_p < N$. There are two cases.

- $a_1 = a_2$ and $a_3 \geq a_4$. By definition of h, we have that, modulo 2, $InCount(a_3)_s = First_p \text{ div } N = a_1$ and $OutCount(i)_s = Last_p \text{ div } N = a_2$ for all $i \geq Last_p \text{ mod } N = a_4$. Since $a_3 \geq a_4$, $(OutCount(a_3) = a_2 = a_1 = InCount(a_3))_s$.
- $a_1 = (a_2 + 1)$ and $a_3 < a_4$. By definition of h, we have that, modulo 2, $InCount(a_3)_s = First_p \text{ div } N = a_1$ and $OutCount(i)_s = 1 - (Last_p \text{ div } N) = a_2 + 1$ for all $i < Last_p \text{ mod } N = a_4$. Since $a_3 < a_4$, $(OutCount(a_3) = a_2 + 1 = a_1 = InCount(a_3))_s$.

On the other hand, suppose $(First - Last)_p = N$. Then $First_p$ *div N* and $Last_p$ *div N* differ by 1. By definition of h, $InCount(i)_s = 1 - OutCount(i)_s$ for all i, including k. □

Lemma 4.10 *Rule Output is enabled in state p iff rule Put is enabled in state $s = h([p])$ for agent Out(s).*

Proof: Similar to that of the last theorem. □

Lemma 4.11 *Suppose that rule Input is enabled in a state p of $\mathcal{R}$ and rule Get is enabled in a state $s = h([p])$ of $\mathcal{C}$ for agent In(s). Let q = Result(I,p) and t = Result(In(s), s). Then $t = h([q])$.*

Proof: We check that $h([q]) = t$.

- Both Input and Get execute *MyBit := InBit*
- Input executes *Buffer(First mod N) := InData.* Get executes *Buffer(In(s)) := InData.* By Lemma 4.8, *In(s)* $=$ *First*$_p$ *mod N*, so these updates are identical.
- Input executes *First := First + 1.* Get executes *InCount(In(s)) := 1 - InCount(In(s)).* The definition of h and the fact that *InCount(i)*$_s$ $=$ *InCount(i)*$_{h([p])}$ imply that *InCount(i)*$_t$ $=$ *InCount(i)*$_{h([q])}$.
- Get executes *Mode(In(s)) := put.* By Lemma 4.7, this update is redundant and need not have a corresponding update in Input. □

Lemma 4.12 *Suppose that rule* Output *is enabled in a state p of* $\mathcal{R}$ *and rule* Put *is enabled in a state* $s = h([p])$ *of* $\mathcal{C}$ *for agent Out(s). Let q = Result(O,p) and t = Result(Out(s), s). Then* $t = h([q])$.

Proof: Parallel to that of the last theorem. □

Theorem 1 $\mathcal{R}$ *is lock-step equivalent to* $\mathcal{C}$.

Proof: Let $\Lambda(\mu) = \Lambda_{\mathcal{R}}(\mu)$ and $\Lambda'(\mu) = \Lambda_{\mathcal{C}}(\mu)$.

We begin by showing that any run (M, A, σ) of $\mathcal{R}$ is h-similar to a run of $\mathcal{C}$, using the definition of h given earlier. Construct a run (M, A', σ') of $\mathcal{C}$, where $\sigma'(X) = h([\sigma(X)])$ and A' is defined as follows. Let μ be a move of M and let $s = h([\Lambda(\mu)])$. Then $A'(\mu) = In(s)$ if $A(\mu) = I$, and $A'(\mu) = Out(s)$ if $A(\mu) = O$.

We check that (M, A', σ') satisfies the four requirements for a run of $\mathcal{C}$ stated in Section 3.5.

1. Trivial, since (M, A, σ) is a run.
2. By Lemma 4.3, it suffices to show that for any μ, if $A'(\mu) = k$, then $A(\mu)$ is a k-slot move. By the construction above and Lemma 4.8, we have modulo N that $k = In(s) = First_{\Lambda(\mu)}$ if $A(\mu) = I$ and $k = Out(s) = Last_{\Lambda(\mu)}$ if $A(\mu) = O$. In either case, μ is a k-slot move.
3. Since $\sigma' = h \circ \sigma$, σ' maps finite initial segments of M to states of $\mathcal{C}$.
4. *Coherence.* Let Y be a finite initial segment of M with a maximal element μ, and $X = Y - \{\mu\}$. Thus *Result(A(μ),σ(X))* $= \sigma(Y)$. By Lemma 4.9 or 4.10, $A'(\mu)$ is enabled in $\sigma'(X)$. By Lemma 4.11 or 4.12, *Result(*$A'(\mu), \sigma'(X)$*)* $= \sigma'(Y)$.

Continuing, we must also show that for any run (M, A', σ') of $\mathcal{C}$, there is a run (M, A, σ) of $\mathcal{R}$ which is h-similar to it.

We define A as follows. Consider the action of agent $A'(\mu)$ at state $\Lambda'(\mu)$. If $A'(\mu)$ executes rule Get, set $A(\mu) = I$. If $A'(\mu)$ executes rule Put, set $A(\mu) = O$.

We check that $\{\mu : A(\mu) = I\}$ is linearly ordered. By Lemma 4.6, it suffices to show that if $A(\mu) = I$, then $A'(\mu)$ executes Get in state $\Lambda'(\mu)$ — which is true by construction of A. $\{\mu : A(\mu) = O\}$ is linearly ordered by a similar argument.

We define σ inductively over finite initial segments of M. $\sigma(\emptyset)$ is the unique initial state in $h^{-1}(\sigma'(\emptyset))$.

Let Y be a finite initial segment with a maximal element μ such that σ is defined at $X = Y - \{\mu\}$. Choose $\sigma(Y)$ from $h^{-1}(\sigma'(Y))$ such that $\sigma(Y)^- = Result(A(\mu), \sigma(X))$. Is it possible to select such a $\sigma(Y)$? Yes. By Lemma 4.9 or 4.10, $A(\mu)$ is enabled in $\sigma(X)$ iff $A'(\mu)$ is enabled in $\sigma'(X)$. By Lemma 4.11 or 4.12, *Result(*$A(\mu), \sigma(X)$*)* $\in$ h^{-1}*(Result(*$A'(\mu), \sigma'(\mu)$*))*.

It is easy to check that (M, A, σ) is a run of $\mathcal{R}$ which is h-similar to (M, A', σ'). □

5 Inequivalence

Even though we have proven that $\mathcal{R}$ and $\mathcal{C}$ are lock-step equivalent, there are meaningful differences between $\mathcal{R}$ and $\mathcal{C}$. We consider several differences which could be detected by an external observer.

Magnitude of Values. $\mathcal{R}$ uses unrestricted integers as its counters; in constrast, $\mathcal{C}$ uses only single bits for the same purpose. Imagine that the universe of *data* is finite and small, and that an observer uses a computer with little memory to execute $\mathcal{R}$ and $\mathcal{C}$. $\mathcal{R}$'s counters may eventually exceed the memory capacity of the computer. $\mathcal{C}$ would have no such difficulties.

Types of Sharing. $\mathcal{R}$ shares access to the buffer between both agents; each agent in $\mathcal{C}$ has exclusive access to its portion of the buffer. Conversely, agents in $\mathcal{C}$ share access to both the input and output channels; each agent in $\mathcal{R}$ has exclusive access to one channel. Imagine an architecture in which input or output channels may not be exclusively held by an agent. It may not be possible to implement $\mathcal{R}$ directly on such a system.

Degree of Sharing. $\mathcal{R}$ has $N+2$ shared locations: the N locations of the buffer and 2 counter variables. $\mathcal{C}$ shares access to $2(N+2)$ locations: the input and output channel functions and the $2N$ counter variables. Sharing locations can be an expensive matter; if a location is being shared, some provision must be made for handling read/write conflicts to a given location. Imagine that a user must pay (either in time or in money) for each shared location (but not for private variables), regardless of size. In such a scenario, $\mathcal{C}$ would be more expensive than $\mathcal{R}$ to run.

These constrasts can be made more a little more dramatic. For example, one could construct an algorithm which uses $2N$ agents, each of which is responsible for an input or output action (but not both) to a particular buffer position. Virtually all of the locations it uses will be shared; yet, it is lock-step equivalent to $\mathcal{R}$ and $\mathcal{C}$. Yet few people would use this algorithm instead of $\mathcal{R}$ or $\mathcal{C}$ because it combines their disadvantages. Alternatively, one could write a single processor (sequential) algorithm which is equivalent in a different sense to $\mathcal{R}$ and $\mathcal{C}$; it would produce the same output as $\mathcal{R}$ and $\mathcal{C}$ when given the same input, but would not allow all orderings of actions possible for $\mathcal{R}$ and $\mathcal{C}$.

References

[1] Y. Gurevich. "Logic and the challenge of computer science." In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pp. 1–57, Computer Science Press, 1988.

[2] Y. Gurevich, "Logic Activities in Europe", ACM SIGACT News 25, No. 2, June 1994, 11–24.

[3] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide", in *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995.

[4] C.A.R. Hoare, "Communicating sequential processes." *Communications of the ACM*, 21(8):666-667, August 1978.

[5] L. Lamport, "How to write a proof." Research Report 94, Digital Equipment Corporation, Systems Research Center, February 1993. To appear in American Mathematical Monthly.

[6] L. Lamport, "The temporal logic of actions." *ACM Transactions on Programming Languages and Systems*, 16(3):872-923, May 1994.

[7] L. Lamport, "Processes are in the Eye of the Beholder." Research Report 94, Digital Equipment Corporation, Systems Research Center, December 1994.