# DISTRIBUTED COMPUTING *and* EDUCATION COLUMN
## *Special Issue*

BY

**JURAJ HROMKOVIC, STEFAN SCHMID**

ETH Zurich (Switzerland), University of Vienna (Austria)

In this joint distributed computing and educational column, Uri Abraham revisits the use of invariant-based proofs in teaching concurrency, comparing them to approaches using Tarskian system executions. Based on a simple example, he sketches the vision of a framework which allows to develop the details of an intuitive argument leading to a mathematically sound proof.

Enjoy!

# On Lamport's "Teaching Concurrency"

Uri Abraham
Departments of Mathematics and of Computer Sciences
Ben-Gurion University of the Negev
abraham@cs.bgu.ac.il

**Abstract**

In his article *Teaching Concurrency* Lamport stresses the importance of invariants for the education of engineers and computer students, and presents a short algorithm with a challenge to the reader: find an invariant with which a certain simple property of that distributed algorithm can be proved. Our aim is to compare the invariant proof approach to a different one which uses Tarskian system executions rather than invariants. By comparing the details of the two proofs for this simple algorithm we gain a better understanding of these approaches.

## 1 Introduction

In his article *Teaching Concurrency* [5] Leslie Lamport argues that learning about invariants is important not only for practical reasons but also because "When you use invariance, you are teaching not just an algorithm, but how to think about algorithms." To make his point, Lamport presents the following simple and yet interesting algorithm.

1. $x[i] := 1$;
2. $y[i] := x[(i-1) \mod N]$
3. end.

Figure 1: The Simple Concurrent Algorithm of Lamport. The protocol for process $p_i$, $0 \leq i < N$.

Here $N$ is the number of processes, $p_0, \ldots, p_{N-1}$. Process $p_i$ writes on register $x[i]$ and all other processes can read it. The registers are serial, which means that reading and writing actions of any register are assumed to be linearly ordered in time, and such that each read action of a register gets the value of the last write on

that register. The initial value of all registers $x[i]$ is assume to be 0. Process $p_i$ has $y[i]$ as a local variable.

In his article Lamport stats the following theorem:

**Theorem 1.1.** *After every process has stopped executing its simple protocol of Figure 1, at least one process $p_i$ has set $y[i] = 1$.*

The short argument that Lamport gives for this claim is very convincing. Consider the last process $p_i$ to execute line $1 : x[i] := 1$. Then, when $p_i$ reads its neighbor (executing line 2) it obtains the value 1 since by that time all other processes have already writen 1 on their $x[j]$ registers, and hence $p_i$ writes 1 on $y[i]$ which proves the claim.

This attractive argument is informal, and although for such a short and simple protocol there is no reason to object to this type of arguments, for larger programs it is necessary to provide a fuller guarantee that only mathematical proofs can provide. It is perhaps not immediately clear in what sense the previous argument is informal, and it may take some time to find that the argument implicitly uses some assumptions that need to be explicated if one wants to get a formal mathematical proof. For example that there is a last write is such an assumption. With larger programs one cannot be sure that an informal argument is not based on some invalid hidden assumption or that some case was missed in the argument. An advantage of the invariant based proof is that it leaves no hidden assumption in its proof. It is to this sort of advantage that Lamport refers to when he writes that "Invariance is the key to understanding concurrent systems,[. . . ]." Concerning his Simple algorithm and the correctness claim of Theorem 1.1 Lamport says "The algorithm satisfies this property because it maintains an inductive invariant. Do you know what that invariant is? If not, then you do not completely understand why the algorithm satisfies this property".

The invariant method is important in practice, in theory, and in teaching. But convincing arguments as the one given above for the Simple algorithm are also important because they are natural and are useful for guiding intuition and understanding. Our aim here is to describe the elements of a framework within which it is possible to develop the details of the intuitive argument in a way that makes it a mathematically sound proof.

In the following section we describe the well-established state and history approach to concurrency correctness and show an invariant for the Simple algorithm of Figure 1 by which its correctness is proved. A short section 2.1 points to some limitations of the invariant approach by showing how even small changes in the Simple algorithm may be quite a challenge to that approach, but require only minor modifications in the intuitive argument. In Section 3 Tarskian system executions are defined and it is shown (Section 4) how the correctness of the Simple algorithm of Lamport can be conducted in a mathematical way that follows and

reflects the intuitive argument. In this framework, the challenging modifications of Section 2.1 do not pose any problem.

# 2 States, actions, and histories

A state is a representation of the essential features of the system at some moment. Formally we have a collection, *SysVar*, of *system variables*, and for every $v \in SysVar$ we have a set $Type(v)$ which is the set of value that variable $v$ is expected to take. We then have the following definition. A state is a function $S$ defined over *SysVar* and such that, for every $v \in SysVar$, $S(v) \in Type(v)$. Let *States* be the collection of all states thus defined.

The notion of typeless states is also needed (but only for a short while): a typeless state is just a function defined over *SysVar* without any limitations on the possible values that a variable takes. So "*States*" refers to type-respecting functions, and typeless-states are non-restricted functions. The need for typeless states is clarified below.

In many cases, variables can be composed into *terms*. For example, if $m$ and $n$ are variables of type natural-number then the term $n + m$ can be formed and it aquires a value whenever $m$ and $n$ do. The value of variable $m$ in state $S$ is denoted $m^S$, and the value of $m + n$ is denoted $(m + n)^S$.

For the Simple algorithm of Figure 1 we have the following system variables.

$$SysVar = \{x[0], \ldots, x[N-1], y[0], \ldots, y[N-1], PC_0, \ldots, PC_{N-1}\},$$

where $PC_i$ is the program counter of process $p_i$. We have $Type(x[i]) = Type(y[i]) = \{0, 1\}$, and $Type(PC_i) = \{1, 2, 3\}$. The meaning of $PC_i = k$ for $k = 1, 2$, is that $p_i$'s next action will be an execution of the atomic instruction at line $k$. But the meaning of $PC_i = 3$ is that process $p_i$ stops and no step by $p_i$ is enabled.

There is one initial state here: $I_0$ has all program counters at 1 (ready to execute the first instruction) and all registers have value 0. So $I_0(PC_i) = 1$, and $I_0(x[i]) = 0$ for all $i$. The local variables $y[i]$ of $p_i$ are set to 0.

The protocol of Figure 1 contains just two sorts of instructions: write and read of registers. Generally, a write instruction has the form $R := t$ where $R$ is a register and $t$ is a term which can be evaluated in any (typed) state. A read instruction has the form $v := R$ where $R$ is a register and $v$ a local variable of the executing process.

A *step* is a description of an atomic change of the system executed by one of the processes. A step is represented by a pair of states $(S, T)$: $S$ is the state before the execution and $T$ is the resulting state. For the Simple algorithm considered here we have just two kinds of steps: read and write.

1. An execution of an instruction $R := t$ ($R$ a register and $t$ a term) by process $p_i$ is a pair $(S, T)$ where $S$ is a state and $T$ a typeless state such that:

   (a) $S(PC_i)$ points to that instruction, and $T(PC_i)$ points to the next instruction in the program.

   (b) $T(R) = t^S$. That is, the value of register variable $R$ in $T$ is the result of evaluating the term $t$ in state $S$.

   (c) For any state variable $v$ other than $PC_i$ and $R$, $T(v) = S(v)$.

2. An execution of an instruction $v := R$ by process $p_i$ is a pair $(S, T)$ where $S$ is a state and $T$ a typeless state such that:

   (a) $S(PC_i)$ points to that instruction, and $T(PC_i)$ points to the next instruction.

   (b) $T(v) = S(R)$.

   (c) For any variable $v'$ other than $v$ and $PC_i$, $T(v') = S(v')$.

Specifically, for Lamport's Simple algorithm the write instructions are the $N$ instructions at line 1: $x[i] := 1$, and the read instructions are the instructions at line 2: $y[i] := x[(i-1)\ modN]$. The notion of "next instruction" is obvious here. The steps of $p_i$ of the Simple algorithm are as follows.

1. $(1_i, 2_i)$ steps are executions of the instruction $x[i] := 1$ of line 1.

$$(1_i, 2_i) = \{(S, T) \in States \times \text{typeless-states} \mid S(PC_i) = 1 \wedge T(PC_i) = 2 \wedge T(x[i]) = 1 \wedge Eq(S, T, PC_i, x[i])\}.$$

   Here, $Eq(S, T, PC_i, x[i])$ is the formula which says that $T(v) = S(v)$ holds for every $v \in SysVar \setminus \{PC_i, x[i]\}$ .

2. $(2_i, 3_i)$ steps are executions of the instruction $y[i] := x[(i-1)\ modN]$.

$$(2_i, 3_i) = \{(S, T) \in States \times \text{typeless-states} \mid S(PC_i) = 2 \wedge T(PC_i) = 3 \wedge T(y[i]) = S(x[(i-1)\ modN]) \wedge Eq(S, T, PC_i, y[i])\}.$$

The set of all steps by process $p_i$ is $(1_i, 2_i) \cup (2_i, 3_i)$ , and the set of all steps is $Steps = \bigcup\{(1_i, 2_i) \cup (2_i, 3_i) \mid 0 \leq i < N\}$. There is no step $(S, T)$ in which $S(PC_i) = 3$. This indicates that $p_i$ stops when its program counter reaches line 3.

The following lemma is obvious, but in large programs it may involve an intensive checking of the algorithm; going over all steps and proving that they retain the type of every state variable.

**Lemma 2.1** (Type consistency of the steps)**.** *If* $(S, T)$ *is any step of the Simple algorithm, then* $T \in States$*, i.e.* $T$ *is a type respecting state.*

As a consequence of this lemma we may restrict our attention to type respecting states and the steps are henceforth only pairs of (type respecting) states. We will not mention again typeless states. Note, that if $S$ is any state and $i$ a process index such that $S(PC_i) = k$, where $k = 1, 2$, then there exists a state $T$ such that $(S, T)$ is a $(k_i, (k+1)_i)$ step (by $p_i$).

**Definition 2.2.** *A history sequence is a sequence of states* $S_0, \ldots,$ *such that* $S_0$ *is some initial state and every two adjacent states in the sequence are steps. I.e.* $(S_k, S_{k+1}) \in$ Steps.

A history sequence is a representation of some possible execution of the system. In Lamport's Simple algorithm, since any process can execute only two steps, any history sequence contains $\leq 2N$ steps. Moreover, a maximal history sequence reaches a terminating state, that is a state $S$ such that $S(PC_i) = 3$ for every index $i$. Indeed, if the last state, $S_k$, in some history sequence, is not terminating, and $i$ is an index such that $S_k(PC_i) \neq 3$, then $S_k(PC_i) \in \{1, 2\}$ and there is a state $S_{k+1}$ such that $(S_k, S_{k+1})$ is a $p_i$ step that extends the given history. Thus the last state in a maximal history is a terminating state $S_{2N}$.

An *invariant* of a distributed algorithm is a statement $\sigma$ such that (1) the initial state satisfies $\sigma$, and (2) for every step $(S, T)$ if $S$ satisfies $\sigma$ then $T$ satisfies $\sigma$ as well. In order to completely understand this definition we must explain what is a statement and what it means for a state to satisfy it.

First define the atomic propositions from which all statements are built with the help of the logical connectives. In our case, the atomic propositions are very simple; they have the form $v = a$ where $v \in SysVar$ is any system variable and $a$ is some value. For example, $PC_5 = 2$ is an atomic proposition. We prefer to present $(PC_5 = 2)$ as a single symbol, an atomic proposition, rather than a compound equational formula.

The logical connectives are the negation symbol $\neg$, the conjunction $\wedge$, disjunction $\vee$, and implication $\Rightarrow$. Then the *propositional formulas* are defined by applying the following rules.

1. Any atomic proposition is also a propositional formula.
2. If $\alpha$ is a propositional formula then so is $\neg\alpha$.
3. If $\alpha$ and $\beta$ are propositional formulas then so are $(\alpha \wedge \beta)$, $\alpha \vee \beta)$, and $(\alpha \Rightarrow \beta)$.

Here is an example of a propositional formula: $((PC_5 = 2) \Rightarrow (x[5] = 1))$.

Given a state $S$ and a propositional formula $\sigma$, the relation $S \models \sigma$ which says that $S$ satisfies $\sigma$ (or $\sigma$ holds in $S$, or is true in $S$) can be defined by induction on the structure of $\sigma$.

1. For atomic $\sigma$ of the form $v = a$ we define $S \models (v = a)$ iff $S(v) = a$.

2. $S \models \neg\alpha$ iff $S \not\models \alpha$ (which means that it is not the case that $S \models \alpha$).

3. $S \models \alpha \wedge \beta$ iff $S \models \alpha$ and $S \models \beta$. Also $S \models \alpha \vee \beta$ iff $S \models \alpha$ or $S \models \beta$.

4. $S \models \alpha \Rightarrow \beta$ iff $S \models \beta$, or $S \not\models \alpha$.

For example, $S \models (PC_7 = 2 \Rightarrow \neg PC_7 = 3)$. (The main point is that if $S(PC_7) = 2$ then it is not the case that $S(PC_7) = 3$.)

Now that we have defined propositional formulas and the satisfaction relation we can give a precise definition: a propositional formula $\sigma$ is an invariant when the initial state satisfies $\sigma$, and for every step $(S, T)$ if $S \models \sigma$ then $T \models \sigma$. But we soon discover that in practice the statements that we want to prove about our states are more complex than those simple propositional formulas that we have defined. Here are some examples that illustrate this and the ways with which we can cope with more complex but obviously needed statements.

1. Suppose that $X \subseteq Type(v)$ is some finite set, and consider the statement $\sigma$ written as $v \in X$. Surely $\sigma$ holds in state $S$ iff $S(v) \in X$, but we do not have the membership relation in the language for which satisfaction was defined and the set $X$ itself is not a term in that language. In case $X = \{a_1, \ldots, a_m\}$ we can view $v \in X$ as a shorthand for $v = a_1 \vee \cdots \vee v = a_m$. For another example, the proposition $PC_i = 1, 2$ is a shorthand for $PC_i = 1 \vee PC_i = 2$. When $\sigma$ is $v \in \{a_1, \ldots, a_m\}$ then viewing $\sigma$ as a shorthand for a well-defined propositional formula allows us to understand $S \models \sigma$ in a well defined way.

2. Consider a statement of the form $\exists j\,(y[j] = 1)$. Assuming as we do that variable $j$ varies over the set of indexes $\{0, \ldots, N-1\}$ (process indexes) the meaning of this statement is clear: for some $j$, $y[j] = 1$. This is exaclty the statement that we must prove to hold in the final state of any history sequence of Lamport's Simple algorithm. Here too we can view this proposition as a shorthand for the disjunction $y[0] = 1 \vee y[1] = 1 \vee \cdots \vee y[N-1] = 1$ (whose length depends on the number of processes $N$). But this translation is quite cumbersome, would it not be more natural to define the notion of state not as a function that assigns values to the state variables but as a structure in the model theoretic sense in which quantifications of the form $\exists j\,\varphi$ are allowed? In that approach, $y$ would be a function name and $j$ a variable that varies over the sort of indexes, $\{0, \ldots, N-1\}$, so that the quantification sentence $\exists j\,(y(j) = 1)$ is a well-formed first-order sentence. Viewing states as Tarskian structure is a useful approach which is not pursued here, since we do not want to delay too much the presentation of the required

invariant. Hence we view a statement such as $\exists j\,(y[j] = 1)$ as long disjunction. Likewise, a statement such as $\forall j(PC_j = 1)$ is a shorthand for the long conjunction $\bigwedge_{0\le j<N}(PC_j = 1)$.

3. Here is an example of a statement about states that acquires its meaning from our external knowledge of mathematics. Suppose we want to say that there are two successive registers (in the circular array $x[0], \ldots, x[N-1]$) that have the same value. The most natural way would be $\exists j\,(x[j] = x[j+1 \pmod N])$. It is possible to translate this statement into a propositional formula, for example in case $N = 3$ we can rewrite it as $x[0] = x[1] \vee x[1] = x[2] \vee x[2] = x[0]$. But most readers, I believe, would prefer the compact formula that relies on our understanding of what $(x \mod N)$ means.

Now the usefulness of invariants comes from the following obvious theorem.

**Theorem 2.3.** *If $\sigma$ is an invariant, then for every history sequence $S_0, \ldots$ and for every index $i$ in this sequence, $S_i \models \sigma$.*

Be careful however: there are statements that hold in every state of any history sequence and yet are not invariants. For example, the statement that we have to prove is the following implication, $\sigma$,

$$(\forall j\ PC_j = 3) \Rightarrow (\exists j\, y[j] = 1)$$

which says that if all processes have stopped then there is (at least) one index $j$ with $y[j] = 1$. When we proved intuitively that after every process stops, at least one process $p_i$ has set $y[i] = 1$, we proved that *every* state in the history satisfies this proposition $\sigma$ (because only the final state in the history satisfies the antecedent of $\sigma$). And yet, $\sigma$ is not an invariant.

We are ready to suggest an invariant for the simple algorithm

1. Let $\beta$ be the conjunction $\bigwedge_{0\le j<N}\beta_j$ where for every index $j$, $\beta_j$ is the proposition

$$PC_j = 2, 3 \Rightarrow x[j] = 1.$$

2. Let $\gamma$ be the following implication assertion:

$$\left(\bigwedge_{0\le i<N} PC_i = 3\right) \Rightarrow \exists j\,(y[j] = 1).$$

**Theorem 2.4.** *The conjunction $\beta \wedge \gamma$ is an invariant*[1]

*Proof.* Firstly we have to prove that the initial state $I_0$ satisfies $\beta \wedge \gamma$. To prove that $I_0 \models \bigwedge_{0 \le j < N} \beta_j$, we have to prove for every index $j$ that $I_0 \models \beta_j$. Since $I_0(PC_j) = 1$ for every $j$, $I_0$ negates the antecedent of $\beta_j$, which implies that it satisfies the required implication. Also, $I_0 \models \gamma$ because $I_0$ negates the antecedent of $\gamma$.

For the main part of the proof that $\beta \wedge \gamma$ is an invariant we take an arbitrary step $(S, T)$ of the Simple algorithm (by an arbitrary process $p_{i_0}$), assume that $S \models \beta \wedge \gamma$ and prove that $T \models \beta \wedge \gamma$. We first deal with $(1_{i_0}, 2_{i_0})$ steps and then with $(2_{i_0}, 3_{i_0})$ steps. For each kind of step we prove that $T \models \beta$ and then that $T \models \gamma$.

1. *Case* $(S, T) \in (1_{i_0}, 2_{i_0})$. To prove that $T \models \beta$ we prove separately that $T \models \beta_{i_0}$ and that $T \models \beta_j$ for any $j \neq i_0$. To prove that $T \models \beta_{i_0}$, we note that $T \models x[i_0] = 1$ by definition of this step. Thus $T$ satisfies the consequent of $\beta_{i_0}$ and hence $T \models \beta_{i_0}$.

   For $j \neq i_0$ we note that $(1_{i_0}, 2_{i_0})$ steps do not change any of the variables $PC_j$, $y[j]$, and $x[j]$. The assumption that $S \models \beta$ implies that $S \models \beta_j$ and it immediately follows that $T \models \beta_j$ as well, for $\beta_j$ is built solely from $PC_j$ and $x[j]$. (If for every variable $v$ that appears in formula $\sigma$, $T(v) = S(v)$, then $T \models \sigma$ iff $S \models \sigma$.)

   To prove that $T \models \gamma$, we note that $T \models PC_{i_0} = 2$ and hence that $T$ negates the antecedent of $\gamma$.

2. *Case* $(S, T) \in (2_{i_0}, 3_{i_0})$. We first prove that $T \models \beta$. Since $S \models \beta$ and $S(PC_{i_0}) = 2$, $S \models x[i_0] = 1$. The step $(S, T)$ is a reading step which did not change the value of $x[i_0]$ which is 1, and hence $T \models PC_{i_0} = 2, 3 \Rightarrow x[i_0] = 1$. That is, $T \models \beta_{i_0}$. For $j \neq i_0$ it is clear that $T \models \beta_j$ (since $S \models \beta_j$ and this step did not change any of the variables of $\beta_j$). So $T \models \beta$.

   To prove that $T \models \gamma$ we assume that $T$ satisfies the antecedent of $\gamma$, i.e. $T \models \bigwedge_{0 \le i < N} PC_i = 3$. Hence for every index $i \neq i_0$, $S \models PC_i = 3$, and since $S \models \beta$, $S \models x[i] = 1$ for every $i \neq i_0$. For $i_0$, $S(PC_{i_0}) = 2$ and hence $S \models x[i_0] = 1$ as well. In particular, $S(x[(i_0 - 1) \mod N]) = 1$, and hence $T(y[i_0]) = 1$. So $T \models \exists j\, (y[j] = 1)$.

□

The following is a reformulation of Theorem 1.1 in the invariant-based framework. As we argued, any execution of the Simple algorithm reaches a final state,

[1] This invariant is due to Lorin Hochstein, https://stackoverflow.com/questions/24989756/what-is-the-inductive-invariant-of-the-simple-concurrent-program. It is much simpler than the invariant that I have had in mind before seeing Hochstein's.

a state in which $\bigwedge_{0\le i<N} PC_i = 3$ holds. The invariant holds in that state as it holds in any state of the execution. Hence the following theorem implies that $y[j] = 1$ holds for some $j$ in the final state.

**Theorem 2.5.** *The following formula holds in every state.*

$$\gamma \wedge \left( \bigwedge_{0\le i<N} PC_i = 3 \right) \Rightarrow \exists j\, (y[j] = 1).$$

*Proof.* This is obvious since $(\bigwedge_{0\le i<N} PC_i = 3)$ is the antecedent of $\gamma$ and $\exists j\, (y[j] = 1)$ is its consequent. □

An attentive reader may have noticed that the particular function $i \mapsto ((i-1) \bmod N)$ plays no role in the proof that $\beta \wedge \gamma$ is an invariant, and we can replace it with an arbitrary function $F$ from the index set $[0, \ldots, N)$ to itself. Thus line 2 of the algorithm (i.e. $y[i] := x[(i-1) \mod N]$) can be changed to $y[i] := x[F(i)]$ and the correctness proof remains (almost) the same. So, from now on, Lamport's Simple Algorithm refers to the following code.

```
1. x[i] := 1;
2. y[i] := x[F(i)]
3. end.
```

Figure 2: Lamport's Simple algorithm with an arbitrary function $F$ from the index set $[0, \ldots, N)$ to itself instead of the function $i \mapsto ((i-1) \bmod N)$.

## 2.1 Challenges to the invariant approach

The intuitive argument of Lamport for Theorem 1.1 is flexible in the sense that it can easily be adapted to other communication modes, such as message passing and regular registers. The invariant-based approach, so it seems, does not display such a measure of adaptability. We shall consider two example which show that adaptation of the intuitive argument to the message-passing and to the regular-register modes of communication is indeed easy. For the invariant-based proof, however, arguing about regular registers seems to be a challenge.

For the first example, assume that in addition to the $N$ processes $p_0, \ldots, p_{N-1}$, we have a certain number of relay stations $L_1, \ldots, L_k$ (normally $K$ is smaller than $N$). There are message channels connecting every process to every relay, and we assume that messages arrive in order and are never lost. There is however a temporal gap between the time a message is sent and the time it reaches its destination. Every relay $L$ has for every $0 \le i < N$ a local variable $V_i^L$ with initial

value 0. The idea in this arrangement is to implement for each $p_i$ a register $R_i$ on which $p_i$ can write and every process can read as in the following scheme. To write a message $m$ on $R_i$ process $p_i$ sends $m$ to every relay and waits for a confirmation message. When relay $L$ receives $m$ from $p_i$ it writes $m$ on $V_i^L$ and sends back a confirmation. To read register $R_i$, process $p_j$ sends a request to some relay, $L$, and waits to get an answer from $L$ of the the local value kept at $V_i^L$. Process $p_j$ may also ask a relay for the sequence $V_0^L, \ldots, V_{N-1}^L$ of all local values. If after awhile $p_j$ does not get an answer (perhaps $L$ is very busy or the channel is slow) $p_j$ may try to get an answer from another relay. This scheme makes it harder to write a value, but it ofers a greater flexibility for the readers.

The variant of Lamport's Simple algorithm that we consider is the following. Each process $p_i$, in the first phase of the algorithm, writes a certain value (say 1 or some nonzero value that depends on $i$) on register $R_i$ by sending messages and waiting for confirmations from all relays. Then, in the second phase, it sends a reading request to some relay $L$ (or possibly to a small number of them), asking to get the values of all registers that $L$ has locally kept.

We claim that there is at least one process $p_{j_0}$ that gets in its execution an answer containing all the value that the $N$ processes have sent (i.e. no initial value is in that answer). Indeed, for every process $j$ consider the messages sent in the first phase of $p_j$ and their confirmations, and let $T_j$ be the moment of the last confirmation arrival to $p_j$. Then let $j_0$ be the index with maximal $T_{j_0}$. We claim that every relay $L$ has received by moment $T_{j_0}$ first phase messages from every $p_i$. Indeed, as $T_i \leq T_{j_0}$ process $p_i$ has by moment $T_{j_0}$ received confirmation from every $L$ which ensures that the message of $p_i$ has reached $L$. Hence, when $p_{j_0}$ executes its second phase, any answers that it obtains from any relay $L$ contains only non-initial values.

The second modification that we consider is the usage of regular registers instead of serial ones. This is our main example, which accompanies our discussion and introduction of new concepts. We ask if the Simple algorithm of Figure 2 is still correct with this replacement of serial with regular registers. Before giving a positive answer we need a definition of regularity, a notion introduced and defined by Lamport [4] in order to investigate the situation where read and write operation executions can be concurrent. (Two events $e_1$ and $e_2$ are said to be concurrent if they are incomparable in the precedence relation ordering $<$, that is neither $e_1 < e_2$ nor $e_2 < e_1$.)

**Definition 2.6.** *Register R is regular if (1) there is a specific (serial) process that can write on R (we say that R is a single-writer register), (2) there is an initial write on R (which precedes every read and every other write event on that register), and (3) a read of R (by any process) returns a value v that satisfies the following requirement. If the read is not concurrent with any write on R, then v is*

*the value of the last write on R that precedes the read, but if the read is concurrent with one or any number of write events, then either v is the value of the last write on R that precedes the read or else v is the value of some write event that is concurrent with the read.*

It would be difficult to give an invariant based proof even to the simplest algorithm in case the registers are regular, but the intuitive argument of Section 1 applies almost verbatim in that case. Indeed, suppose that the temporal extension of every read/write event is represented by an interval (of rational numbers for example), and let $begin(e)$ and $end(e)$ denote the left and right end points of the interval of event $e$. Since the number of processes is finite, the number of executions of line 1 (i.e., $x[i] := 1$) is finite and among these executions let $w$ be one with $end(w)$ maximal. Suppose that $p_{i_0}$ is the process that executes $w$, and let $r$ be the read event by $p_{i_0}$ that corresponds to the execution of line 2, $y[i_0] := x[j]$ where $j = F(i_0)$ ($F$ an arbitrary function from the index set to itself). Now, if $e_1$ is the execution of line 1 by $p_j$, then we have the following temporal relations: $end(e_1) \leq end(w) < begin(r)$. Hence $e_1 < r$. That is, the write by $p_j$ on $x[j]$ precedes the read of that register by $p_{i_0}$, and since there are just two write events on $x[j]$ (the initial of value 0 and $e_1$ of value 1) it follows from the definition of regular registers that $r$ returns the value 1 that was written by $p_j$ on $x[j]$.

These challenging examples suggest that there could be some value in searching for formal mathematical proofs that follow the intuitive and informal arguments. Such proofs would convey a greater assurance in the correctness of the protocol than that of the intuitive argument, but would share some of its benefits. In particular, we expect that such proofs will be more flexible in adapting themselves to different communication frames.

The plan for the rest of the paper is to present a mathematical framework that allows proofs of properties of distributed algorithms that do not search for invariants but rather follow the intuitive arguments that are based on investigations of temporal relations between the events of the distributed system. In the following section the notion of system executions as Tarskian structures is introduced and used to define regular registers. Then in Section 4 we characterize those system executions that arise as executions of the Simple algorithm of Lamport when the registers employed are assumed to be regular, and we prove the required property of these executions. Namely that whenever all processes terminate there is some index $i$ for which $y[i] = 1$. That proof resembles in its shape and flavor the intuitive correctness argument that was given above. Section 5 concludes the article.

# 3 Tarskian System executions

We shall be interested here in a very specific kind of structures which we call *moment based system executions* (or *system executions* for short, borrowing the term introduced by Lamport [4]). A system execution is an abstract representation of some execution of a system that consists of concurrently executing processes; it is one execution out of the manifold of all possible executions. So that a proof that a certain property holds in all system executions is a proof that the protocols that the processes employ ensure that property. As structures in the model theoretic sense, these are interpretation of some logical language in which the properties that interest us can be formulated. System executions are "many sorted" structures, which means that the universe of each structure is a union of its different sorts (types of elements): events, data etc. Our structures are *moment based* which means that one of the sorts of its members is the sort *Moment* which represents time. We can take the natural numbers or the rational numbers as our *Moment* sort, but it makes sense to leave this sort as some unspecified linear ordering, and to determine a specific ordering only if needed.

When specifying a logical language, the term *signature* refers to the collection of (non-logical) symbols that the language employs.

**Definition 3.1.** *A moment based system execution* signature *is a collection of symbols that contains the following items (and possibly more).*

1. There are three kinds of sorts: *Event*, *Atemportal*, and *Moment*. *Event* and *Atemportal* are disjoint, and *Moment* is a subsort of *Atemportal*. The idea is that an event is located in time, it has a beginning and an end which are moments. A moment is atemporal in the sense that it is an instant of time not something that instants of time can characterize. Another atemporal sort can be, for example, the sort of values of messages or the sort of register values.

2. There is a binary relation symbol $<$ defined on *Moment*. We also write $x \leq y$ as a shorthand of $x < y \vee x = y$. For moments $t_1, t_2$, relation $t_1 < t_2$ means that $t_1$ is earlier than $t_2$.

3. There are two function symbols *begin* and *end* from *Event* into *Moment*. We think of event $e$ as extended in time and its temporal extension is represented by the interval $[begin(e), end(e))$ where $begin(e) < end(e)$.

4. There are possibly other predicates, functions, and constants in the signature. These depend on the particular application for which the language is designed.

The following (moment based) signature will be used in our proof for the Simple algorithm of Lamport.

**Definition 3.2.** *The $L_{simple}$ signature consists of the following symbols.*

1. The sorts are *Event*, *Moment*, *Index*, *Data*, and *Address*. (Members of sort *Event* are the read and write events; $Data = \{0, 1\}$ which is the sort of values read and written; *Index* which is the set of indexes of the processes ($0, \ldots, N-1$ in our case); and *Address* which is the sort of registers. Sorts can be used as predicates in the language: we can write, for example, $Moment(t)$ to say that variable $t$ is a moment.

2. The set of logical variables[2] are part of the signature. For many-sorted signatures it is convenient to have specific variables for every sort. For example, we shall use $e, f, g, h$ and $r$ and $w$ (possibly with indexes) to vary over the *Event* sort; $i, j$ are *Index* variables, $t, s$ are *Moment* variables, the letter $a$ (with indexes) is an *Address* variable and so on.

3. The language has *constants* (names of objects). Here we have 0 and 1 to denote the *Data* values. We also have a special "undefined" constant $\perp$.

4. We have two unary predicates, *read* and *write* over the events. There is also a binary predicate $init(w, a)$ (to say that $w$ is the initial write event on address $a$. And there is a binary predicate $<$ on the *Moment* sort. (Namely the "earlier than" relation.)

5. The function symbols are the following.

   (a) $begin, end : Event \to Moment$. (For every event $e$, the (left-closed right-open) temporal interval $[begin(e), end(e))$ is the extension of $e$.)

   (b) $register : Event \to Address$. (Every read/write event $e$ is associated with the register, $register(e)$, from which/to which it reads or writes.)

   (c) $pid : Event \to Index \cup \{\perp\}$. For every event $e$, $pid(e) \in Index$ is the (index of the) process that executes $e$. Thus, $pid(e) = \perp$ indicates that it is the system, not any of the $p_i$ processes, that executed $e$. We shall use $pid(e) = \perp$ to say that $e$ is the initial write event on $register(e)$.

   (d) $x : Index \to Address$. (For every index $i$, $x(i)$ is the register denoted $x[i]$ in the protocol.)

   (e) $val : Event \to Data$. (The intention is that if $write(e)$ (or $read(e)$) then $val(e)$ is the value written (respectively returned) by event $e$.)

[2]Logical variables are used in formulas as "place holders" and for quantification; they are different from system variables and program variables.

(f) $\rho : \mathit{Event} \to \mathit{Event} \cup \{\perp\}$ is the "return" function defined over the read events and returning write events (so if $\mathit{write}(e)$, then $\rho(e) = \perp$ indicates that $\rho$ is not defined on write events). The role of $\rho$ is explained later when regular registers are defined. (Roughly speaking, for every read event $r$, $\rho(r)$ is that write event whose value was returned by $r$.)

(g) $F : \mathit{Index} \to \mathit{Index}$. In the simple algorithm of Figure 1, $F(i)$ is $i - 1 \pmod N$, but any function from the index set and into the index set works as well when line 2 of that algorithm is changed into

$$y[i] := x[F(i)].$$

We prefer this slight generalization of the algorithm which is in Figure 2.

The $L_{simple}$ language is the set of (first-order) formulas that can be obtained with the symbols of the signature $L_{simple}$ and the logical symbols: $\forall$ (forall), $\exists$ (there exists), $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication), $\Leftrightarrow$ (bi-implication), and $\neg$ (negation).

It is useful to enlarge our language with definable predicates. For example, for event variables $e_1, e_2$, formula

$$\mathrm{PREC}(e_1, e_2) \equiv \mathit{end}(e_1) \leq \mathit{begin}(e_2) \tag{1}$$

says that event $e_1$ precedes event $e_2$. Abusing notation, we shall henceforth write $e_1 < e_2$ instead of $\mathrm{PREC}(e_1, e_2)$. We write $e_1 \leq e_2$ for $e_1 < e_2 \vee e_1 = e_2$.

Note that since we decided that variable $i$ varies over *Index* values, instead of writing $(\forall i \in \mathit{Index})\, \varphi$ (or $\forall i(\mathit{Index}(i) \Rightarrow \varphi)$) we can write $\forall i \varphi$.

Whenever we have a logical language signature, a class of structures (Tarskian structures) that interpret this signature is defined. An interpreting structure $M$ consists of a universe $|M|$ of *members* of the structure which is the union of the interpretations in $M$ of the diverse sorts of the signature. Any sort $S$ is interpreted in $M$ as a set $S^M \subseteq |M|$. And every predicate $P$ and function symbol $G$ are interpreted in $M$ as a relation $P^M$ and function $G^M$ over the members of the structure. These relations and functions have to respect the types that the signature determines for them.

In our case the interpreting structures, called *moment based system execution* interpret the language $L_{simple}$, and in details we have the following definition.

**Definition 3.3.** *$M$ is a moment based system execution that interprets the signature $L_{simple}$ defined above when the following holds.*

1. *The univers $|M|$ of the structure is the disjoint union of the sets that interpret the sorts.*

$$|M| = \mathit{Event}^M \cup \mathit{Moment}^M \cup \mathit{Index}^M \cup \mathit{Data}^M \cup \mathit{Address}^M \cup \{\perp^M\}.$$

2. *Moment*$^M = \mathbb{N}$ *is the set of natural numbers and* $<^M$ *is its natural linear ordering*[3].

3. *For some* $N \in \mathbb{N}$, *Index*$^M = \{0, \ldots, N-1\}$ *is the set of the first* $N$ *natural numbers. And Data*$^M = \{0, 1\}$. *Sort Event is interpreted as an arbitrary nonempty set Event*$^M$.

4. *For every* $e \in$ *Event*$^M$, *begin*$(e) <$ *end*$(e)$ *holds in* $M$.

5. *A precedence relation* $<$ *is defined on the events as follows: for every* $e_1$ *and* $e_2$ *in Event*$^M$,
$$e_1 < e_2 \text{ iff } end^M(e_1) < begin^M(e_2). \tag{2}$$
*So we use the same symbol,* $<$, *for two purposes.* $m_1 < m_2$ *denotes that moment* $m_1$ *is before moment* $m_2$, *and* $e_1 < e_2$ *denotes that event* $e_1$ *is earlier than event* $e_2$.

6. *The interpretations of the predicates and functions of* $L_{simple}$ *respect their sorts. For example, predicate write is interpreted as a subset write*$^M$ *of Event*$^M$, *and the function* $\rho$ *is interpreted as an arbitrary function* $\rho^M$ *from Event*$^M$ *to Event*$^M$. *Likewise,* $x^M$ *is a function,* $x^M : Index^M \to Address^M$.

An arbitrary interpretation of the $L_{simple}$ language is devoid of interest since there is no reason to expect that it resembles an execution of the simple algorithm or that it has any meaningful relevance. We need to further specify the structures in order to make them useful. This further specification can be done by writing a sentence $\varphi$ in the $L_{simple}$ language which expresses a required property. Then, the manifold of all interpretations that satisfy $\varphi$ (i.e. *models* of $\varphi$) yields a univocal expression of the intended meaning of that property. As an example of properties that can be expressed in the $L_{simple}$ language consider the following which we first express in mathematical English.

> Every register $x[i]$ has exactly two write events: (3)
> the initial write and the write by process $p_i$.

There is more than one way to formally write this proposition. In the following approach we first write a formula $\alpha(w, i)$ which says that $w$ is a write event on register $x[i]$.

$$\alpha(w, i) \equiv write(w) \wedge register(w) = x(i).$$

[3]It is not strictly necessary to interpret *Moment* as the set of natural numbers, but doing so saves us some work since the finiteness conditions (i.e. that every event is preceded by a finite number of events) already follows from this choice since the temporal extensions assigned to different events are different.

Recall that $x$ is a function symbol in $L_{simple}$ and $x(i)$ is a term that denotes an address. So we write in this formula $x(i)$ rather than $x[i]$ because $x[i]$ is appropriate when writing a code.

Formula $init(w, i)$ says that $w$ is the (unique) initial write on register $x[i]$. In words, it says that $w$ is a write event on register $x[i]$ with value 0, that $pid(w) = \perp$, and that for every event $e$ (read or write) that acts on $x[i]$, if $pid(e) = \perp$ then $e = w$, and if $pid(e) \neq \perp$, then $w < e$.

The required formal rendering of (3) is then:

$$\begin{aligned} \forall i \, \exists \, w_0, w_1 \, (init(w_0, x(i)) \wedge \alpha(w_1, i) \wedge pid(w_1) = i \wedge val(w_1) = 1 \wedge \\ \forall w(\alpha(w, i) \Rightarrow w = w_0 \vee w = w_1)). \end{aligned} \tag{4}$$

## 3.1 Regular registers

In his influential paper [4] Lamport defines a regular register as a safe register[4] in which a read that overlaps a write obtains either the old or the new value. We already gave in section 2.1 a definition of regularity which is similar to the definition given in textbooks and articles that use this concept. Here we want to give a definition of regularity by writing an appropriate sentence in $L_{simple}$. In fact, we want to write a formula $reg(a, i)$ (where $a$ is an address variable, and $i$ is an *Index* variable) which says that $a$ is a single-writer regular register whose owner (writer) is the process $p_i$. $reg(a, i)$ is the conjunction of the following four formulas which are expressed here in English rather than in $L_{simple}$.

1. Every event on address $a$ is either a write event or else a read event.

2. Any write event on address $a$ is by process $p_i$, except for the initial write event on $a$. (Since all write events on $a$ are by process $p_i$ (except for the initial write on $a$), and since every process is serial it follows that the write events on $a$ are linearly ordered in time. That is, for all events $w_1$ and $w_2$, if $write(w_1) \wedge write(w_2)$ then $w_1 \leq w_2 \vee w_2 \leq w_1$.)

3. Finally, the heart of regularity of register address $a$ is the following formula expressed by means of the return function $\rho$.

   For every read event $r$ of address $a$, $\rho(r)$ is a write event on that register, with the same value and such that the following two properties hold:

   (a) $\neg(r < \rho(r))$,

   (b) $\forall w \, (write(w) \wedge register(w) = a \rightarrow \neg(\rho(r) < w < r))$.

   (5)

[4] I.e. a read not concurrent with a write gets the correct value.

It is not difficult to check that this definition of regularity by means of the return function $\rho$ is equivalent to the one given in Definition 2.6.

# 4 Correctness of the Simple algorithm for regular registers

Suppose we disregard concurrency issues, and assume no properties whatsoever for the registers that the processes use. In such a system, the solipsistic processes obtain arbitrary values in their read actions, and the write actions have no effect at all. We say that such a system is *non-restricted* because there is no restriction on the behavior of the communication objects that the processes use. In the first part of this section we ask what can still be deduced from the program code in such a case of non-restricted executions? Surely very little, but in the second part we will see that this minute information suffices, together with the specification of regular registers that was given in section 3.1, to conclude the correctness of Lamport's Simple algorithm for regular registers. The point in this maneuver is to obtain a correctness proof that does not need complex invariants when regular registers are used because the issue of regularity is separated from the issue of representing the semantics of distributed protocol executions.

Consider the code of $p_i$ again (but now with an arbitrary function $F$ as in Figure 2.)

1. $x[i] := 1$;
2. $y[i] := x[F(i)]$
3. end.

Even without knowing anything about the registers $x[i]$ and their properties we can say at least this: each process $p_i$ contains two events that are executed one after the other, a write event on register $x[i]$ of value 1 and then a read of register $x[F(i)]$ which returns an arbitrary value in $\{0, 1\}$ and assigns this value to variable $y[i]$. Even without any information about the possible ways in which such write and read events are executed, it is quite obvious that the code dictates the existence of these two events.

How can we *prove* that indeed this is the case that each $p_i$ has these two events? The reader may feel that this statement is so obvious that it needs no proof, and this indeed may be true in our case, but for longer protocols one must have a way to handle such proofs. We must have a framework that relates the code of $p_i$ to the set of events that it generates in non-restricted executions. Note that the properties that we want to prove involve no interleaving of actions since no action of one process can have any effect on the actions of another process in case of non-restricted executions. In other words, we can think of a system in which $p_i$

runs alone; its write actions have no effect and its read actions return arbitrary values that the system somehow randomly furnishes. It would still be true in such a system that $p_i$ contains only two events as stated above. For that reason we expect the proof to be rather simple, and even if it involves the usage of invariants, these invariants must be as simple as invariants of algorithms designed for a single process. However, we shall not develop such a required framework in this short paper[5], and ask the reader to agree that any non-restricted execution of Lamport's Simple algorithm satisfies the minimal requirements embodied in the following definition.

**Definition 4.1.** *A moment-based system-execution structure that interprets the $L_{simple}$ language is said to be a non-restricted system-execution of Lamport's Simple algorithm if it satisfies the following sentence. For every index i ($0 \leq i < N$)*

$$\begin{aligned} \exists w, r\,(w < r \wedge write(w) \wedge read(r) \wedge pid(w) = pid(r) = i \wedge \\ val(w) = 1 \wedge val(r) \in Data \wedge register(w) = x(i) \wedge \\ register(r) = x(F(i)) \wedge \forall e(pid(e) = i \Rightarrow (e = w \vee e = r))). \end{aligned} \tag{6}$$

In words, formula (6) says that for every process index $i$ there are a write event $w$ and a read event $r$, both by $p_i$ (i.e. with their *pid* equal to $i$) with $w$ of value 1 and an unspecified value of $r$ in $Data = \{0, 1\}$, such that $w$ is a write on register $x(i)$ and $r$ a read of register $x(F(i))$, and such that every event $e$ in $p_i$ is either $w$ or $r$.

**Definition 4.2.** *We say that M is a system-execution of Lamport's Simple algorithm if M is a moment-based system-execution that satisfies the conjunction of two properties expressed in the $L_{simple}$ language:*

1. *the non-restricted properties of (6),*
2. *the statement $\forall i\,(reg(x(i), i)$ which says that for every $i \in Index$, register $x(i)$ is a regular single writer register of process $p_i$. (See section 3.1 for the definition of $reg(x(i), i)$.)*

**Theorem 4.3.** *Let M be any system execution of Lamport's Simple algorithm with regular registers, then there is an index i such that the read event by process $p_i$ obtains the value* 1 *in its read of register $x(F(i))$*[6].

*Proof.* Let's write down in details an enumeration of our assumptions, so that every step in the proof can be justified by one of the assumptions.

[5] A forthcoming paper "Kishon's Poker game" will elaborate on this issue.

[6] Since any serial register is a fortiori regular, a proof of the theorem for regular registers establishes it for serial registers as well.

1. *Properties of non-restricted executions.*

   (a) For every $i \in Index$ there are two events in $p_i$: $w(i)$ and $r(i)$ such that $w(i)$ is a write of value 1 on register $x(i)$, and $r(i)$ is a read of register $x(F(i))$.

   (b) Every event $e$ such that $pid(e) = i$ is either $w(i)$ or $r(i)$. (I.e. $p_i$ contains no events other than $w(i)$ and $r(i)$.)

2. All addresses $x(i)$ are regular registers. *Properties of the regular registers* $x(i)$.

   (a) No event is both a read and a write event.

   (b) Any write event on address $x(i)$ is by process $p_i$, except for the initial write event on $x(i)$ which is denoted $init(i)$. The value of this initial write is 0 and it precedes any read event and non-initial write event.

   (c) If $r$ is a read of register $x(i)$, then $\rho(r)$ is a write on register $x(i)$ such that the following hold.

$$\begin{aligned} &val(r) = val(\rho(r)) \wedge \neg(r < \rho(r)) \wedge \\ &\forall w(write(w) \wedge register(w) = x(i) \rightarrow \neg(\rho(r) < w < r)) \end{aligned} \tag{7}$$

The proof of our theorem relies on the assumed finiteness of the set *Index*, which implies the following property.

> Suppose that $m$ is a definable function which maps every $i \in Index$ to some moment $m(i) \in Moment$. Then there is some index $i_0$ such that $m(i) \leq m(i_0)$ for every $i \in Index$.

Using this property the proof proceed as follows. For every index $i \in Index$ let $init(i)$, $w(i)$ and $r(i)$ be as described above. That is, $init(i)$ is the initial write event on register $x(i)$ (of value 0), $w(i)$ is the write of value 1 by process $p_i$ on register $x(i)$, and $r(i)$ is the read of register $x(F(i))$ with value $val(r(i))$. Then define $m(i) = end(w(i))$. (That is, $m(i)$ is the right-end of the temporal interval of the write event $w(i)$.) The function $i \mapsto m(i)$ is definable, and hence there is by the finiteness property an index $i_0$ such that

$$m(i) \leq m(i_0) \text{ for every } i \in Index. \tag{8}$$

We shall prove that $r(i_0)$ is a read of value 1[7]. We have the following ordering relations and their consequences.

[7]It is easier to prove this by yourself than to read it. That's the problem with too formal proofs.

1. For every index $j$, $init(j) < w(j) < r(j)$.

2. In particular for $j = F(i_0)$, $init(j) < w(j) < r(j)$.

3. By maximality of $m(i_0)$, $end(w(j)) = m(j) \leq m(i_0)$.

4. But $w(i_0) < r(i_0)$, and hence $m(i_0) = end(w(i_0)) < begin(r(i_0))$.

5. Hence $m(j) < begin(r(i_0))$, which implies that $w(j) < r(i_0)$. (See equation (2).)

6. Thus we have $init(j) < w(j) < r(i_0)$.

7. But $init(j)$ and $w(j)$ are the only write events on register $x(j)$. (Use 2(b) and 1(b): by 2(b) the write events on $x(j)$ are $init(j)$ and a write by $p_j$; by 1(b), the write by $p_j$ can only be $w(j)$ because $r(j)$ is not a write event.)

8. Since $F(i_0) = j$, $r(i_0)$ is a read of register $x(j)$ (by 1(a)). Since $x(j)$ is regular, equations (5) imply that

$$val(r(i_0)) = val(\rho(r(i_0))).$$

   Being a write on register $x(j)$, $\rho(r(i_0)))$ is either $init(j)$ or $w(j)$. But it cannot be $init(i)$ (by the second line of (7)). Hence $w(j) = \rho(r(i_0))$, and so $1 = val(w(j)) = val(r(i_0))$.

9. So there is a read event of register $x(F(i_0))$ by $p_{i_0}$ of value 1, and this concludes the proof of Theorem 4.3.

□

# 5 Conclusion

The Simple algorithm of Lamport in his short and thought provoking article [5] serves here as a platform to present an approach to concurrency that is different from the one that is usually taught in courses and textbooks. The correctness proof in this approach is not concerned with invariants but rather relies on two kinds of properties from the conjunction of which the desired correctness conclusions follow. The first is said to be *non-restricted* because it does not restrict the behavior of the communication devices that the processes use, and it specifies the system execution as a disjoint union of processes that run in isolation from each other. The second kind of properties is the specification of the communication devices that are used (regular registers in our case). The thesis proposed in

[1] and [2] is that, in general, correctness proofs of distributed system can be obtained by investigating the conjunction of these two kinds of properties. In the first one, properties of communication devices are ignored, and in the second the code of the executing process is irrelevant. So, unlike the state-step-history approach which looks for invariants with which the correctness can be established, we have here a clear separation between the two aspects of the system: the meaning of its programs and the specification of its communication devices.

The approach to correctness proofs of distributed algorithms in which properties of Tarskian system executions are used has some merit which comparison with the invariant-based approach may highlight. The most notable one is in obtaining a formal proof that resembles the intuitive arguments with which the investigator understands the distributed algorithm and its correctness. This understanding allows a greater flexibility in applying the ideas of the algorithm to similar algorithms and in extending its range of applications. There certainly are advantages of the invariant-based approach: it is based on relatively simpler concepts and many tools were developed that facilitate its applications (especially to large scale systems). In my experience with distributed algorithms, there are cases in which their Tarskian abstract modeling brought about a clearer correctness proof, but there are cases in which the invariants revealed something about the program that is not evident from its intuitive analysis and its mathematical explication.

But think about the developer of a system who has found a suitable invariant which establishes the correctness of some distributed algorithm. Would this invariant help the programmer in answering the question about extending the algorithm to work with regular registers or with messages? Probably not, probably an intuitive argument would be more useful in reaching a positive answer or a negative counterexample. But intuitive arguments may be misleading and a good education of the programmer, one that contains Tarskian structures in its syllabus, can help in reshaping intuitive arguments as trustworthy mathematical proofs.

It is by nature of the invariant approach that the semantics of a concurrent program and the specification of communication devices that it uses are encompassed in a single framework, that of global states, steps, and histories. Indeed, a global state records the values of the registers, and a writing-step represents the atomic change that the write action brings about. Thus serial registers are easily represented in a framework that takes care at once of both the specification of serial registers and the semantics of distributed program executions. It is possibly because of this naturalness of dual representation in one framework that the impression was created that this must always be the case, and the paradigm that the semantics of distributed code and specification of communication devices should be lumped in the same concept of state-step-history was accepted. But this is not necessarily so; a separation between the semantics of the communication devices and the program semantics is possible, and even desired I think in cases that it

simplifies the correctness proofs and makes them more natural. Such a separation was exemplified in the second part of our paper which deals with regular registers.

I believe that with further experience, time will bring a deeper understanding of the different possibilities for modeling concurrent systems and proving their properties. Different approaches will then be available to choose from for each particular application. An education of a computer scientist or engineer could benefit from a program that includes (in addition to learning about invariants of course) openings to other proof methods in general and to the possible usage of Tarskian structures in particular.

# References


[1] U. Abraham. On interprocess communication and the implementation of multi-writer atomic registers. *Theoretical Computer Science*, 149(2): 257-298, 1995.

[2] U. Abraham. *Models for Concurrency*. (Algebra, Logic and Applications, Vol. 11). CRC Press (Gordon and Breach, Taylor and Francis), 1999.

[3] L. Hochstein. https://stackoverflow.com/questions/24989756/what-is-the-inductive-invariant-of-the-simple-concurrent-program.

[4] L. Lamport. On interprocess communication, Part I: Basic formalism, Part II: Algorithms. *Distributed Computing*, 1(2): 77–85 and 86–101, 1986.

[5] L. Lamport. Teaching concurrency. ACM SIGACT News (Distrib. Comput. Column). 40(1): 58-62, 2009.