# Abstracting Communication to Reason about Distributed Algorithms*

Michel Charpentier, Mamoun Filali, Philippe Mauran,
Gérard Padiou, Philippe Quéinnec

LIMA - IRIT, ENSEEIHT
2, rue Charles Camichel
31071 Toulouse cedex, FRANCE
e-mail: {charpov,mauran,padiou,queinnec}@enseeiht.fr, filali@irit.fr

**Abstract.** In distributed systems, message passing is a low level representation of communication resulting in intricate designs and proofs. This paper presents a new abstraction to express communication: the observation. This notion provides a more concise expression of programs and properties, and consequently is an effective help in understanding and reasoning about distributed algorithms. Observations are formalized in the UNITY framework.

We define the observation relation and state its main properties. Then, we present the description and the proof of a generic problem. The abstract level of description separates concerns between the algorithm and its communication pattern. Thus, the topology of observations can be changed while preserving the algorithm properties.

## 1 Introduction

Designing and reasoning about distributed systems is more often a matter of ingenuity than a matter of engineering. This difficulty seems to stem from the very nature of distributed systems:

- distributed systems are *parallel* systems, which results in a combinatorial explosion of the states of the global system (when compared to the states of its component sites);
- distributed systems are *asynchronous* systems, which introduces one more level into the combinatorics: the current state of a site depends on the history of remote sites; this difficulty comes from the impossibility for a site to maintain an accurate knowledge about the current (global) state of the system.

In this context, it seems crucial to support the design of distributed algorithms by providing conceptual tools that:

* 10th International Workshop on Distributed Algorithms (WDAG'96), October 1996, LNCS 1151, pages 89–104.

– ensure the correctness of algorithms, to achieve a safe design, if not a systematic one;
– make it easier to reason about distributed systems by proposing an efficient (i.e. abstract and concise) representation of such systems.

In this purpose, we use the UNITY formalism [2] which provides a sound formal framework [7, 8] and appears to be well suited to describing distributed systems. Our contribution provides an abstraction of interactions between sites. The construct we propose is intended to be in the spirit of UNITY minimalism: it solely aims at expressing what is – according to us – the essential feature of distributed systems: the (a priori) impossibility for a site to maintain an up-to-date global knowledge, due to the lack of a common memory (or clock).

More precisely, we define an *observation* as a relation between two variables, the *source* and the *image*. It states that the values of the image are some of the former values of the source, in a chronological order. We present a set of theorems to relate the properties of the images to the properties of the sources on the one hand, and to link the observation relation to UNITY logical operators on the other hand.

Thus, a first interest of this construct is to provide a set of results that can be used for reasoning about a wide range of distributed algorithms. A second contribution of this construct is to offer a declarative and progressive approach to distributed algorithms development: the observation relations specify the "interactions topology", independently from the specification of the algorithm itself. The observation relation bridges a gap between the UNITY model, where interaction is described with global shared variables, and the message passing model, close to the physical world.

The declarative aspect results in a more concise expression of programs and properties, and consequently is an effective help in understanding and reasoning about distributed algorithms. We illustrate the contribution of observations from this point of view as well as the interest of a sound formal framework through the expression of a solution (based on [15]) to a well-known problem: distributed mutual exclusion.

In the next section, we present the UNITY formalism. The third section introduces the observation relation and states its main properties. Then, we illustrate the use of observations to express and to prove a distributed mutual exclusion algorithm. The last section concludes and describes our current work.

## 2 UNITY

The purpose of the UNITY formalism [2] is to provide a framework to reason about the *correctness* of programs. It is a general model that allows the specification of parallel programs from an operational point of view (writing program statements) as well as from a declarative point of view (stating program properties). Besides, this formalism is designed to make it easier to check the consistency of these two points of view. The minimalism of the UNITY formalism is one of its strong points: it ensures its generality, while guaranteeing its simplicity of use.

The UNITY formalism consists of two parts: a programming language, based on transition systems, and a specification language, based on a linear temporal logic [12], to express safety and progress properties of the corresponding transition systems.

This section describes parts of UNITY syntax and semantics useful to understand the remainder of the paper. A more comprehensive presentation can be found in [2] or [14].

### 2.1 The UNITY Language

A UNITY program $F$ describes a state transition system. It consists of a declaration of variables, a specification of their initial values, and a set of multiple-assignement statements:

- the **declare** section defines the set of state variables along with their type;
- the **initially** section consists of a predicate on state variables, called $init.F$, expressing the possible initial states for the program;
- the **assign** section, denoted by $assign.F$, consists of a set of multiple guarded assignments separated by the $[\!]$ mark.
  UNITY also allows quantification over statements to describe a family of assignments, written as $\langle [\!] v : condition :: instructions(v) \rangle$.

Semantically, executing a UNITY program consists in choosing non-deterministically one assignment and then executing it atomically. This choice is repeated infinitely, assuming weak fairness: each statement is executed infinitely often.

### 2.2 Operational Semantics

To formalize the semantics of observations in accordance with the UNITY formalism, we use the notion of operational semantics of a program. These semantics models define the non-empty set of infinite sequences of states corresponding to possible computations of the program. Given an operational semantics $\mathcal{O}$ and a UNITY program $F$, $\mathcal{O}.F$ is the set of possible computations $\sigma$. For each sequence $\sigma$, the state $\sigma_i$ is a mapping between the state variables and their value.[2] In this paper, variables are handled as state functions and the value of variable $v$ in state $\sigma_i$ is denoted by $v.\sigma_i$. By extension, $v.\sigma$ represents the successive values of the variable $v$ in the computation $\sigma$, and $e.\sigma_i$ denotes the value of the *expression* $e$ in the state $\sigma_i$.

For example, a well known operational semantics is $\mathcal{O}_1$, presented in [17]. For a program $F$, $\mathcal{O}_1.F$ is the set of sequences of states $\sigma$ for which the following properties hold:

[2] To denote the $i$ th element of a sequence $\sigma$, we use the mathematical notation $\sigma_i$ instead of $\sigma.i$.

- $init.F.\sigma_0$
- $\langle \forall i :: \langle \exists s : s \in assign.F :: \sigma_i \; s \; \sigma_{i+1} \rangle \rangle$
- $\langle \forall s : s \in assign.F :: \langle \exists_i^{\infty} :: \sigma_i \; s \; \sigma_{i+1} \rangle \rangle$

In the following, all the definitions and theorems are valid given any operational semantics.

### 2.3 The UNITY Logic

We now define the basic operators of the UNITY logic in an operational semantics $\mathcal{O}$.

- $p$ **unless** $q$ : in any computation, if $p$ becomes true, then it remains true at least until $q$ becomes true. In semantics $\mathcal{O}$, it becomes:

$$p \textbf{ unless } q \textbf{ in } F \equiv \langle \forall \sigma, i : \sigma \in \mathcal{O}.F :: \; (p \wedge \neg q).\sigma_i \Rightarrow (p \vee q).\sigma_{i+1} \rangle$$

- **stable** $p$ : once $p$ becomes true, it remains true[3]:

$$\textbf{stable } p \textbf{ in } F \equiv \langle \forall \sigma, i : \sigma \in \mathcal{O}.F \;\; :: \;\; p.\sigma_i \Rightarrow p.\sigma_{i+1} \rangle$$

- $p \mapsto q$: if $p$ is true, then $q$ will eventually become true:

$$p \mapsto q \textbf{ in } F \equiv \langle \forall \sigma, i : \sigma \in \mathcal{O}.F :: \;\; p.\sigma_i \Rightarrow \langle \exists j : j \geq i :: \;\; q.\sigma_j \rangle \rangle$$

In the operational semantics $\mathcal{O}_1$, these operators are equivalent to Sanders' [16].

### 2.4 Representing distributed algorithms

In distributed computing, a program is structured as a set of processes (or nodes) and communication is usually described in terms of message passing [1, 6, 10].

In the UNITY model, each node is described as a program component and the whole system is implemented as a quantified union:

$$DC = \langle [\!] s : 0 < s \leq N :: Node(s) \rangle$$

A common approach to program message exchanges consists in translating send and receive primitives into append and extract operations on sequences. All components share the variables of type sequence that implement communications. However, readability is poor since message passing is not part of the syntax, and proofs are difficult since message semantics is not part of the logic. We propose another description using observations.

[3] **stable** $p$ is equivalent to $p$ **unless** $false$.

## 3 Observation Semantics

An observation defines a relation over the variables of a distributed program: $`x \prec\!\cdot\; x$ (read $`x$ observes $x$) states that the history of the variable $`x$ is a sub-history of the (remote) variable $x$. The variable $x$ (resp. $`x$) is called the *source* (resp. the *image*).

In the message passing paradigm, communication and synchronization cannot be described separately. Observation aims at describing solely communication. If a specific scheme of synchronization is required, it can be explicitly programmed. For instance, a handshake protocol can be implemented with two mutual observations.

Before describing the observation relation, we define a abstract model of time.

### 3.1 Clock Functions

We assume that an ideal discrete totally ordered time exists. In the following, such a time is represented by the set of natural numbers. A clock function is then defined as an approximation of this ideal time. This approximation satisfies safety and liveness constraints.

More precisely, a clock function is a mapping $C$ from natural numbers to natural numbers, satisfying the following properties:

- **safety 1:** a clock never outgrows the ideal time: $\langle \forall t :: \; C.t \leq t \rangle$
- **safety 2:** a clock is monotonously increasing: $\langle \forall t :: \; C.t \leq C.(t+1) \rangle$
- **liveness:** a clock eventually increases: $\langle \forall t :: \; \langle \exists t' :: \; C.t < C.t' \rangle \rangle$

We write $Clock(C)$ when the mapping $C$ satisfies these properties.

*Properties.*

- the value of any clock at the initial time is 0;
- the identity function is a representation of the ideal time;
- the composition of two clocks is a clock.

### 3.2 The Observation Relation $\prec\!\cdot$

**Definition.** The observation relation between two variables of the same program is defined by:

$$`v \prec\!\cdot\, v \;\mathbf{in}\; F \equiv \langle \forall \sigma : \sigma \in \mathcal{O}.F :: \langle \exists\; C \;:\; Clock(C) :: \langle \forall t :: `v.\sigma_t = v.\sigma_{C.t} \rangle \rangle \rangle \quad (1)$$

This definition states that any value of $`v$ is a previous value of $v$ (clock safety 1), that $`v$ is assigned its values in a chronological order (clock safety 2) and that $`v$ is eventually assigned more recent values of $v$ (clock liveness).

The variable $v$ is called the *observed variable*, or, in short, the *source*, and $`v$ is the *image variable*, or, in short, the *image of* $v$.

If $\langle \forall t : t \neq 0 :: C.t < t \rangle$, the observation is *strict*: the image cannot reflect immediately a change in the value of the source.

**Generalization.** In the previous definition, an observation is a relation between variables of a program. This definition is generalized to take into account any expression defined on the variables, or on the state of the program. Then, the formula:

$$`e \prec e \ \mathbf{in}\ F$$

expresses that the value of the expression $`e$ is always a former value of the expression $e$, that $`e$ takes (part of) $e$ values in a chronological order, and that $`e$ is eventually updated with a more recent value of $e$.

Now, we can define the observation of a tuple of variables. The relation:

$$(`u_1, `u_2, \cdots, `u_n) \prec (u_1, u_2, \cdots, u_n) \ \mathbf{in}\ F$$

states that the variables $u_i$ are observed simultaneously, that is with the same clock function.

We can also consider the observation relation to be a new operator in the UNITY logic. Then, the observation property deals with state predicates, i.e. boolean expressions.

### 3.3 Properties

We consider two kinds of properties: properties of the observation relation itself, and properties about the observation relation and the UNITY logic. Proofs of the following properties are given in [3].

*Notation.* Properties are stated as deduction rules:

$$\frac{p}{q}\ \mathtt{prop} \ \text{ means that } \ \forall F :: \ p \ \mathbf{in}\ F \ \Rightarrow q \ \mathbf{in}\ F$$

where

- $F$ is a UNITY program;
- `prop` is the symbolic name of the property.

$u$ and $v$ stand for expressions of any type, and $p$ is a predicate.

#### Some Properties of the Relation $\prec$

- The observation relation is reflexive, antisymmetric and transitive:

$$\frac{\mathtt{true}}{u \prec u}\ \mathtt{OBS\_REFL}$$

$$\frac{u \prec v \ , \ v \prec u}{\mathbf{invariant}\ u = v}\ \mathtt{OBS\_ANTISYM}$$

$$\frac{u \prec v \ , \ v \prec w}{u \prec w}\ \mathtt{OBS\_TRANS}$$

- Compatibility with function application:

$$\frac{u \prec v}{f.u \prec f.v}\ \mathtt{OBS\_F} \qquad \frac{f.u \prec f.v \ , \ f \text{ injection}}{u \prec v}\ \mathtt{OBS\_F\_INJ}$$

**The Observation Relation and the UNITY Logic**

$$\frac{(\text{`}u, \text{`}v) \prec (u, v)}{\text{`}u \prec u}\ \texttt{OBS\_L} \qquad \frac{(\text{`}u, \text{`}v) \prec (u, v)}{\text{`}v \prec v}\ \texttt{OBS\_R}$$

$$\frac{\text{`}p \prec p\ ,\ \textbf{stable}\ p}{\textbf{stable}\ \text{`}p}\ \texttt{OBS\_STABLE} \qquad \frac{\text{`}p \prec p\ ,\ \textbf{stable}\ p}{\text{`}p\ \textbf{detects}\ p}\ \texttt{OBS\_DETECTS}$$

$$\frac{\text{`}p \prec p\ ,\ p\ \textbf{unless}\ \text{`}p}{p \mapsto \text{`}p}\ \texttt{OBS\_LEADSTO1} \qquad \frac{\text{`}p \prec p\ ,\ \textbf{stable}\ p}{p \mapsto \text{`}p}\ \texttt{OBS\_LEADSTO2}$$

$$\frac{\text{`}u \prec u\ ,\ \forall k :: \textbf{stable}\ u > k}{\textbf{invariant}\ \text{`}u \le u}\ \texttt{OBS\_UPB}$$

The first two theorems state that the projections of an image tuple are images of the projections of the source tuple; the third one states that the image of a stable predicate is stable; `OBS_DETECTS` states that a stable predicate can be detected by its image[4]; the two `LEADSTO` theorems ensures the occurrence of $\text{`}p$ given some properties of $p$; `OBS_UPB` means that the image of a non-decreasing counter is always smaller than the source.

In the framework of reactive programs, which are characterized by an interaction with their environment, the properties between the observation relation and the UNITY logical predicates can be used in two different ways:

- either to deduce properties of the observations based on environment properties, e.g. `OBS_STABLE`,
- or to deduce properties of the environment based on properties of the observation, e.g. `OBS_DETECTS`.

### 3.4 Observations as a Mechanism

In the preceding section, an observation is defined as a relation between variables (or expressions) and we have stated several theorems relating the relation to other UNITY predicates. From this point of view, $\text{`}e \prec e$ is a property that, for a given program, is either true or false. We now intend to use observations to avoid an explicit description of the communication between nodes of a distributed system.

[4] $p$ **detects** $q$ means **invariant** $p \Rightarrow q\ \wedge\ q \mapsto p$.

**The `observe` Section.** An observation is now seen as a mechanism which satisfies the properties of the observation relation. UNITY is extended with a new section called `observe` which describes observation relations assumed by the program. The declaration of these observations replaces an explicit implementation of the mechanism, including the initialization of the images.

When the `observe` section contains $`x \prec x$, a mechanism is assumed to update $`x$ so that $`x$ is assigned former values of $x$ in a chronological order. Moreover, only this mechanism is allowed to modify $`x$.

The UNITY language extended with the `observe` section must be given an operational semantics. Such a semantics, defined as an extension of $\mathcal{O}_1$, can be found in [4].

**An Implementation of an Observation.** We present here a simple implementation of an observation based on the non-deterministic assignment of the current value of the source to the image. The liveness property is ensured by the weak fairness of the UNITY execution model.

**Program** *Obs*
**declare**
  $`v : T;$  $\{\ T$ is the type of $v\ \}$
**initially**
  $`v = v$  $\{$ due to $C.0 = 0\ \}$
**assign**
  $`v := v$
**end**

The proof of $`v \prec v$ **in** $Obs \| G$, where $G$ does not modify $`v$, is given in [3].

## 4 A Mutual Exclusion Algorithm Using Observations

This section gives an extended example of a mutual exclusion algorithm based on observations.

We consider a system consisting of a fixed number of nodes (or processes) and a critical resource that at most one process is allowed to use at any time. We present here an algorithm that ensures this mutual exclusion property. More precisely, the following properties must be achieved:

- safety property: at most one process has permission to use the resource;
- liveness property: every request for the resource is eventually granted.

With respect to this problem, the behavior of a node is a three state automaton. A variable records the current state of each node. Initially, in the "thinking" state, the node is outside the critical section. After a request, it remains in a "requesting" state until executing its critical section in the "mutex" state. Each node completes its critical section in a finite time and returns to the "thinking" state.

A solution to this problem is based on logical clocks. Such an algorithm has been proposed by Ricart and Agrawala [15]. When a node requires mutual exclusion, its request is assigned a timestamp. A node is allowed to enter mutual exclusion when its request is the oldest one. For the thinking nodes, a default request is assumed with a maximal value (infinite value).

In the following, we give a solution based on the same principle, using a slightly different approach for handling logical clocks and request timestamping.

### 4.1 An Observation Oriented Solution

To solve this problem by means of observations, we assign a clock vector[5] $C[1..N]$ to each node called $Mutex(s)$. In this vector, the component $Mutex(s).C[s]$ is the local clock of the node $s$ and the other components are clock images of the other nodes. More precisely, $Mutex(s).C[s']$ with $s' \neq s$ is an image of the remote clock $Mutex(s').C[s']$ (Fig. 1). We specify this observation relation as[6]:

$$\forall s, s' \ :: \ Mutex(s).C[s'] \prec\!\cdot\, Mutex(s').C[s']$$

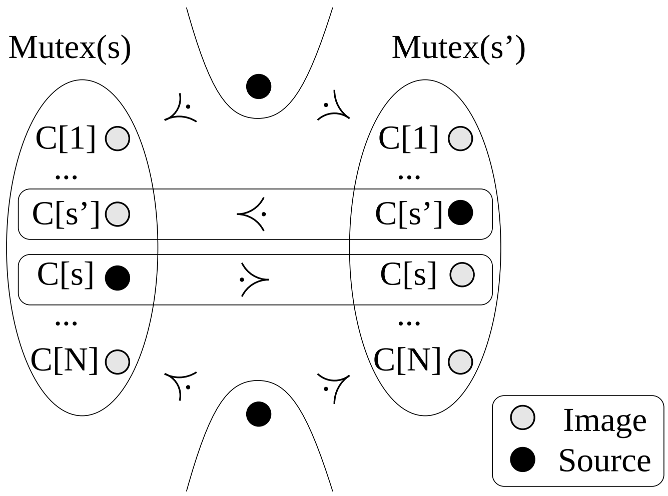


**Fig. 1.** Node observations

The node behavior is described by the initial conditions and the actions taken at each state.

[5] Here, clocks are variables; they must not be confused with the clock functions of the previous section.

[6] This specification includes the relation $Mutex(s).C[s] \prec\!\cdot\, Mutex(s).C[s]$. This relation is redundant ($\prec\!\cdot$ is a reflexive relation) but simplifies the writing of observations.

- Initially, a node is thinking.
- When a node $Mutex(s)$ is thinking, it cannot prevent another node from entering. For this purpose, it infinitely often increases its local clock $C[s]$.
- When a node is requesting, it stops its local clock. It will enter mutual exclusion when its "suspended" clock becomes smaller than all the local images.

Consequently, a local clock $Mutex(s).C[s]$ is updated only by the node $s$ and is observed by the other nodes.

The following UNITY program is a model of this behavior.

**Program** $Mutex(s\ :\ 1..N)$
    $C$ : **array** $1..N$ **of** $Int$;
    $state$: $\{thinking, requesting, mutex\}$;
  **initially**
    $C[s] = s\ \wedge\ state = thinking$;
  **always**
    $Minimum = \langle \forall s'\ :\ 0 < s' \leq N\ \wedge\ s' \neq s\ ::\ (C[s] < C[s']) \rangle$
  **observe**
    $\langle \forall s'\ :\ 0 < s' \leq N ::\ C[s'] \prec Mutex(s').C[s'] \rangle$
  **assign**
      { think and tick }
      $C[s] := C[s] + N$ **if** $state = thinking$
    $[\!]$
      { request and wait }
      $state := requesting$ **if** $state = thinking$
    $[\!]$
      { enter and use }
      $state := mutex$ **if** $(state = requesting) \wedge Minimum$
    $[\!]$
      { leave and tick }
      $state, C[s] := thinking, C[s] + N$ **if** $state = mutex$
**end** $Mutex$

Then, the distributed configuration $DC$ is instantiated as the union of $Mutex$ component programs:

$$DC\ =\ \langle [\!]\ s\ :\ 0 < s \leq N\ ::\ Mutex(s) \rangle$$

The **observe** clause of this union is the conjunction of the corresponding clauses in each component program. For this configuration we obtain the previously specified observations:

$$\forall s, s' ::\ Mutex(s).C[s'] \prec Mutex(s').C[s']$$

*Notation.* In the following, $s.anything$ is an abbreviation for $Mutex(s).anything$. Moreover, $s.mutex$, $s.requesting$ and $s.thinking$ are used for, respectively, $Mutex(s).state = mutex$, $Mutex(s).state = requesting$ and $Mutex(s).state = thinking$.

*Specification.* This algorithm must satisfy the following safety and liveness properties:

– safety:

$$\textbf{invariant } \langle \forall s, s' :: s.mutex \wedge s'.mutex \;\Rightarrow\; (s = s') \rangle \textbf{ in } DC \tag{2}$$

– liveness:

$$\forall s \;::\; s.resquesting \mapsto s.mutex \textbf{ in } DC \tag{3}$$

### 4.2 Proof of the Algorithm

**Clock Properties.** In this solution, a local clock is implemented by means of an integer. Its initial value is the number of the corresponding node. A tick consists in incrementing the clock by $N$. Such an implementation can be seen as a simplified version of Lamport's logical clocks [11] and ensures the following properties:

– the value domain of two clocks are disjoint:

$$\langle \forall s, s' :\; s \neq s' \;:: Dom(s.C[s]) \cap Dom(s'.C[s']) = \emptyset \rangle$$

– a clock is an increasing counter:

$$\forall s, k :: \;\textbf{stable } C[s] > k$$

– clock values are totally ordered:

$$\langle \forall s, s' :\; s \neq s' \;::\; (s.C[s] < s'.C[s']) \vee (s.C[s] > s'.C[s']) \rangle$$

**Mutual Exclusion Property.**

**Lemma 1.** *If a local clock is smaller than the local images, it is also smaller than all the remote clocks.*

*Proof.* Let us consider a node $Mutex(s)$ such that its clock is the minimum of the images:

$$\langle \forall s' :\; s' \neq s \;:: s.C[s] < s.C[s'] \rangle$$

Since local clocks are increasing counters, clock images satisfy (theorem `OBS_UPB`):

$$\langle \forall s' :\; s' \neq s \;::\; s.C[s'] \leq s'.C[s'] \rangle$$

From both preceding properties, it follows:

$$\langle \forall s' :\; s' \neq s \;:: s.C[s] < s.C[s'] \leq s'.C[s'] \rangle$$

Consequently, $C[s]$ is also a global minimum:

$$\langle \forall s, s' :\; s' \neq s \;:: s.C[s] < s'.C[s'] \rangle$$

□

When a node is in mutual exclusion, its clock is the minimum of the images and remains constant. From the previous lemma, we deduce that its clock remains the global minimum.

Since the local clocks have disjoint domains and the clock values are totally ordered, this minimum is unique. If the invariant was satisfied for two different nodes $s_1$ and $s_2$, this would imply:

$$s_1.Minimum \wedge s_2.Minimum$$

Then, by the unicity of the minimum, it follows that $s_1 = s_2$ which contradicts the hypothesis.
□

**Liveness Property.** Let $s_0$ be the node with the smallest clock:

$$s_0.C[s_0] = \langle \min s :: s.C[s] \rangle$$

We show through the next three lemmata that the local clock of $s_0$ eventually increases. Next, we show that the global minimum eventually increases. Then we deduce the liveness property.

**Lemma 2.** *If the node $s_0$ is in the mutex state, it eventually ticks:*

$$\forall k :: \ \ s_0.mutex \wedge s_0.C[s_0] = k \ \mapsto \ s_0.C[s_0] > k \tag{4}$$

*Proof.* This is ensured by the weak fairness of the UNITY execution model, which eventually executes the "leave and tick" statement. □

**Lemma 3.** *If the node $s_0$ is in the requesting state, it eventually ticks :*

$$\forall k :: \ \ s_0.requesting \wedge s_0.C[s_0] = k \ \mapsto \ s_0.C[s_0] > k \tag{5}$$

*Proof.* The program $DC$ satisfies:

$$\forall k :: s_0.requesting \wedge s_0.C[s_0] = k \textbf{ unless } s_0.mutex \tag{6}$$

With `OBS_LEADSTO2`, we deduce that if a local clock is greater than $l$, its image eventually becomes greater than $l$:

$$\forall s :: \frac{s_0.C[s] \prec s.C[s], \ \forall m :: \textbf{stable } s.C[s] > m}{\forall l :: s.C[s] > l \mapsto s_0.C[s] > l}$$

Thanks to the PSP (Progress-Safety-Progress) theorem[7], we deduce:

$$\begin{gathered} \forall s, k, l : s \neq s_0 :: \\ s_0.requesting \wedge s_0.C[s_0] = k \wedge s.C[s] > l \\ \mapsto \\ s_0.mutex \vee (s_0.requesting \wedge s_0.C[s_0] = k \wedge s_0.C[s] > l) \end{gathered}$$

[7] PSP Theorem [2]: $\frac{p \mapsto q \ , \ r \textbf{ unless } b}{p \wedge r \ \mapsto \ (q \wedge r) \vee b}$

Let $k = l = s_0.C[s_0]$, then this formula simplifies into:

$$\begin{array}{c}\forall s : s \neq s_0 :: \\ s_0.requesting \mapsto s_0.mutex \vee (s_0.requesting \wedge s_0.C[s_0] < s_0.C[s])\end{array} \tag{7}$$

Using `OBS_STABLE` and `OBS_F`, it follows that $s_0.C[s]$ is non-decreasing. With (6), we prove that:

$$\forall s : s \neq s_0 :: s_0.requesting \wedge s_0.C[s_0] < s_0.C[s] \textbf{ unless } s_0.mutex \tag{8}$$

We can then apply the completion theorem[8] to (7) and (8) and deduce:

$$\begin{array}{c}s_0.requesting \\ \mapsto \\ s_0.mutex \vee (s_0.requesting \wedge \langle \forall s : s \neq s_0 :: s_0.C[s_0] < s_0.C[s] \rangle)\end{array} \tag{9}$$

As the program $DC$ satisfies:

$$s_0.requesting \wedge \langle \forall s : s \neq s_0 :: s_0.C[s_0] < s_0.C[s] \rangle \textbf{ unless } s_0.mutex$$

the instruction "enter and use" is eventually chosen and performed. Consequently, we have:

$$s_0.requesting \wedge \langle \forall s : s \neq s_0 :: s_0.C[s_0] < s_0.C[s] \rangle \;\mapsto\; s_0.mutex$$

We deduce by cancelling[9] the right hand of the disjunction in (9):

$$s_0.requesting \mapsto s_0.mutex \tag{10}$$

And using lemma 2, we conclude:

$$\forall k :: \;\; s_0.requesting \wedge s_0.C[s_0] = k \;\mapsto\; s_0.C[s_0] > k$$

□

**Lemma 4.** *If the node $s_0$ is in the thinking state, it eventually ticks.*

*Proof.* We have (from the program text):

$$\begin{array}{c}\forall k :: s_0.thinking \wedge s_0.C[s_0] = k \\ \mapsto \\ (s_0.thinking \wedge s_0.C[s_0] > k) \vee (s_0.requesting \wedge s_0.C[s_0] = k)\end{array}$$

Using lemma 3 and cancellation, we deduce:

$$\forall k :: \;\; s_0.thinking \wedge s_0.C[s_0] = k \;\mapsto\; s_0.C[s_0] > k \tag{11}$$

□

[8] Completion theorem [2]: $\frac{\langle \forall i :: p_i \mapsto q_i \vee b \rangle \;,\; \langle \forall i :: q_i \textbf{ unless } b \rangle}{\langle \forall i :: p_i \rangle \mapsto \langle \forall i :: q_i \rangle \vee b}$, for any finite set of predicates $p_i$, $q_i$.

[9] Cancellation theorem: $\frac{p \mapsto q \vee b \;,\; b \mapsto r}{p \mapsto q \vee r}$. Here we have $r = q$.

**Lemma 5.** *The global minimum* $\langle \min s :: s.C[s] \rangle$ *is unbounded.*

*Proof.* Applying the leadsto finite disjunction to the three lemmata 2, 3 and 4, we obtain:

$$\forall k :: \; s_0.C[s_0] = k \mapsto s_0.C[s_0] > k$$

Therefore, as $s_0$ was the global minimum, this global minimum is increased:

$$\forall k :: \; \langle \min s :: s.C[s] \rangle = k \mapsto \; \langle \min s :: s.C[s] \rangle > k$$

By induction, we deduce:

$$\forall k, l :: \; \langle \min s :: s.C[s] \rangle = k \mapsto \; \langle \min s :: s.C[s] \rangle > l$$

Using the leadsto infinite disjunction, we finally get:

$$\forall l :: \; \texttt{true} \; \mapsto \; \langle \min s :: s.C[s] \rangle > l \tag{12}$$

□

*Liveness Property.* since the global minimum is unbounded, all clocks are unbounded. Since the clock of a requesting node is stopped unless it enters mutual exclusion, it eventually does. □

### 4.3 Observation Topology

In the preceding algorithm, each node observes all the other nodes. Therefore, the underlying network topology is a mesh. We can also regard this mesh as an observation topology. We can change the observation topology without modifying the algorithm properties. In other words, some topology transformations preserve the algorithms.

For instance, a ring topology may replace the mesh. Each node only observes a single predecessor. This observation ring is specified by:

$$\forall s, s' \; : \; s \neq s' \; :: \; s.C[s'] \prec (pred.s).C[s']$$

A node $Mutex(s)$ only observes its predecessor directly. For the other nodes, it observes their remote clocks through the images provided by its predecessor. For this transformation, thanks to the transitivity of observations, the properties of the solution are preserved. We have again:

$$\forall s, s' :: \; s.C[s'] \prec s'.C[s']$$

At an implementation level, this observation topology leads to message exchanges according to a ring topology.

Finally, a node may also observe simultaneously the entire array of clocks of its predecessor:

$$\forall s \; : \; (s.C[s'] : s' \neq s) \prec \; ((pred.s).C[s'] : s' \neq s)$$

## 5 Conclusion

In this paper, we have introduced the observation relation as an abstraction of the properties of usual underlying networks. After showing how its semantics and syntax could be embedded within the UNITY approach, we have stated general observation properties. Finally, we have considered the description and the validation of a distributed mutual exclusion algorithm. We believe that the observation relation is interesting not only for the description of distributed algorithms since it provides an abstraction of communications, but also for the proof of distributed algorithms since it has interesting properties which should facilitate proof steps.

Among our investigations on observations, let us mention the expression of distributed algorithms. For instance we have designed an observations-based version of Mattern's termination detection algorithm [13]. We are also studying the use of observations as a tool to support the development of distributed algorithms. We are working on the properties which are preserved when an initial algorithm with global variables is transformed into another one where some of these variables are replaced by their images. We plan to validate such transformations within an HOL [9] observation theory and to make them available within a UNITY development environment [5]. A further research aspect concerns the study of the observation of several variables, the crucial point being how the dependence between these variables should be dealt with.

## References


1. Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *Computing Surveys*, 15(1):3–43, March 1983.
2. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
3. M. Charpentier, M. Filali, P. Mauran, G. Padiou, and P. Quéinnec. Répartition par observation dans Unity. Technical Report 96-01-R, IRIT, 27 pages, January 1996.
4. Michel Charpentier. Une sémantique pour Unity avec section observe. Technical report, IRIT, April 1996.
5. Michel Charpentier, Abdellah El Hadri, and Gérard Padiou. A Unity-based algorithm design assistant. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 131–145, Aarhus, Denmark, May 1995. BRICS Notes Series NS-95-2.
6. Vilay K. Garg. *Principles of Distributed Systems*. Kluwer Academic Publisher, 1996.
7. Rob Gerth and Amir Pnueli. Rooting UNITY. In *Fifth Int'l Workshop on Software Specification and Design*, pages 11–19, May 1989.
8. David Moshe Goldschlag. *Mechanically Verifying Concurrent Programs*. PhD thesis, University of Texas at Austin, May 1992.
9. M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

10. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1984.
11. Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
12. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1991.
13. Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.
14. Jayadev Misra. A logic for concurrent programming. Technical report, The University of Texas at Austin, April 1994.
15. G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, January 1981.
16. Beverly A. Sanders. Eliminating the substitution axiom from Unity logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.
17. R.T. Udink and J.N. Kok. On the relation between Unity properties and sequence of states. In J.W de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Semantics: Foundations and Applications,* volume 666 of *Lecture Notes in Computer Science*, pages 594–608, 1993.

This article was processed using the LaTeX macro package with LLNCS style