Equivalence Is In The Eye Of The Beholder

Yuri Gurevich [1] and James K. Huggins [1]

EECS Department, University of Michigan, Ann Arbor, MI, 48109-2122, USA.

1 Introduction

This is a reaction to Leslie Lamport's "Processes are in the Eye of the Beholder" [13]. 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. He represents the two algorithms by temporal formulas and proves the equivalence of the two temporal formulas.

We analyze in what sense the two algorithms are and are not equivalent. There is no one notion of equivalence appropriate for all purposes and thus the "insubstantiality of processes" may itself be in the eye of the beholder. There are other issues where we disagree with Lamport. In particular, we give a direct equivalence proof for two programs without representing them by means of temporal formulas.

This paper is self-contained. In the remainder of this section, we explain the two ring buffer algorithms and discuss our disagreements with Lamport. In Section 2, we give a brief introduction to evolving algebras. In Section 3, we present our formalizations of the ring buffer algorithms as evolving algebras. In Section 4, we define a version of lock-step equivalence and prove that our formalizations of these algorithms are equivalent in that sense. Finally, we discuss the inequivalence of these algorithms in Section 5.

[1] Partially supported by ONR grant N00014-94-1-1182 and NSF grant CCR-95-04375. The first author was with the Centre National de la Recherche Scientifique, Paris, France, during the final stage of this work.

1.1 Ring Buffer Algorithms

The ring buffer in question is implemented by means of an array of N elements. The ith input (starting with $i = 0$) is stored in slot $i \bmod N$ until it is sent out as the ith output. Items may be placed in the buffer if and only if the buffer is not full; of course, items may be sent from the buffer if and only if the buffer is not empty. Input number i cannot occur until (1) all previous inputs have occurred and (2) either $i < N$ or else output number $i - N$ has occurred. Output number i cannot occur until (1) all previous outputs have occurred and (2) input number i has occurred. These dependencies are illustrated pictorially in Figure 1, where circles represent the actions to be taken and arrows represent dependency relationships between actions.

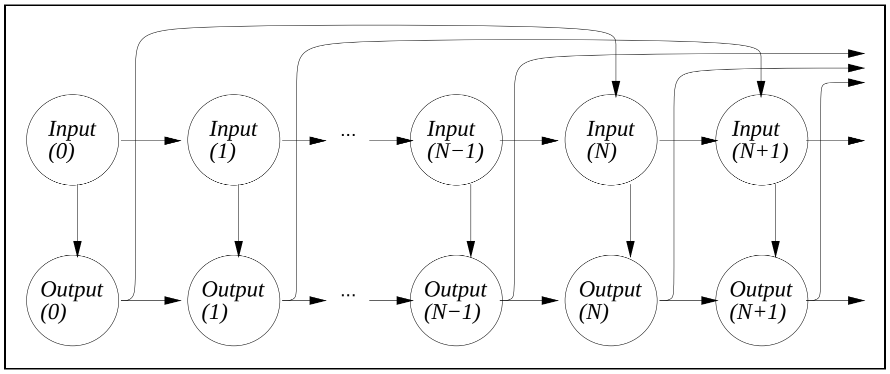

Fig. 1. Moves of the ring-buffer algorithm.

Lamport writes the two programs in a semi-formal language reminiscent of CSP [9] which we call Pseudo-CSP. The first program, which we denote by $\mathcal{R}_{\text{pcsp}}$, is shown in Figure 2. It operates the buffer using two processes; one handles input into the buffer and the other handles output from the buffer. It gives rise to a row-wise decomposition of the graph of moves, as shown in Figure 3. The second program, which we denote by $\mathcal{C}_{\text{pcsp}}$, is shown in Figure 4. It uses N processes, each managing input and output for one particular slot in the buffer. It gives rise to a column-wise decomposition of the graph of moves, as shown in Figure 5.

In Pseudo-CSP, the semicolon represents sequential composition, $\|$ represents parallel composition, and $*$ represents iteration. The general meanings of ? and ! are more complicated; they indicate synchronization. In the context of $\mathcal{R}_{\text{pcsp}}$ and $\mathcal{C}_{\text{pcsp}}$, "in ?" is essentially a command to place the current input into the given slot, and "out !" is essentially a command to send out the datum in the given slot as an output. In Section 3, we will give a more complete explanation of the two programs in terms of evolving algebras.

After presenting the two algorithms in Pseudo-CSP, Lamport describes them by means of formulas in TLA, the Temporal Logic of Actions [12], and proves

```
$in,\ out$ : channel of $Value$
$buf$ : array $0 \ldots N-1$ of $Value$
$p, g$ : internal $Natural$ initially $0$
$Receiver :: *\left[\begin{array}{l} p - g \neq N \rightarrow in\ ?buf\,[p \bmod N]; \\ \qquad p := p + 1 \end{array}\right]$
    $\|$
$Sender :: *\left[\begin{array}{l} p - g \neq 0 \rightarrow out\ !buf\,[g \bmod N]; \\ \qquad g := g + 1 \end{array}\right]$
```

Fig. 2. A two-process ring buffer $\mathcal{R}_{\text{pcsp}}$, in Pseudo-CSP.

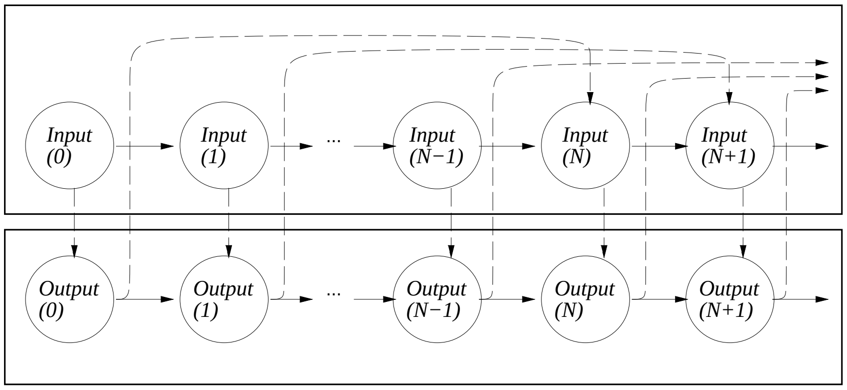

Fig. 3. Moves of $\mathcal{R}_{\text{pcsp}}$.

the equivalence of the two formulas in TLA. He does not prove that the TLA formulas are equivalent to the corresponding Pseudo-CSP programs. The Pseudo-CSP presentations are there only to guide the reader's intuition. As we have mentioned, Pseudo-CSP is only semi-formal; neither the syntax nor the semantics of it is given precisely.

However, Lamport provides a hint as to why the two programs themselves are equivalent. There is a close correspondence of values between p and pp, and between g and gg. Figure 6, taken from [13], illustrates the correspondence between p and pp for $N = 4$. The nth row describes the values of variables p and pp after n inputs. The predicate IsNext(pp,i) is intended to be true only for one array position i at any state (the position that is going to be active); the box indicates that position.

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

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

Fig. 4. An N process ring buffer $\mathcal{C}_{\text{pcsp}}$, in Pseudo-CSP.

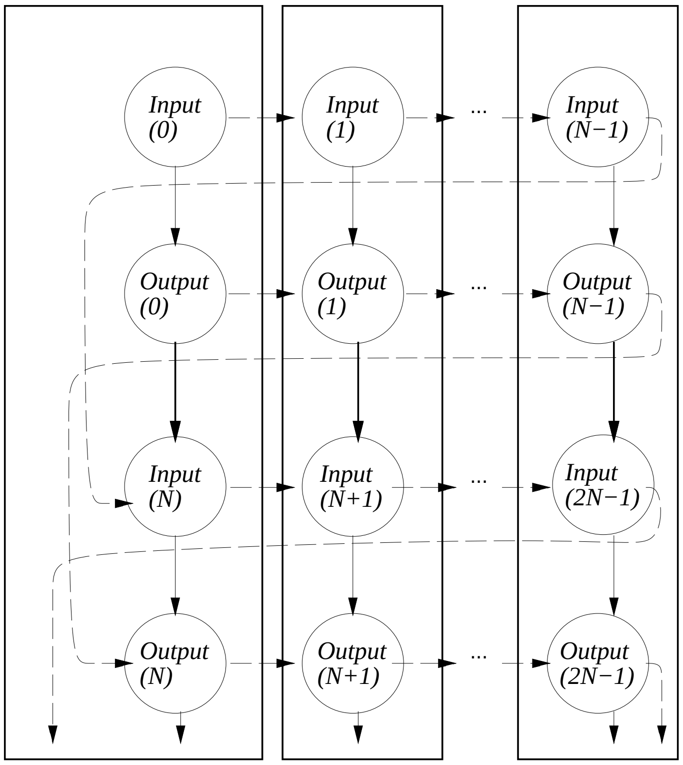

Fig. 5. Moves of $\mathcal{C}_{\text{pcsp}}$.

p	$pp[0]$	$pp[1]$	$pp[2]$	$pp[3]$
0	$\boxed{0}$	0	0	0
1	1	$\boxed{0}$	0	0
2	1	1	$\boxed{0}$	0
3	1	1	1	$\boxed{0}$
4	$\boxed{1}$	1	1	1
5	0	$\boxed{1}$	1	1
6	0	0	$\boxed{1}$	1
⋮	⋮	⋮	⋮	⋮

Fig. 6. The correspondence between values of pp and p, for $N = 4$.

1.2 Discussion

There are three issues where we disagree with Lamport.

Issue 1: The Notion of Equivalence. What does it mean that two programs are equivalent? In our opinion, the answer to the question depends on the desired abstraction [4]. There are many reasonable definitions of equivalence. Here are some examples.

(i) The two programs produce the same output on the same input.
(ii) The two programs produce the same output on the same input, and the two programs are of the same time complexity (with respect to your favorite definition of time complexity).
(iii) Given the same input, the two programs produce the same output and take *precisely* the same amount of time.
(iv) No observer of the execution of the two programs can detect any difference.

The reader will be able to suggest numerous other reasonable definitions for equivalence. For example, one could substitute space for time in conditions (ii) and (iii) above. The nature of an "observer" in condition (iv) admits different plausible interpretations, depending upon what aspects of the execution the observer is allowed to observe.

Let us stress that we do not promote any particular notion of equivalence or any particular class of such notions. We only note that there are different reasonable notions of equivalence and there is no one notion of equivalence that is best for all purposes. The two ring-buffer programs are indeed "strongly equivalent"; in particular, they are equivalent in the sense of definition (iii) above. However, they are not equivalent in the sense of definition (iv) for certain observers, or in the sense of some space-complexity versions of definitions (ii) and (iii). See Section 5 in this connection.

Issue 2: Representing Programs as Formulas. Again, we quote Lamport [13]:

> *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 [12].*

We believe that it is not only possible but also beneficial to give a rigorous meaning to one's programming language and to prove the desired equivalence of programs directly. The evolving algebra method has been used to give rigorous meaning to various programming languages [1,10]. In a similar way, one may try to give formal semantics to Pseudo-CSP (which is used in fact for describing algorithms to humans, not compilers). Taking into account the modesty of our goals in this paper, we do not do that and represent $\mathcal{R}_{\text{pcsp}}$ and $\mathcal{C}_{\text{pcsp}}$ directly as evolving algebra programs $\mathcal{R}_{\text{ea}}$ and $\mathcal{C}_{\text{ea}}$ and then work with the two evolving algebras.

One may argue that our translation is not perfectly faithful. Of course, no translation from a semi-formal to a formal language can be proved to be faithful. We believe that our translation is reasonably faithful; we certainly did not worry about the complexity of our proofs as we did our translations. Also, we do not think that Lamport's TLA description of the Pseudo-CSP is perfectly faithful (see the discussion in subsection 3.2) and thus we have two slightly different ideals to which we can be faithful. In fact, we do not think that perfect faithfulness is crucially important here. We give two programming language representations $\mathcal{R}_{\text{ea}}$ and $\mathcal{C}_{\text{ea}}$ of the ring buffer reflecting different decompositions of the buffer into processes. Confirming Lamport's thesis, we prove that the two programs are equivalent in a very strong sense; our equivalence proof is direct. Then we point out that our programs are inequivalent according to some natural definitions of equivalence. Moreover, the same inequivalence arguments apply to $\mathcal{R}_{\text{pcsp}}$ and $\mathcal{C}_{\text{pcsp}}$ as well.

Issue 3: The Formality of Proofs. Continuing, Lamport writes [13]:

> *We now give a hierarchically structured proof that Π_2 and Π_N [the TLA translations of $\mathcal{R}_{\mathrm{pcsp}}$ and $\mathcal{C}_{\mathrm{pcsp}}$ – GH] are equivalent [11]. 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 [7]. 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. In this connection we note that a human-oriented proof can be a starting point for mechanical verification. Let us stress also that a human-oriented proof need not be less precise than a machine-oriented proof; it simply addresses a different audience.

Revisiting Lamport's Thesis 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.

2 Evolving Algebras

Evolving algebras were introduced in [5]; a more detailed definition has appeared in [6]. Since its introduction, this methodology has been used for a wide variety of applications: programming language semantics, hardware specification, protocol verification, *etc.*. It has been used to show equivalences of various kinds, including equivalences across a variety of abstraction levels for various real-world systems, *e.g.* [3]. See [1,10] for numerous other examples.

We recall here only as much of evolving algebra definitions [6] as needed in this paper. Evolving algebras (often abbreviated *ealgebras* or *EA*) have many other capabilities not shown here: for example, creating or destroying agents during the evolution.

Those already familiar with ealgebras may wish to skip this section.

2.1 States

States are essentially logicians' structures except that relations are treated as special functions. They are also called *static algebras* and indeed they are algebras in the sense of the science of universal algebra.

A *vocabulary* is a finite collection of function names, each of fixed arity. Every vocabulary contains the following *logic symbols*: nullary function names *true, false, undef*, the equality sign, (the names of) the usual Boolean operations and (for convenience) a unary function name Bool. Some function symbols are tagged as relation symbols (or predicates); for example, Bool and the equality sign are predicates.

A *state S of vocabulary* Υ is a non-empty set X (the *basic set* or *superuniverse* of S), together with interpretations of all function symbols in Υ over X (the *basic functions* of S). A function symbol f of arity r is interpreted as an r-ary operation over X (if $r = 0$, it is interpreted as an element of X). The interpretations of predicates (the *basic relations*) and the logic symbols satisfy the following obvious requirements. The elements (more exactly, the interpretations of) *true* and *false* are distinct. These two elements are the only possible values of any basic relation and the only arguments where Bool produces *true*. They are operated upon in the usual way by the Boolean operations. The interpretation of *undef* is distinct from those of *true* and *false*. The equality sign is interpreted as the equality relation. We denote the value of a term t in state S by t_S.

Domains. Let f be a basic function of arity r and $\bar{x}$ range over r-tuples of elements of S. If f is a basic relation then the *domain of f* at S is $\{\bar{x} : f(\bar{x}) = \text{true}\}$. Otherwise the *domain of f* at S is $\{\bar{x} : f(\bar{x}) \neq \text{undef}\}$.

Universes. A basic relation f may be viewed as the set of tuples where it evaluates to *true*. If f is unary it can be viewed as a *universe*. For example, Bool is a universe consisting of two elements (named) *true* and *false*. Universes allow us to view states as many-sorted structures.

Types. Let f be a basic function of arity r and $U_0, \ldots, U_r$ be universes. We say that f is *of type* $U_1 \times \cdots \times U_r \rightarrow U_0$ in the given state if the domain of f is $U_1 \times \cdots \times U_r$ and $f(\bar{x}) \in U_0$ for every $\bar{x}$ in the domain of f. In particular, a nullary f is of type U_0 if (the value of) f belongs to U_0.

Example. Consider a directed ring of nodes with two tokens; each node may be colored or uncolored. We formalize this as a state as follows. The superuniverse contains a non-empty universe Nodes comprising the nodes of the ring. Also present is the obligatory two-element universe Bool, disjoint from Nodes. Finally, there is an element (interpreting) *undef* outside of Bool and outside of Nodes. There is nothing else in the superuniverse. (Usually we skip the descriptions of Bool and *undef*). A unary function Next indicates the successor to a given node in the ring. Nullary functions Token1 and Token2 give the positions of the two tokens. A unary predicate Colored indicates whether the given node is colored.

2.2 Updates

There is a way to view states which is unusual to logicians. View a state as a sort of memory. Define a *location* of a state S to be a pair $\ell = (f, \bar{x})$, where f is a function name in the vocabulary of S and $\bar{x}$ is a tuple of elements of (the superuniverse of) S whose length equals the arity of f. (If f is nullary, ℓ is simply f.) In the two-token ring example, let a be any node (that is, any element of the universe Nodes). Then the pair (Next,a) is a location.

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 (including the superuniverse) is changed. We say that an update (ℓ, y) of state S is *trivial* if y is the content of ℓ in S. In the two-token ring example, let a be any node. Then the pair (Token1, a) is an update. To fire this update, move the first token to the position a.

Remark to a curious reader. If $\ell =$ (Next,a), then (ℓ, a) is also an update. To fire this update, redefine the successor of a; the new successor is a itself. This update destroys the ring (unless the ring had only one node). To guard from such undesirable changes, the function Next can be declared static (see [6]) which will make any update of Next illegal.

An *update set* over a state S is a set of updates of S. An update set 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. In the two-token ring example, let a, b be two nodes. Then the update set $\{(Token1, a), (Token1, b)\}$ is consistent if and only if $a = b$.

2.3 Basic Transition Rules

We introduce rules for changing states. The semantics for each rule should be obvious. At a given state S whose vocabulary includes that of a rule R, R gives rise to an update set $\mathrm{US}(R, S)$; to execute R at S, one fires $\mathrm{US}(R, S)$. We say that R is *enabled* at S if $\mathrm{US}(R, S)$ is consistent and contains a non-trivial update. We suppose below that a state of discourse S has a sufficiently rich vocabulary.

An *update instruction* R has the form

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

where f is a function name of arity r and each t_i is a term. (If $r = 0$ we write "$f := t_0$" rather than "$f() := t_0$".) The update set $\mathrm{US}(R, S)$ contains a single element (ℓ, y), where y is the value $(t_0)_S$ of t_0 at S and $\ell = (f, (x_1, \ldots, x_r))$ with $x_i = (t_i)_S$. In other words, to execute R at S, set $f((t_1)_S, \ldots, (t_r)_S)$ to $(t_0)_S$ and leave the rest of the state unchanged. In the two-token ring example, "Token1 := Next(Token2)" is an update instruction. To execute it, move token 1 to the successor of (the current position of) token 2.

A *block rule* R is a sequence $R_1, \ldots, R_n$ of transition rules. To execute R at S, execute all the constituent rules at S simultaneously. More formally, $\mathrm{US}(R, S) = \bigcup_{i=1}^{n} \mathrm{US}(R_i, S)$. (One is supposed to write "**block**" and "**endblock**" to denote the scope of a block rule; we often omit them for brevity.) In the two-token ring example, consider the following block rule:

Token1 := Token2
Token2 := Token1

To execute this rule, exchange the tokens. The new position of Token1 is the old position of Token2, and the new position of Token2 is the old position of Token1.

A *conditional rule* R has the form

if g **then** R_0 **endif**

where g (the *guard*) is a term and R_0 is a rule. If g holds (that is, has the same value as *true*) in S then $\mathrm{US}(R, S) = \mathrm{US}(R_0, S)$; otherwise $\mathrm{US}(R, S) = \emptyset$. (A more general form is "**if** g **then** R_0 **else** R_1 **endif**", but we do not use it in this paper.) In the two-token ring example, consider the following conditional rule:

```
if Token1 = Token2 then
    Colored(Token1) := true
endif
```

Its meaning is the following: if the two tokens are at the same node, then color that node.

2.4 Rules with Variables

Basic rules are sufficient for many purposes, e.g. to give operational semantics for the C programming language [8], but in this paper we need two additional rule constructors. The new rules use variables. Formal treatment of variables requires some care but the semantics of the new rules is quite obvious, especially because we do not need to nest constructors with variables here. Thus we skip the formalities and refer the reader to [6]. As above S is a state of sufficiently rich vocabulary.

A *parallel synchronous rule* (or *declaration rule*, as in [6]) R has the form:

```
var $x$ ranges over $U$
    $R(x)$
endvar
```

where x is a variable name, U is a universe name, and $R(x)$ can be viewed as a rule template with free variable x. To execute R at S, execute simultaneously all rules $R(u)$ where u ranges over U. In the two-token ring example, (the execution of) the following rule colors all nodes except for the nodes occupied by the tokens.

```
var $x$ ranges over Nodes
    if $x \neq$ Token1 and $x \neq$ Token2 then
        Colored(x) := true
    endif
endvar
```

A *choice rule* R has the form

```
choose $x$ in $U$
    $R(x)$
endchoose
```

where x, U and $R(x)$ are as above. It is nondeterministic. To execute the choice rule, choose arbitrarily one element u in U and execute the rule $R(u)$. In the two-token ring example, each execution of the following rule either colors an unoccupied node or does nothing.

```
choose $x$ in Nodes
    if $x \neq$ Token1 and $x \neq$ Token2 then
        Colored(x) := true
    endif
endchoose
```

2.5 Distributed Evolving Algebra Programs

Let Υ be a vocabulary that contains the universe *Agents*, the unary function *Mod* and the nullary function *Me*. A *distributed EA program* Π of vocabulary Υ consists of a finite set of *modules*, each of which is a transition rule with function names from Υ. Each module is assigned a different name; these names are nullary function names from Υ different from *Me*. Intuitively, a module is the program to be executed by one or more agents.

A (global) *state* of Π is a structure S of vocabulary $\Upsilon - \{\mathrm{Me}\}$ where different module names are interpreted as different elements of S and the function *Mod* assigns (the interpretations of) module names to elements of *Agents*; *Mod* is undefined (that is, produces *undef*) otherwise. If *Mod* maps an element α to a module name M, we say that α is an *agent* with program M.

For each agent α, $\mathrm{View}_\alpha(S)$ is the reduct of S to the collection of functions mentioned in the module $\mathrm{Mod}(\alpha)$, expanded by interpreting *Me* as α. Think about $\mathrm{View}_\alpha(S)$ as the local state of agent α corresponding to the global state S. We say that an agent α is *enabled* at S if $\mathrm{Mod}(\alpha)$ is enabled at $\mathrm{View}_\alpha(S)$; that is, if the update set generated by $\mathrm{Mod}(\alpha)$ at $\mathrm{View}_\alpha(S)$ is consistent and contains a non-trivial update. This update set is also an update set over S. To *fire* α at S, execute that update set.

2.6 Runs

In this paper, agents are not created or destroyed. Taking this into account, we give a slightly simplified definition of runs.

A *run* ρ of a distributed ealgebra program Π of vocabulary Υ from the initial state S_0 is a triple (M, A, σ) satisfying the following conditions.

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

Intuitively, $\nu < \mu$ means that move ν completes before move μ begins. If M is totally ordered, we say that ρ is a *sequential* run.

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

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

3. σ maps finite initial segments of M (including $\emptyset$) to states of Π.

Intuitively, $\sigma(X)$ is the result of performing all moves of X; $\sigma(\emptyset)$ is the initial state S_0. States $\sigma(X)$ are the *states of* ρ.

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

It may be convenient to associate particular states with single moves. We define $\Lambda(\mu) = \sigma(\{\nu : \nu < \mu\})$.

The definition of runs above allows no interaction between the agents on the one side and the external world on the other. In such a case, a distributed evolving algebra is given by a program and the collection of initial states. In a more general case, the environment can influence the evolution. Here is a simple way to handle interaction with the environment which suffices for this paper.

Declare some basic functions (more precisely, some function names) *external.* Intuitively, only the outside world can change them. If S is a state of Π let S^- be the reduct of S to (the vocabulary of) non-external functions. Replace the coherence condition with the following:

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

In applications, external functions usually satisfy certain constraints. For example, a nullary external function Input may produce only integers. To reflect such constraints, we define *regular runs* in applications. A distributed evolving algebra is given by a program, the collection of initial states and the collection of regular runs. (Of course, regular runs define the initial states, but it may be convenient to specify the initial states separately.)

3 The Ring Buffer Evolving Algebras

The evolving algebras $\mathcal{R}_{\text{ea}}$and $\mathcal{C}_{\text{ea}}$, our "official" representations of $\mathcal{R}_{\text{pcsp}}$ and $\mathcal{C}_{\text{pcsp}}$, are given in subsections 3.3 and 3.4; see Figures 9 and 10. The reader may proceed there directly and ignore the preceding subsections where we do the following. We first present in subsection 3.1 an elaborate ealgebra R1 that formalizes $\mathcal{R}_{\text{pcsp}}$ together with its environment; R1 expresses our understanding of how $\mathcal{R}_{\text{pcsp}}$ works, how it communicates with the environment and what the environment is supposed to do. Notice that the environment and the synchronization magic of CSP are explicit in R1. In subsection 3.2, we then transform R1 into another ealgebra R2 that performs synchronization implicitly. We transform R2 into $\mathcal{R}_{\text{ea}}$ by parallelizing the rules slightly and making the environment implicit; the result is shown in subsection 3.3. (In a sense, R1, R2, and $\mathcal{R}_{\text{ea}}$ are all equivalent to another another, but we will not formalize this.) We performed a similar analysis and transformation to create $\mathcal{C}_{\text{ea}}$ from $\mathcal{C}_{\text{pcsp}}$; we omit the intermediate stages and present $\mathcal{C}_{\text{ea}}$ directly in subsection 3.4.

3.1 R1: The First of the Row Evolving Algebras

The program for R1, given in Figure 7, contains six modules. The names of the modules reflect the intended meanings. In particular, modules BuffFrontEnd and BuffBackEnd correspond to the two processes Receiver and Sender of $\mathcal{R}_{\text{pcsp}}$.

Comment for ealgebraists. In terms of [6], the InputChannel agent is a two-member team comprising the InputEnvironment and the BuffFrontEnd agents; functions Sender and Receiver are similar to functions Member_1 and Member_2. Similarly the OutputChannel agent is a team. This case is very simple and one can get rid of unary functions Sender and Receiver by introducing names for the sending and receiving agents.

Comment for CSP experts. Synchronization is implicit in CSP. It is a built-in magic of CSP. We have doers of synchronization. (In this connection, the reader may want to see the EA treatment of Occam in [2].) Nevertheless, synchronization remains abstract. In a sense the abstraction level is even higher: similar agents can synchronize more than two processes.

Comment. The nondeterministic formalizations of the input and output environments are abstract and may be refined in many ways.

Initial states. In addition to the function names mentioned in the program (and the logic names), the vocabulary of R1 contains universe names Data, Integers, $\mathcal{Z}_N$, $\mathcal{Z}_2$, Modes and a subuniverse Senders-and-Receivers of Agents. Initial states of R1 satisfy the following requirements.

(i) The universe Integers and the arithmetical function names mentioned in the program have their usual meanings. The universe $\mathcal{Z}_N$ consists of integers modulo N identified with the integers $0, \ldots, N-1$. The universe

```
Module InputEnvironment
  **if** Mode(Me) = Work **then**
      **choose** $v$ **in** Data
          InputDatum := $v$
      **endchoose**
      Mode(Me) := Ready
  **endif**
```

```
Module OutputEnvironment
  **if** Mode(Me) = Work **then** Mode(Me) := Ready **endif**
```

```
Module InputChannel
  **if** Mode(Sender(Me)) = Ready **and** Mode(Receiver(Me)) = Ready **then**
      Buffer($p$ mod $N$) := InputDatum
      Mode(Sender(Me)) := Work
      Mode(Receiver(Me)) := Work
  **endif**
```

```
Module OutputChannel
  **if** Mode(Sender(Me)) = Ready **and** Mode(Receiver(Me)) = Ready **then**
      OutputDatum := Buffer($g$ mod $N$)
      Mode(Sender(Me)) := Work
      Mode(Receiver(Me)) := Work
  **endif**
```

```
Module BuffFrontEnd
Rule FrontWait
  **if** Mode(Me) = Wait **and** $p - g \neq N$ **then** Mode(Me) := Ready **endif**
Rule FrontWork
  **if** Mode(Me) = Work **then** $p := p + 1$, Mode(Me) := Wait **endif**
```

```
Module BuffBackEnd
Rule BackWait
  **if** Mode(Me) = Wait **and** $p - g \neq 0$ **then** Mode(Me) := Ready **endif**
Rule BackWork
  **if** Mode(Me) = Work **then** $g := g + 1$, Mode(Me) := Wait **endif**
```

Fig. 7. The program for R1.

$\mathcal{Z}_2$ is similar. $p = g = 0$. Buffer is of type $\mathcal{Z}_N \rightarrow$ Data; InputDatum and OutputDatum take values in Data.

(ii) The universe Agents contains six elements to which Mod assigns different module names. We could have special nullary functions to name the six agents but we don't; we will call them with respect to their programs:

the input environment, the output environment, the input channel, the output channel, buffer's front end and buffer's back end respectively. Sender(the input channel) = the input environment, Receiver(the input channel) = buffer's front end, Sender(the output channel) = buffer's back end, and Receiver(the output channel) = the output environment. The universe Senders-and-Receivers consists of the two buffer agents and the two environment agents. Nullary functions Ready, Wait and Work are distinct elements of the universe Modes. The function Mode is defined only over Senders-and-Receivers. For the sake of simplicity of exposition, we assign particular initial values to Mode: it assigns Wait to either buffer agent, Work to the input environment agent, and Ready to the output environment agent.

Analysis In the rest of this subsection, we prove that R1 has the intended properties.

Lemma 1 (Typing Lemma for R1) *In every state of any run of R1, the dynamic functions have the following (intended) types.*

(i) Mode: Senders-and-Receivers → Modes.
(ii) InputDatum, OutputDatum: Data.
(iii) p, g: Integers.
(iv) Buffer: $\mathcal{Z}_N$ → Data.

Proof. By induction over states. □

Lemma 2 (The p and g Lemma for R1) *Let ρ be an arbitrary run of R1. In every state of ρ, $0 \leq p-g \leq N$. Furthermore, if $p-g=0$ then Mode(buffer's back end) = Wait, and if $p-g=N$ then Mode(buffer's front end) = Wait.*

Proof. An obvious induction. See Lemma 11 in this regard. □

Lemma 3 (Ordering Lemma for R1) *In any run of R1, we have the following.*

(i) If μ is a move of the input channel and ν is a move of buffer's front end then either $\mu < \nu$ or $\nu < \mu$.
(ii) If μ is a move of the output channel and ν is a move of buffer's back end then either $\mu < \nu$ or $\nu < \mu$.
(iii) For any buffer slot k, if μ is a move of the input channel involving slot k and ν is a move of the output channel involving slot k then either $\mu < \nu$ or $\nu < \mu$.

Proof. Let $\rho = (M, A, \sigma)$ be a run of R1.

(i) Suppose by contradiction that μ and ν are incomparable and let $X = \{\pi : \pi < \mu \vee \pi < \nu\}$ so that, by the coherence requirements on the run, both agents are enabled at $\sigma(X)$, which is impossible because their guards are contradictory.

Since the input channel is enabled, the mode of buffer's front end is Ready at X. But then buffer's front end is disabled at X, which gives the desired contradiction.

(ii) Similar to part (i).

(iii) Suppose by contradiction that μ and ν are incomparable and let $X = \{\pi : \pi < \mu \vee \pi < \nu\}$ so that both agents are enabled at $\sigma(X)$. Since μ involves k, $p = k \bmod N$ in $\sigma(X)$. Similarly, $g = k \bmod N$ in $\sigma(X)$. Hence $p - g = 0 \bmod N$ in $\sigma(X)$. By the p and g lemma, either $p - g = 0$ or $p - g = N$ in $\sigma(X)$. In the first case, the mode of buffer's back end is Wait and therefore the output channel is disabled. In the second case, the mode of buffer's front end is Wait and therefore the input channel is disabled. In either case, we have a contradiction. $\square$

Recall that the state of move μ is $\Lambda(\mu) = \sigma(\{\nu : \nu < \mu\})$. By the coherence requirement, the agent $A(\mu)$ is enabled in $\Lambda(\mu)$.

Consider a run of R1. Let μ_i (respectively, ν_i) be the ith move of the input channel (respectively, the output channel). The value a_i of InputDatum in $\Lambda(\mu_i)$ (that, is the datum to be transmitted during μ_i) is the *ith input datum*, and the sequence $a_0, a_1, \ldots$ is the *input data sequence.* (It is convenient to start counting from 0 rather than 1.) Similarly, the value b_j of OutputDatum in $\Lambda(\nu_j)$ is the *jth output datum of R* and the sequence $b_0, b_1, \ldots$ is the *output data sequence.*

Lamport writes:

> *To make the example more interesting, we assume no liveness properties for sending values on the* in *channel, but we require that every value received in the buffer be eventually sent on the* out *channel.*

With this in mind, we call a run *regular* if the output sequence is exactly as long as the input sequence.

Theorem 4 *For a regular run, the output sequence is identical with the input sequence.*

Proof. Let $\mu_0, \mu_1, \ldots$ be the moves of the input channel and $\nu_0, \nu_1, \ldots$ be the moves of the output channel. A simple induction shows that μ_i stores the ith input datum a_i at slot $i \bmod N$ and $p = i$ at $\Lambda(\mu_i)$. Similarly, ν_j sends out the jth output datum b_j from slot $j \bmod N$ and $g = j$ at $\Lambda(\nu_j)$. If $\mu_i < \nu_i < \mu_{i+N}$, then $a_i = b_i$. We show that, for all i, $\mu_i < \nu_i < \mu_{i+N}$.

By the p and g lemma, $p - g > 0$ in $\Lambda(\nu_j)$ for any j, and $p - g < N$ in $\Lambda(\mu_j)$ for any j.

(i) Suppose $\nu_i < \mu_i$. Taking into account the monotonicity of p, we have the following at $\Lambda(\nu_i)$: $p \leq i$, $g = i$ and therefore $p - g \leq 0$ which is impossible.

(ii) Suppose $\mu_{i+N} < \nu_i$. Taking into account the monotonicity of g, we have the following at $\Lambda(\mu_{i+N})$: $p = i + N$, $g \leq i$, and therefore $p - g \geq N$ which is impossible.

By the ordering lemma, ν_i is order-comparable with both μ_i and μ_{i+N}. It follows that $\mu_i < \nu_i < \mu_{i+N}$. $\square$

3.2 R2: The Second of the Row Evolving Algebras

One obvious difference between $\mathcal{R}_{\text{pcsp}}$ and R1 is the following: R1 explicitly manages the communication channels between the buffer and the environment, while $\mathcal{R}_{\text{pcsp}}$ does not. By playing with the modes of senders and receivers, the channel modules of R1 provide explicit synchronization between the environment and the buffers. This synchronization is implicit in the "?" and "!" operators of CSP. To remedy this, we transform R1 into an ealgebra R2 in which communication occurs implicitly. R2 must somehow ensure synchronization. There are several options.

(i) Allow BuffFrontEnd (respectively, BuffBackEnd) to modify the mode of the input environment (respectively, the output environment) to ensure synchronization.

    This approach is feasible but undesirable. It is unfair; the buffer acts as a receiver on the input channel and a sender on the output channel but exerts complete control over the actions of both channels. Imagine that the output environment represents another buffer, which operates as our buffer does; in such a case both agents would try to exert complete control over the common channel.

(ii) Assume that BuffFrontEnd (respectively, BuffBackEnd) does not execute until the input environment (respectively, the output environment) is ready.

    This semantical approach reflects the synchronization magic of CSP. It is quite feasible. Moreover, it is common in the EA literature to make assumptions about the environment when necessary. It is not necessary in this case because there are very easy programming solutions (see the next two items) to the problem.

(iii) Use an additional bit for either channel which tells us whether the channel is ready for communication or not.

In fact, a state of a channel comprises a datum and an additional bit in the TLA part of Lamport's paper. One can avoid dealing with states of the channel by requiring that each sender and receiver across a channel maintains its own bit (a well-known trick) which brings us to the following option.

(iv) Use a bookkeeping bit for every sender and every receiver.

It does not really matter, technically speaking, which of the four routes is chosen. To an extent, the choice is a matter of taste. We choose the fourth approach. The resulting ealgebra R2 is shown in Figure 8.

Notice that the sender can place data into a channel only when the synchronization bits match, and the receiver can read the data in a channel only when the synchronization bits do not match.

The initial states of R2 satisfy the first condition on the initial states of R1. The universe Agents contains four elements to which Mod assigns different module names; we will call them with respect to their programs: the input environment, the output environment, buffer's front end, and buffer's back end, respectively. The universe BufferAgents contains the buffer's front end and buffer's back end agents. Nullary functions InSendBit, InReceiveBit, OutSendBit, OutReceiveBit are all equal to 0. Nullary functions Ready, Wait and Work are distinct elements of the universe Modes. The function Mode is defined only over BufferAgents; it assigns Wait to each buffer agent. InputDatum and OutputDatum take values in Data. Define the input and output sequences and regular runs as in R1.

Let Υ_1 be the vocabulary of R1 and Υ_2 be the vocabulary of R2.

Lemma 5 *Every run $R = (M, A, \sigma)$ of R1 induces a run $\rho = (M, B, \tau)$ of R2 where:*

(i) If $\mu \in M$ and $A(\mu)$ is not a channel agent, then $B(\mu) = A(\mu)$. If $A(\mu) =$ the input channel, then $B(\mu) =$ buffer's front end. If $A(\mu) =$ the output channel, then $B(\mu) =$ buffer's back end.

(ii) Let X be a finite initial segment of M. $\tau(X)$ is the unique state satisfying the following conditions:

(a) $\tau(X)|(\Upsilon_1 \cap \Upsilon_2) = \sigma(X)|(\Upsilon_1 \cap \Upsilon_2)$

(b) InReceiveBit $= p \bmod 2$ if the mode of buffer's front end is Wait or Ready, and $1 - p \bmod 2$ otherwise.

(c) OutSendBit $= g \bmod 2$ if the mode of buffer's back end is Wait or Ready, and $1 - g \bmod 2$ otherwise.

```
Module InputEnvironment
  if InSendBit = InReceiveBit Then
      choose $v$ in Data
          InputDatum := $v$
      endchoose
      InSendBit := 1 – InSendBit
  endif
```

```
Module OutputEnvironment
  if OutSendBit $\neq$ OutReceiveBit then
      OutReceiveBit := 1 – OutReceiveBit
  endif
```

```
Module BuffFrontEnd
Rule FrontWait
  if Mode(Me) = Wait and $p - g \neq N$ then Mode(Me) := Ready endif

Rule FrontCommunicate
  if Mode(Me) = Ready and InSendBit $\neq$ InReceiveBit then
      Buffer($p$ mod $N$) := InputDatum
      Mode(Me) := Work
      InReceiveBit := 1 – InReceiveBit
  endif

Rule FrontWork
  if Mode(Me) = Work then $p := p + 1$, Mode(Me) := Wait endif
```

```
Module BuffBackEnd
Rule BackWait
  if Mode(Me) = Wait and $p - g \neq 0$ then Mode(Me) := Ready endif

Rule BackCommunicate
  if Mode(Me) = Ready and OutSendBit = OutReceiveBit then
      OutputDatum := Buffer($g$ mod $N$)
      Mode(Me) := Work
      OutSendBit := 1 – OutSendBit
  endif

Rule BackWork
  if Mode(Me) = Work then $g := g + 1$, Mode(Me) := Wait endif
```

Fig. 8. The program for R2.

(d) *InSendBit* = *InReceiveBit if the mode of the input environment is Work, and* $1-$ *InReceiveBit otherwise.*
(e) *OutReceiveBit* = *OutSendBit if the mode of the output environment is Ready, and* $1-$ *OutSendBit otherwise.*

Proof. We check that ρ is indeed a run of R2. By the ordering lemma for R1, the moves of every agent of R2 are linearly ordered. It remains to check only the coherence condition; the other conditions are obvious. Suppose that Y is a finite initial segment of N with a maximal element μ and $X = Y - \{\mu\}$. Using the facts that $A(\mu)$ is enabled in $\sigma(X)$ and $\sigma(Y)$ is the result of executing $A(\mu)$ in $\sigma(X)$, it is easy to check that $B(\mu)$ is enabled in $\tau(X)$ and $\tau(Y)$ is the result of executing $B(\mu)$ at $\tau(X)$. $\square$

Lemma 6 *Conversely, every run of R2 is induced (in the sense of the preceding lemma) by a unique run of R1.*

The proof is easy and we skip it.

3.3 $\mathcal{R}_{\text{ea}}$*: The Official Row Evolving Algebra*

After establishing that $p - g \neq N$ and before executing the FrontCommunicate rule, buffer's front end goes to mode Ready. This corresponds to nothing in $\mathcal{R}_{\text{pcsp}}$ which calls for merging the FrontWait and FrontCommunicate rules. On the other hand, $\mathcal{R}_{\text{pcsp}}$ augments p *after* performing an act of communication. There is no logical necessity to delay the augmentation of p. For aesthetic reasons we merge the FrontWork rule with the other two rules of BuffFrontEnd. Then we do a similar parallelization for BuffBackEnd. Finally we simplify the names BuffFrontEnd and BuffBackEnd to FrontEnd and BackEnd respectively.

A certain disaccord still remains because the environment is implicit in $\mathcal{R}_{\text{pcsp}}$. To remedy this, we remove the environment modules, asserting that the functions InputDatum, InSendBit, and OutReceiveBit which were updated by the environment modules are now external functions. The result is our official ealgebra $\mathcal{R}_{\text{ea}}$, shown in Figure 9.

The initial states of $\mathcal{R}_{\text{ea}}$ satisfy the first condition on the initial states of R1: The universe Integers and the arithmetical function names mentioned in the program have their usual meanings; the universe $\mathcal{Z}_N$ consists of integers modulo N identified with the integers $0, \ldots, N-1$; the universe $\mathcal{Z}_2$ is similar; $p = g = 0$; Buffer is of type $\mathcal{Z}_N \rightarrow$ Data; InputDatum and OutputDatum take values in Data.

```
Module FrontEnd
  if $p - g \neq N$ and InSendBit $\neq$ InReceiveBit then
      Buffer($p$ mod $N$) := InputDatum
      InReceiveBit := 1 - InReceiveBit
      $p := p + 1$
  endif
```

```
Module BackEnd
  if $p - g \neq 0$ and OutSendBit = OutReceiveBit then
      OutputDatum := Buffer($g$ mod $N$)
      OutSendBit := 1 - OutSendBit
      $g := g + 1$
  endif
```

Fig. 9. The program for $\mathcal{R}_{\text{ea}}$.

Additionally, the universe Agents contains two elements to which Mod assigns different module names. InSendBit, InReceiveBit, OutSendBit, and OutReceiveBit are all equal to 0. InputDatum and OutputDatum take values in Data.

The definition of regular runs of $\mathcal{R}_{\text{ea}}$ is slightly more complicated, due to the presence of the external functions InputDatum, InSendBit, and OutReceiveBit. We require that the output sequence is at least as long as the input sequence, InputDatum is of type Data, and InSendBit and OutReceiveBit are both of type $\mathcal{Z}_2$.

We skip the proof that $\mathcal{R}_{\text{ea}}$ is faithful to R2.

3.4 $\mathcal{C}_{\text{ea}}$: The Official Column Evolving Algebra

The evolving algebra $\mathcal{C}_{\text{ea}}$ is shown in figure 10 below. It can be obtained from $\mathcal{C}_{\text{pcsp}}$ in the same way that $\mathcal{R}_{\text{ea}}$ can be obtained from $\mathcal{R}_{\text{pcsp}}$; for brevity, we omit the intermediate stages.

Initial states The initial states of $\mathcal{C}_{\text{ea}}$ satisfy the following conditions.

(i) The first condition for the initial states of R1 is satisfied except we don't have functions p and g now. Instead we have dynamic functions pp and gg with domain $\mathcal{Z}_N$ and $pp(i) = gg(i) = 0$ for all i in $\mathcal{Z}_N$.
(ii) The universe Agents consists of the elements of $\mathcal{Z}_N$, which are mapped by Mod to the module name Slot. Nullary functions Get and Put are

```
Module Slot

Rule Get
  if Mode(Me)=Get and InputTurn(Me)
          and InSendBit $\neq$ InReceiveBit then
      Buffer(Me) := InputDatum
      InReceiveBit := 1 - InReceiveBit
      $pp$(Me) := $1 - pp$(Me)
      Mode(Me) := Put
  endif

Rule Put
  if Mode(Me)=Put and OutputTurn(Me)
          and OutSendBit = OutReceiveBit then
      OutputDatum := Buffer(Me)
      OutSendBit := 1 - OutSendBit
      $gg$(Me) := $1 - gg$(Me)
      Mode(Me) := Get
  endif

InputTurn(x) abbreviates
  $[x = 0$ and $pp(0) = pp(N-1)]$ or $[x \neq 0$ and $pp(x) \neq pp(x-1)]$
OutputTurn(x) abbreviates
  $[x = 0$ and $gg(0) = gg(N-1)]$ or $[x \neq 0$ and $gg(x) \neq gg(x-1)]$
```

Fig. 10. The program for $\mathcal{C}_{\text{ea}}$.

distinct elements of the universe Modes. The dynamic function Mode is defined over Agents; Mode(x)=Get for every x in $\mathcal{Z}_N$. InputDatum and OutputDatum are elements of Data. Nullary functions InSendBit, InReceiveBit, OutSendBit, OutReceiveBit are all equal to 0.

Regular runs are defined similarly to $\mathcal{R}_{\text{ea}}$; we require that the output sequence is at least as long as the input sequence, InputDatum is of type Data, and InSendBit and OutReceiveBit take values in $\mathcal{Z}_2$.

4 Equivalence

We define a strong version of lock-step equivalence for ealgebras which for brevity we call *lock-step equivalence.* We then prove that $\mathcal{R}_{\text{ea}}$ and $\mathcal{C}_{\text{ea}}$ are lock-step equivalent. We start with an even stronger version of lock-step equivalence which we call *strict lock-step equivalence.*

For simplicity, we restrict attention to ealgebras with a fixed superuniverse. In other words, we suppose that all initial states have the same superuniverse. This assumption does not reduce generality because the superuniverse can be always chosen to be sufficiently large.

4.1 Strict Lock-Step Equivalence

Let $\mathcal{A}$ and $\mathcal{B}$ be ealgebras with the same superuniverse and suppose that h is a one-to-one mapping from the states of $\mathcal{A}$ onto the states of $\mathcal{B}$ such that if $h(a) = b$ then a and b have identical interpretations of the function names common to $\mathcal{A}$ and $\mathcal{B}$. Call a run (M, A, σ) of $\mathcal{A}$ *strictly h-similar* to a partially ordered run (N, B, τ) of $\mathcal{B}$ if there is an isomorphism $\eta : M \to N$ such that for every finite initial segment X of M, $h(\sigma(X)) = \tau(Y)$, where $Y = \{\eta(\mu) : \mu \in X\}$. Call $\mathcal{A}$ and $\mathcal{B}$ *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 $\mathcal{A}$. Finally call $\mathcal{A}$ and $\mathcal{B}$ *strictly lock-step equivalent* if there exists an h such that they are strictly h-similar.

Ideally we would like to prove that $\mathcal{R}_{\text{ea}}$ and $\mathcal{C}_{\text{ea}}$ are strictly lock-step equivalent. Unfortunately this is false, which is especially easy to see if the universe Data is finite. In this case, any run of $\mathcal{C}_{\text{ea}}$ has only finitely many different states; this is not true for $\mathcal{R}_{\text{ea}}$ because p and g may take arbitrarily large integer values. One can rewrite either $\mathcal{R}_{\text{ea}}$ or $\mathcal{C}_{\text{ea}}$ to make them strictly lock-step equivalent. For example, $\mathcal{C}_{\text{ea}}$ can be modified to perform math on pp and gg over Integers instead of $\mathcal{Z}_2$. We will not change either ealgebra; instead we will slightly weaken the notion of strict lock-step equivalence.

4.2 Lock-Step Equivalence

If an agent α of an ealgebra $\mathcal{A}$ is enabled at a state a, let $\text{Result}(\alpha, a)$ be the result of firing α at a; otherwise let $\text{Result}(\alpha, a) = a$.

Say that an equivalence relation $\cong$ on the states of $\mathcal{A}$ *respects* a function name f of $\mathcal{A}$ if f has the same interpretation in equivalent states. The equivalence classes of a will be denoted $[a]$ and called the *configuration* of a. Call $\cong$ a *congruence* if $a_1 \cong a_2 \to \text{Result}(\alpha, a_1) \cong \text{Result}(\alpha, a_2)$ for any states a_1, a_2 and any agent α.

Let $\mathcal{A}$ and $\mathcal{B}$ be ealgebras with the same superuniverse and congruences $\cong_{\mathcal{A}}$ and $\cong_{\mathcal{B}}$ respectively. (We will drop the subscripts on $\cong$ when no confusion arises.) We suppose that either congruence respects the function names common to $\mathcal{A}$ and $\mathcal{B}$. Further, let h be a one-to-one mapping of $\cong_{\mathcal{A}}$-configurations onto $\cong_{\mathcal{B}}$-configurations such that, for every function name f common to $\mathcal{A}$ and $\mathcal{B}$, if $h([a]) = [b]$, then $f_a = f_b$.

Call a partially ordered run (M, A, σ) of $\mathcal{A}$ *h-similar* to a partially ordered run (N, B, τ) of $\mathcal{B}$ if there is an isomorphism $\eta : M \to N$ such that, for every finite initial segment X of M, $h([\sigma(X)]) = [\tau(Y)]$, where $Y = \{\eta(\mu) : \mu \in X\}$. Call $\mathcal{A}$ and $\mathcal{B}$ *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 $\mathcal{A}$. Call $\mathcal{A}$ and $\mathcal{B}$ *lock-step equivalent* (with respect to $\cong_{\mathcal{A}}$ and $\cong_{\mathcal{B}}$) if there exists an h such that $\mathcal{A}$ and $\mathcal{B}$ are h-similar.

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.

Assuming that $\mathcal{R}_{\text{ea}}$ and $\mathcal{C}_{\text{ea}}$ have the same superuniverse, we will show that $\mathcal{R}_{\text{ea}}$ is lock-step equivalent to $\mathcal{C}_{\text{ea}}$ with respect to the congruences defined below.

Remark. The assumption that $\mathcal{R}_{\text{ea}}$ and $\mathcal{C}_{\text{ea}}$ have the same superuniverse means essentially that the superuniverse of $\mathcal{C}_{\text{ea}}$ contains all integers even though most of them are not needed. It is possible to remove the assumption. This leads to slight modifications in the proof. One cannot require that a common function name f has literally the same interpretation in a state of $\mathcal{R}_{\text{ea}}$ and a state of $\mathcal{C}_{\text{ea}}$. Instead require that the interpretations are essentially the same. For example, if f is a predicate, require that the set of tuples where f is true is the same.

Definition 7 *For states c, d of $\mathcal{C}_{\text{ea}}$, $c \cong d$ if $c = d$.*

Since each configuration of $\mathcal{C}_{\text{ea}}$ has only one element, we identify a state of $\mathcal{C}_{\text{ea}}$ with its configuration. Let e_a denote the value of an expression e at a state a.

Definition 8 *For states a, b of $\mathcal{R}_{\text{ea}}$, $a \cong b$ if:*

- $g_a = g_b \mod 2N$
- $(p - g)_a = (p - g)_b$
- $f_a = f_b$ *for all other function names f.*

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

Lemma 9 *If $a \cong_{\mathcal{R}} b$ then we have the following modulo 2:*

- $p_a \operatorname{div} N = p_b \operatorname{div} N$
- $g_a \operatorname{div} N = g_b \operatorname{div} N$

Proof. We prove the desired property for p; the proof for g is similar.

By the definition of $\cong_{\mathcal{R}}$, we have the following modulo $2N$: $p_a = g_a + (p-g)_a = g_b + (p-g)_b = p_b$. Thus, there are non-negative integers x_1, x_2, x_3, y such that $p_a = 2Nx_1 + Nx_2 + x_3$, $p_b = 2Ny + Nx_2 + x_3$, $x_2 \leq 1$, and $x_3 < N$. Hence $p_a \operatorname{div} N = 2x_1 + x_2$ and $p_b \operatorname{div} N = 2y + x_2$, which are equal modulo 2. $\square$

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

Definition 10 *If a is a state of $\mathcal{R}_{\text{ea}}$, then $h([a])$ is the state c of $\mathcal{C}_{\text{ea}}$ such that*

$$pp(i)_c = \begin{cases} p_a \operatorname{div} N \bmod 2 & \text{if } i \geq p_a \bmod N \\ 1 - (p_a \operatorname{div} N) \bmod 2 & \text{otherwise} \end{cases}$$

$$gg(i)_c = \begin{cases} g_a \operatorname{div} N \bmod 2 & \text{if } i \geq g_a \bmod N \\ 1 - (g_a \operatorname{div} N) \bmod 2 & \text{otherwise} \end{cases}$$

and for all common function names f, $f_c = f_a$.

Thus, h relates the counters p, g used in $\mathcal{R}_{\text{ea}}$ and the counters pp, gg used in $\mathcal{C}_{\text{ea}}$. (Notice that by Lemma 9, h is well-defined.) We have not said anything about *Mode* because *Mode* is uniquely defined by the rest of the state (see Lemma 16 in section 4.3) and is redundant.

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

4.3 Properties of $\mathcal{R}_{\text{ea}}$

We say that a is a state of a run (M, A, σ) if $a = \sigma(X)$ for some finite initial segment X of M.

Lemma 11 *For any state b of any run of $\mathcal{R}_{\text{ea}}$, $0 \leq (p-g)_b \leq N$.*

Proof. By induction. Initially, $p = g = 0$.

Let (M, A, σ) be a run of $\mathcal{R}_{\text{ea}}$. Let X be a finite initial segment of M with maximal element μ, such that $0 \leq p - g \leq N$ holds in $a = \sigma(X - \{\mu\})$. Let $b = \sigma(X)$.

- If $A(\mu)$ is the front end agent and is enabled in a, then $0 \leq (p-g)_a < N$. The front end agent increments p but does not alter g; thus, $0 < (p-g)_b \leq N$.

– If $A(\mu)$ is the back end agent and is enabled in a, then $0 < (p-g)_a \leq N$. The back end agent increments g but does not alter p; thus, $0 \leq (p-g)_b < N$. □

Lemma 12 *Fix a non-negative integer $k < N$. For any run (M, A, σ) of $\mathcal{R}_{\mathrm{ea}}$, the k-slot moves of M (that is, the moves of M which involve Buffer(k)) are linearly ordered.*

Proof. Similar to Lemma 3. □

4.4 Properties of $\mathcal{C}_{\mathrm{ea}}$

Lemma 13 *For any run of $\mathcal{C}_{\mathrm{ea}}$, there is a mapping In from states of $\mathcal{C}_{\mathrm{ea}}$ to $\mathcal{Z}_N$ such that if $In(c) = k$, then:*

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

Proof. By induction. Initially, agent 0 (and no other) satisfies *InputTurn(Me)* and $pp(i) = 0$ holds for every agent i. Thus, if c is an initial state, $In(c) = 0$.

Let (M, A, σ) be a run of $\mathcal{C}_{\mathrm{ea}}$. Let Y be a finite initial segment of M with maximal element μ, such that the requirements hold in $c = \sigma(Y - \{\mu\})$. Let $d = \sigma(Y)$.

If $A(\mu)$ executes rule Put, *pp* is not modified and $In(d) = In(c)$. Otherwise, if rule Get is enabled for $A(\mu)$, executing rule Get increments *pp*; the desired $In(d) = In(c) + 1 \bmod N$. This is obvious if $In(c) < N - 1$. If $In(c) = N - 1$, then all values of *pp* are equal in d and $In(d) = 0$ satisfies the requirements. □

Lemma 14 *For any run of $\mathcal{C}_{\mathrm{ea}}$, there is a mapping Out from states of $\mathcal{C}_{\mathrm{ea}}$ to $\mathcal{Z}_N$ such that if $Out(c) = k$, then:*

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

Proof. Parallel to that of the last lemma. □

It is easy to see that every move μ of $\mathcal{C}_{\mathrm{ea}}$ involves an execution of rule Get or rule Put but not both. (More precisely, consider finite initial segments Y

of moves where μ is a maximal element of Y. Any such Y is obtained from $Y - \{\mu\}$ either by executing Get in state $\sigma(Y - \{\mu\})$, or executing Put in state $\sigma(Y - \{\mu\})$.) In the first case, call μ a Get move. In the second case, call μ a Put move.

Lemma 15 *In any run (M, A, σ) of $\mathcal{C}_{\mathrm{ea}}$, all Get moves are linearly ordered and all Put moves are linearly ordered.*

Proof. We prove the claim for rule Get; the proof for rule Put is similar. By contradiction, suppose that are two incomparable Get moves μ and ν. By the coherence condition for runs, both rules are enabled in state $X = \{\pi : \pi < \mu \vee \pi < \nu\}$. By Lemma 13, $\mathrm{A}(\mu) = \mathrm{A}(\nu)$. But all moves of the same agent are ordered; this gives the desired contradiction. $\square$

Lemma 16 *In any state d of any run of $\mathcal{C}_{\mathrm{ea}}$, for any agent k,*

$$Mode(k)_d = \begin{cases} Get & \text{if } pp(k)_d = gg(k)_d \\ Put & \text{if } pp(k)_d = 1 - gg(k)_d \end{cases}$$

Proof. We fix a k and do induction over runs. Initially, $Mode(k) = Get$ and $pp(k) = gg(k) = 0$ for every agent k.

Let Y be a finite initial segment of a run with maximal element μ such that (by the induction hypothesis) the required condition holds in $c = \sigma(Y - \{\mu\})$. Let $d = \sigma(Y)$.

If $A(\mu) \neq k$, none of $Mode(k)$, $pp(k)$, and $gg(k)$ are affected by executing $A(\mu)$ in c, so the condition holds in d. If $A(\mu) = k$, we have two cases.

- If agent k executes rule Get in state c, we must have $Mode(k)_c = Get$ (from rule Get) and $pp(k)_c = gg(k)_c$ (by the induction hypothesis). Firing rule Get yields $Mode(k)_d = Put$ and $pp(k)_d = 1 - pp(k)_c = 1 - gg(k)_d$.
- If agent k executes rule Put in state c, we must have $Mode(k)_c = Put$ (from rule Put) and $pp(k)_c = 1 - gg(k)_c$ (by the induction hypothesis). Firing rule Get yields $Mode(k)_d = Get$ and $gg(k)_d = 1 - gg(k)_c = pp(k)_d$. $\square$

Remark. This lemma shows that function *Mode* is indeed redundant.

4.5 Proof of Equivalence

Lemma 17 *If $h([a]) = c$, then $In(c) = p_a \bmod N$ and $Out(c) = g_a \bmod N$.*

Proof. Recall that *In(c)* is the agent k for which *InputTurn(k)*$_c$ holds. Lemma 13 asserts that $pp(i)_c$ has one value for $i < k$ and another for $i \geq k$. By the definition of h, this "switch-point" in pp occurs at $p_a \bmod N$. The proof for $Out(c)$ is similar. □

Lemma 18 *Module FrontEnd is enabled in state a of* $\mathcal{R}_{\text{ea}}$ *iff rule Get is enabled in state* $c = h([a])$ *of* $\mathcal{C}_{\text{ea}}$ *for agent* $In(c)$*.*

Proof. Let $k = In(c)$, so that *InputTurn(k)*$_c$ holds. Both FrontEnd and Get have *InSendBit* $\neq$ *InReceiveBit* in their guards. It thus suffices to show that $(p - g)_a \neq N$ iff $Mode(k)_c =$ Get. By Lemma 16, it suffices to show that $(p - g)_a \neq N$ iff $pp(k)_c = gg(k)_c$.

Suppose $(p-g) \neq N$. There exist non-negative integers x_1, x_2, x_3, x_4 such that $p_a = x_1 N + x_3$, $g_a = x_2 N + x_4$, and $x_3, x_4 < N$. (Note that by Lemma 17, $k = p_a \bmod N = x_3$.)

By Lemma 11, $0 \leq (p - g)_a < N$. There are two cases.

- $x_1 = x_2$ and $x_3 \geq x_4$. By definition of h, we have that, modulo 2, $pp(x_3)_c = p_a \operatorname{div} N = x_1$ and for all $i \geq g_a \bmod N = x_4$, $gg(i)_c = g_a \operatorname{div} N = x_2$. Since $x_3 \geq x_4$, we have that, modulo 2, $gg(x_3)_c = x_2 = x_1 = pp(x_3)_c$, as desired.
- $x_1 = (x_2 + 1)$ and $x_3 < x_4$. By definition of h, we have that, modulo 2, $pp(x_3)_c = p_a \operatorname{div} N = x_1$ and for all $i < g_a \bmod N = x_4$, $gg(i)_c = 1$ - $g_a \operatorname{div} N = x_2 + 1$. Since $x_3 < x_4$, we have that, modulo 2, $gg(x_3)_c = x_2 + 1 = x_1 = pp(x_3)_c$, as desired.

On the other hand, suppose $(p - g)_a = N$. Then $p_a \operatorname{div} N$ and $g_a \operatorname{div} N$ differ by 1. By definition of h, $pp(i)_c = 1 - gg(i)_c$ for all i, including k. □

Lemma 19 *Module BackEnd is enabled in state a iff rule Put is enabled in state* $c = h([a])$ *for agent* $Out(c)$*.*

Proof. Similar to that of the last lemma. □

Lemma 20 *Suppose that module FrontEnd is enabled in a state a of* $\mathcal{R}_{\text{ea}}$ *for the front end agent I and rule Get is enabled in a state* $c = h([a])$ *of* $\mathcal{C}_{\text{ea}}$ *for agent* $In(c)$*. Let* $b = Result(I, a)$ *and* $d = Result(In(c), c)$*. Then* $d = h([b])$*.*

Proof. We check that $h([b]) = d$.

- Both agents execute *InReceiveBit := 1 – InReceiveBit*.

- The front end agent executes *Buffer(p mod N) := InputDatum*. Agent $In(c)$ executes *Buffer(In(c)) := InputDatum*. By Lemma 17, *In(c)* $= p_a \bmod N$, so these updates are identical.
- The front end agent executes $p := p+1$. Agent $In(c)$ executes $pp(In(c)) := 1 - pp(In(c))$. The definition of h and the fact that $pp(i)_c = pp(i)_{h([a])}$ for all $i \in \mathcal{Z}_N$ imply that $pp(i)_d = pp(i)_{h([b])}$.
- Agent $In(c)$ executes *Mode(In(c)) := Put*. By Lemma 16, this update is redundant and need not have a corresponding update by the front end agent. □

Lemma 21 *Suppose that module BackEnd is enabled in a state a of* $\mathcal{R}_{\text{ea}}$ *for the back end agent O and rule Put is enabled in a state* $c = h([a])$ *of* $\mathcal{C}_{\text{ea}}$ *for agent* $Out(c)$. *Let* $b = Result(O, a)$ *and* $d = Result(Out(c), c)$. *Then* $d = h([c])$.

Proof. Parallel to that of the last theorem. □

Theorem 22 $\mathcal{R}_{\text{ea}}$ *is lock-step equivalent to* $\mathcal{C}_{\text{ea}}$.

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 $\mathcal{R}_{\text{ea}}$ is h-similar to a run of $\mathcal{C}_{\text{ea}}$, using the definition of h given earlier. Construct a run (M, A', σ') of $\mathcal{C}_{\text{ea}}$, where $\sigma'(X) = h([\sigma(X)])$ and A' is defined as follows. Let μ be a move of M, $a = \Lambda(\mu)$, and $c = h([\Lambda(\mu)])$. Then $A'(\mu) = In(c)$ if $A(\mu)$ is the front end agent, and $A'(\mu) = Out(c)$ if $A(\mu)$ is the back end agent.

We check that (M, A', σ') satisfies the four requirements for a run of $\mathcal{C}_{\text{ea}}$ stated in Section 2.6.

(i) Trivial, since (M, A, σ) is a run.
(ii) By Lemma 12, 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 17, we have modulo N that $k = In(c) = p_a$ if $A(\mu)$ is the front end agent and $k = Out(c) = g_a$ if $A(\mu)$ is the back end agent. In either case, μ is a k-slot move.
(iii) Since $\sigma' = h \circ \sigma$, σ' maps finite initial segments of M to states of $\mathcal{C}_{\text{ea}}$.
(iv) *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)) = σ(Y)*. By Lemma 18 or 19, $A'(\mu)$ is enabled in $\sigma'(X)$. By Lemma 20 or 21, *Result(A′(μ), σ′(X))* $= \sigma'(Y)$.

Continuing, we must also show that for any run (M, A', σ') of $\mathcal{C}_{\text{ea}}$, there is a run (M, A, σ) of $\mathcal{R}_{\text{ea}}$ 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 $A(\mu)$ to be the front end agent. If $A'(\mu)$ executes rule Put, set $A(\mu)$ to be the back end agent.

We check that the moves of the front end agent are linearly ordered. By Lemma 15, it suffices to show that if $A(\mu)$ is the front end agent, then $A'(\mu)$ executes Get in state $\Lambda'(\mu)$ — which is true by construction of A. A similar argument shows that the moves of the back end agent are linearly ordered.

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 $\sigma(Y)^- = \mathit{Result}(A(\mu), \sigma(X))$. Is it possible to select such a $\sigma(Y)$? Yes. By Lemma 18 or 19, $A(\mu)$ is enabled in $\sigma(X)$ iff $A'(\mu)$ is enabled in $\sigma'(X)$. By Lemma 20 or 21, *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 $\mathcal{R}_{\text{ea}}$ which is h-similar to (M, A', σ'). $\Box$

5 Inequivalence

We have proven that our formalizations $\mathcal{R}_{\text{ea}}$ and $\mathcal{C}_{\text{ea}}$ of $\mathcal{R}_{\text{pcsp}}$ and $\mathcal{C}_{\text{pcsp}}$ are lock-step equivalent. Nevertheless, $\mathcal{R}_{\text{pcsp}}$ and $\mathcal{C}_{\text{pcsp}}$ are inequivalent in various other ways. In the following discussion we exhibit some of these inequivalences. The discussion is informal, but it is not difficult to prove these inequivalences using appropriate formalizations of $\mathcal{R}_{\text{pcsp}}$ and $\mathcal{C}_{\text{pcsp}}$. Let $\mathcal{R} = \mathcal{R}_{\text{pcsp}}$ and $\mathcal{C} = \mathcal{C}_{\text{pcsp}}$.

Magnitude of Values. $\mathcal{R}$ uses unrestricted integers as its counters; in contrast, $\mathcal{C}$ uses only single bits for the same purpose. We have already used this phenomenon to show that $\mathcal{R}_{\text{ea}}$ and $\mathcal{C}_{\text{ea}}$ are not strictly lock-step equivalent. One can put the same argument in a more practical way. Imagine that the universe Data is finite and small, and that a computer with limited memory is used 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 problem.

Types of Sharing. $\mathcal{R}$ shares access to the buffer between both processes; in contrast, each process in $\mathcal{C}$ has exclusive access to its portion of the buffer. Conversely, processes in $\mathcal{C}$ share access to both the input and output channels, while each process in $\mathcal{R}$ has exclusive access to one channel. Imagine an architecture in which processes pay in one way or another for acquiring a channel. $\mathcal{C}$ would be more expensive to use on such a system.

Degree of Sharing. How many internal locations used by each algorithm must be shared between processes? $\mathcal{R}$ shares access to $N+2$ locations: the N locations of the buffer and 2 counter variables. $\mathcal{C}$ shares access to $2N$ locations: the $2N$ counter variables. Sharing locations may not be without cost; some provision must be made for handling conflicts (*e.g.* read/write conflicts) at a given location. Imagine that a user must pay 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 contrasts can be made a little more dramatic. For example, one could construct another version of the ring buffer algorithm which uses $2N$ processes, each of which is responsible for an input or output action (but not both) to a particular buffer position. All of the locations it uses will be shared. It is lock-step equivalent to $\mathcal{R}$ and $\mathcal{C}$; yet, few people would choose to use this version because it exacerbates the disadvantages of $\mathcal{C}$. 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 have the disadvantage of not allowing all orderings of actions possible for $\mathcal{R}$ and $\mathcal{C}$.

Acknowledgement

We thank Søren Bøgh Lassen, Peter Mosses, and the anonymous referees for their comments.

References

[1] E. Börger, "Annotated Bibliography on Evolving Algebras", in *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995, 37–51.

[2] E. Börger and I. Đurđanović, "Correctness of compiling Occam to Transputer code." *Computer Journal*, vol. 39, no. 1, 1996, 52–92.

[3] E. Börger and D. Rosenzweig, "The WAM - definition and compiler correctness," In L.C. Beierle and L. Pluemer, eds., *Logic Programming: Formal Methods and Practical Applications*, North-Holland Series in Computer Science and Artificial Intelligence, 1994.

[4] 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.

[5] Y. Gurevich, "Evolving Algebras: An Attempt to Discover Semantics", *Current Trends in Theoretical Computer Science*, eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, 266–292. (First published in Bull. EATCS 57 (1991), 264–284; an updated version appears in [10].)

[6] Y. Gurevich, "Evolving Algebras 1993: Lipari Guide", in *Specification and Validation Methods*, ed. E. Börger, Oxford University Press, 1995, 9–36.

[7] Y. Gurevich, "Platonism, Constructivism, and Computer Proofs vs. Proofs by Hand", Bull. EATCS 57 (1995), 145–166.

[8] Y. Gurevich, and J. Huggins, "The Semantics of the C Programming Language," in Seected papers from *CSL'92* (Computer Science Logic), Springer Lecture Notes in Computer Science 702, 1993, 274-308.

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

[10] J. Huggins, ed., "Evolving Algebras Home Page", EECS Department, University of Michigan, `http://www.eecs.umich.edu/ealgebras/`.

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

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

[13] L. Lamport, "Processes are in the Eye of the Beholder." Research Report 132, Digital Equipment Corporation, Systems Research Center, December 1994.