Parallel Processing Letters Vol. 9, No. 3 (1999) 437–450

THE OBSERVATION: AN ABSTRACT COMMUNICATION MECHANISM

MICHEL CHARPENTIER, MAMOUN FILALI, PHILIPPE MAURAN
GÉRARD PADIOU, PHILIPPE QUÉINNEC
IRIT/INPT
2, rue Charles Camichel, BP 7122
31071 Toulouse cedex 7, France
charpov@cs.caltech.edu, filali@irit.fr, {mauran,padiou,queinnec}@enseeiht.fr

ABSTRACT

In this paper, we introduce an observation relation as an abstraction of point-to-point communication in distributed architectures. After showing how its semantics and syntax can be embedded within the UNITY approach, we state general observation properties. Finally, we consider the description and the validation of a distributed mutual exclusion algorithm. The relevant aspect of such a validation is the exclusive use of refinements and observations properties for the proof of these refinements.

Keywords: distributed algorithms, formal methods, specification, verification, refinements, temporal logic, UNITY

1 Introduction

Designing and reasoning about distributed systems is more often a matter of ingenuity than 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:

- 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 [CM88,Mis94] which provides a sound formal framework [GP89,San91,Gol92] and appears to be well suited to describing distributed systems. Our contribution is to provide 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 —from our point of view— the essential feature of distributed systems: the (a priori) impossibility for a site to maintain 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 properties of images to properties of sources on the one hand, and to link the observation relation to UNITY logical operators on the other.

Thus, a first interest of this construct is to provide a set of proof rules that can be used for reasoning about a wide range of distributed algorithms. A second contribution is to offer a declarative and progressive approach to distributed algorithm 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 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 refine and prove a distributed mutual exclusion algorithm. The last section concludes and describes our current work.

2 UNITY

The purpose of the UNITY formalism [CM88,Mis94] 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). The purpose of this formalism is 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 simplicity of use.

The UNITY formalism consists of two parts: a programming language, based on fair transition systems, and a specification language, based on a linear temporal logic [MP92], to express safety and progress properties of the corresponding transition systems.

This section describes the parts of UNITY syntax and semantics that are useful in understanding the remainder of the paper. A more comprehensive presentation can be found in [CM88] or [Mis94].

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-assignment 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 $F.init$, expressing the possible initial states for the program;
- the **assign** section, denoted by $F.assign$, consists of a set of multiple guarded assignments separated by the $[\!]$ mark.

  UNITY also allows quantification over statements to describe a finite family of assignments, written as $\langle [\!] v : condition :: statements(v) \rangle$.

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. However, when executed with its guard not satisfied, a statement leaves the state unchanged.

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 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 σ. For each sequence σ, the state σ_i is a mapping between the state variables and their value. In this paper, the value of variable v in state σ_i is denoted by $\sigma_i.v$. By extension, $\sigma.v$ represents the successive values of the variable v in the computation σ, and $\sigma_i.e$ denotes the value of the *expression* e in the state σ_i.

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:

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

- `stable` p : once p becomes true, it remains true[a]:

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

- `invariant` p : p is true in all states[b]:

$$\texttt{invariant } p \texttt{ in } F \equiv \langle \forall \sigma, i : \sigma \in \mathcal{O}.F \ :: \ \sigma_i.p \rangle$$

- $p \mapsto q$: (*leads-to* operator) if p is true, then q will eventually become true:

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

- p `detects` q : if p is true, then q is (already) true, and if q becomes true, then p will eventually detect it:

$$p \texttt{ detects } q \equiv (\texttt{invariant } p \Rightarrow q) \wedge (q \ \mapsto \ p)$$

Note that *unless*, *stable* and *invariant* are *safety* operators, while *leads-to* is a *liveness* operator and *detects* mixes both safety and liveness.

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 [AS83,Gar96,Hoa84].

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

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

Channels are introduced to enable communication between these processes. To represent a channel, the most suitable abstract data type is the sequence. The enqueue operation, denoted by $\bullet$, implements a message sending and a head extraction simulates a message reception. A pair of processes (`Sender`, `Receiver`) communicating through a channel `C`, can be described by:

$$\texttt{DS} = \texttt{Sender} \ [\!] \ \texttt{Receiver}$$

[a] `stable` p is equivalent to p `unless` $false$.
[b] `invariant` p is equivalent to : $(init \Rightarrow p) \wedge (\texttt{stable } p)$.

```
Program Sender                Program Receiver
Declare^c                     Declare
    C : sequence(T);              C : sequence(T);
    em : T                        rec : T
Assign                        Assign
    C := C • em                   rec, C := head(C), tail(C)  if ¬empty(C)
[]   ...                      []   ...
End                           End
```

Another model consists in describing the communication medium explicitly. Each process owns ports, and a dedicated "medium process" transfers the messages from output ports to input ports. This approach allows a fine grained description of the underlying communication protocol [CP98]. This model can be easily described in UNITY: as for channels, sequences represent ports; the send primitive on an output port is translated into an enqueuing operation, the receive primitive on an input port is translated into a dequeuing operation; the medium component performs transfers between connected ports.

Both approaches share a common feature: communication is described by shared sequence variables. However, these models do not provide any abstraction with respect to the physical model. For instance, message loss and duplication have to be programmed explicitly. In this paper, we propose a more abstract view of communication, that is a closer fit to distributed architectures.

3 Communication by Observation

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 (but not necessarily a prefix) of the (remote) variable $x$. The variable $x$ (resp. $`x$) is called the *source* (resp. the *image*) of the relation.

The benefits of using a logical operator to describe communication can be combined with the traditional message passing paradigm. See, for instance, [Siv97] where an operator similar to our observation is used to state relationships between input and output ports in a message-based communication. In the message passing paradigm, communication and synchronization cannot be described separately. Observation aims at describing only communication (values transmission). If a specific scheme of synchronization is required, it can be explicitly programmed. For instance, a handshake protocol can be implemented with two observations [CFM$^+$98].

3.1 Clock Functions

Observation relies upon an abstract model of time. We assume that an ideal discrete totally ordered time exists. In the following, such a time is represented by

[c] In UNITY, when a variable is declared in several programs with the same type, these declarations refer to the same shared variable.

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.

$Clock.C$ means that C is an abstract clock, i.e. a mapping from natural numbers to natural numbers such that:

1. $\langle \forall t :: C.t \leqslant t \rangle$: an abstract clock never outgrows the ideal time;
2. $\langle \forall t :: C.t \leqslant C.(t+1) \rangle$: an abstract clock is non-decreasing;
3. $\langle \forall k :: \langle \exists t :: C.t > k \rangle \rangle$: an abstract clock eventually increases.

3.2 The Observation Relation $\prec$

The observation relation between two variables of a program F is defined by:

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

The relation $`v \prec v$ means that, at any time $t$, the image variable $`v$ has the value the source variable v had at time $C.t$. Therefore, the relation states that any value of $`v$ is a previous value of $v$ (clock property 1), that $`v$ is assigned its values in chronological order (clock property 2) and that $`v$ is eventually assigned more recent values of $v$ (clock property 3), even though the delay between $v$ changes and $`v$ updates is unbounded and some changes of v may be lost.

We generalize the definition to take into account any expression defined on the variables, or on the state of the program. When these expressions are boolean valued (i.e. predicates), observation can be considered as a new temporal operator in the UNITY logic.

3.3 Properties

The observation relation has its own properties and properties related to other UNITY logical relations. These properties are valide for any UNITY operational semantic model $\mathcal{O}$. They are proven in [Cha97b].

Observation properties:

- The observation relation is reflexive, antisymmetric and transitive:

$$\frac{}{u \prec u} \qquad \frac{u \prec v \ , \ v \prec u}{\texttt{invariant } u = v} \qquad \frac{u \prec v \ , \ v \prec w}{u \prec w}$$

- The observation relation is compatible with function application:

$$\frac{u \prec v}{f.u \prec f.v} \quad (2) \qquad\qquad \frac{f.u \prec f.v \ , \ f \text{ injection}}{u \prec v} \quad (3)$$

Relationship between observation and the UNITY *logic*:

$$\frac{`p \prec p \ , \ \texttt{stable } p}{\texttt{stable } `p} \quad (4) \qquad\qquad \frac{`p \prec p \ , \ \texttt{stable } p}{`p \texttt{ detects } p} \quad (5)$$

$$\frac{`p \prec p\ ,\ p \texttt{ unless } `p}{p \mapsto `p} \qquad (6) \qquad\qquad \frac{`p \prec p\ ,\ \texttt{stable } p}{p \mapsto `p} \qquad (7)$$

$$\frac{`u \prec u\ ,\ \forall k :: \texttt{stable } u \geqslant k^{\mathrm{d}}}{\forall k :: \texttt{stable } `u \geqslant k} \qquad (8)$$

$$\frac{`u \prec u\ ,\ \forall k :: \texttt{stable } u \geqslant k}{\texttt{invariant } `u \leqslant u} \qquad (9)$$

$$\frac{`u \prec u\ ,\ \forall k :: \texttt{stable } u \geqslant k}{\forall l :: u = l \mapsto `u \geqslant l} \qquad (10)$$

Theorem (4) states that the image of a stable predicate is stable and (5) states that a stable predicate can be detected by its image. Theorems (6) and (7) ensure the occurrence of $`p$ given some properties of p.

Theorem (8) states that the image of a counter is a counter, (9) that the image of a counter is always smaller than its source and (10) shows that the image of a counter will eventually become greater or equal than any value that the counter had. Note that properties (6), (7) and (10) allow to deduce a liveness UNITY property form a safety UNITY property and the liveness of observation.

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

- either to deduce images properties from environment properties, e.g. (4),
- or to deduce environment properties from images properties, e.g. (5).

3.4 Observations as a Mechanism

In the preceding section, observation is defined as a relation between variables (or expressions) and we have stated several theorems relating this relation to other UNITY temporal operators. We now intend to use observations to avoid an explicit description of the communication between the nodes of a distributed system. An observation *mechanism* is defined. This mechanism is part of a program description. It replaces, in a more abstract way, the program part normally devoted to communication description.

In order to avoid difficulties related to the composition of program texts with logically defined components (communication components defined with observation relations), we choose to extend the UNITY language and have the communication description as part of program texts. A new section, named *observe*, is introduced in UNITY programs. It is made of assumed observation *relations* between some variables of the program. To consider the *observe* section of a program at the semantic level, a new operational semantic model is defined. It allows image variables

[d] The UNITY formula $\forall k :: \texttt{stable } u \geqslant k$ expresses that u is a non-decreasing counter (or, in short, a counter).

to change their value while program assignments are executed, in accordance with the stated observation relations. This new semantic model, called $\mathcal{O}bs$, is defined as follows:

- $F.Img$ is the set of the image variables of F, i.e. the variables in the left-hand side of one observation relation. $F.NImg$ is the set of the variables of F that are *not* an image in any *observe* relation, i.e. the variables assigned by the program statements. The variables of the program F are partitioned:

$$F.Img \cup F.NImg = F.declare \quad \wedge \quad F.Img \cap F.NImg = \emptyset$$

- $\mathcal{O}_{assign}.F$ is the set of the computations σ such that:
  - $\sigma_0.F.init$,
  - $\langle \forall t :: \langle \exists s : s \in F.assign :: \sigma_t \xrightarrow{s} \sigma_{t+1} \rangle \rangle$,
  - $\langle \forall s : s \in F.assign :: \langle \forall t :: \langle \exists t' : t' > t :: \sigma_{t'} \xrightarrow{s} \sigma_{t'+1} \rangle \rangle \rangle$.

  where $\sigma_t \xrightarrow{s} \sigma_{t+1} \;\equiv\; \langle \forall x : x \in F.NImg :: \sigma_{t+1}.x = (s.\sigma_t).x \rangle$[e];

- $\mathcal{O}_{observe}.F$ is the set of the computations σ such that:

  $\langle \forall x : x \in F.Img \wedge (x \prec y) \in F.observe :: \langle \exists C : Clock.C :: \langle \forall t :: \sigma_t.x = \sigma_{C.t}.y \rangle \rangle \rangle$;

- $\mathcal{O}bs.F = \mathcal{O}_{assign}.F \bigcap \mathcal{O}_{observe}.F$.

On the one hand, $\mathcal{O}_{assign}$ describes the states of the variables as determined by the program assignments with respect to the weak fairness. The statements are executed on a complete program state (with the image variables), but the images are free to be assigned any value. On the other hand, $\mathcal{O}_{observe}$ describes the states of the images determined by the observation relations of the *observe* section. The operational model $\mathcal{O}bs$ is made of the computations satisfying both constraints.

4 Using Observations to Refine Communication

The example we consider is a mutual exclusion algorithm similar to the algorithm described by E.W. Dijkstra in [Dij74]. The system consists of N identical nodes distributed on a logical ring. Each node manages an increasing counter so that there is always a single node with a counter value smaller than the counter of its predecessor on the ring. The uniqueness of this edge in the counter values ensures the mutual exclusion property. The fairness of the algorithm is guaranteed by the circulation of this edge around the ring.

To decide if it has the privilege (i.e. if it is on the edge), a node must know the counter value of the previous node on the ring. From a centralized point of view, we can assume that each node always knows instantaneously the counter value of its predecessor. However, in a distributed context, a communication between a node

[e] $s.\sigma_t$ denotes the image of the state σ_t by the statement s seen as a mapping.

and its predecessor is necessary. In this section, we show that there exists a refinement relation between the abstract system where each node has an instantaneous vision of its predecessor and the concrete system where the nodes use observations to communicate.

4.1 The Abstract System

The program P that describes the global system is a composition by union of N identical programs $Node_i$:

$$P \ = \ \langle [\!] i : 0 \leqslant i < N :: Node_i \rangle$$

Each node has the following behaviour:

Program $Node_i$
Declare $Req_i, Excl_i : bool; \ c_i : int;$
Initially $c_i = i \land \neg Excl_i \land \neg Req_i$
Assign -- requesting the critical section access
$Req_i := true$
$[\!]$ -- Entering the critical section when requesting and being on the edge
$Excl_i := true$ `if` $Req_i \land \neg Excl_i$[f] $\land\, c_i < c_{i-1}$[g]
$[\!]$ -- leaving the critical section, passing the edge to the next node
$Excl_i, Req_i, c_i := false, false, c_i + N$ `if` $Excl_i$
$[\!]$ -- passing the edge to the next node when no request is pending
$c_i := c_i + N$ `if` $\neg Req_i \land c_i < c_{i-1}$
End $Node_i$

The system properties we are interested in are:

safety: `invariant` $\langle \forall i, j :: Excl_i \land Excl_j \Rightarrow i = j \rangle$
liveness: $\forall i :: Req_i \mapsto Excl_i$

The correctness of this solution is based on the following arguments:

- for safety, we rely on the inductive invariant:

$$\langle \forall i :: (excl_i \Rightarrow c_i < c_{i-1} \land req_i) \land (c_i = c_{i-1} + 1 \ \lor \ c_i = c_{i-1} - N + 1) \land (c_i < c_{i-1} \Rightarrow c_i = \langle \min j :: c_j \rangle) \rangle$$

- for liveness, we rely on the metrics defined as the number of hops to the node with the smallest counter:

$$M(j) = d(j, i \mid c_i = \langle \min k :: c_k \rangle)$$

  where d is the number of clock-wise hops between i and j:

$$d(i,j) = \texttt{if } i < j \texttt{ then } j - i \texttt{ else } N - (i - j)$$

---

[f]The conjunction $\neg Excl_i$ is only useful for the proof. If $Excl_i$ is already *true*, the statement has no effect. Therefore, $\neg Excl_i$ can be omitted in the guard.

[g]The notation "-1" refers to the predecessor on the ring. More precisely, $i-1$ has to be understood as $i - 1 \bmod N$.

4.2 The Concrete System

In the abstract system, the second and fourth statements of the node i use the value c_{i-1} of the counter of the node $i-1$. Such a knowledge is unrealistic in a distributed system. We now consider a distributed description of the system where observation is used by a node to know the counter value of its predecessor:

Program	$Node'_i$
Declare	$Req_i, Excl_i : bool;\; c_i, t_i : int;$
Observe	$\boxed{t_i \prec c_{i-1}}$
Initially	$c_i = i \wedge \neg Excl_i \wedge \neg Req_i$
Assign	$Req_i := true$
$\|$	$Excl_i := true \texttt{ if } Req_i \wedge \neg Excl_i \boxed{\wedge\, c_i < t_i}$
$\|$	$Excl_i, Req_i, c_i := false, false, c_i + N \texttt{ if } Excl_i$
$\|$	$c_i := c_i + N \texttt{ if } \neg Req_i \boxed{\wedge\, c_i < t_i}$
End	$Node'_i$

The only differences between $Node_i$ and $Node'_i$ are the introduction of the *observe* section and that $c_i < t_i$ replaces $c_i < c_{i-1}$ in the second and fourth statement guards.

The distributed system is the composition by union of the programs $Node'_i$:

$$P' \;=\; \langle [\!]\, i : 0 \leqslant i < N :: Node'_i \rangle$$

In the remainder of this section, we prove that the system P' refines the system P. Therefore, the mutual exclusion property and the fairness of the algorithm can be proved on P without considering the communication part of the system.

4.3 Proof by Refinements

Different definitions of refinement exist for UNITY programs [Sin93]. One of the most obvious is simply the preservation of all UNITY properties (except *ensures* properties which are not defined operationally). A program F is refined by a program F' (or F' refines F, or F' is a refinement of F) and we note $F \sqsubseteq F'$ iff any UNITY property of F is satisfied by F'.

To prove that the system P' refines the system P, we use three successive refinements: superposition to add the *observe* section; guards strengthening to constrain the guards of the second and fourth statements with $c_i < t_i$; substitution to remove the now useless conjunction $c_i < c_{i-1}$.

Refinement by superposition: The superposition described in [CM88] is a transformation of UNITY programs that enjoys the refinement property. The superposition proceeds by introducing new variables (here t_i) and then transforming the underlying program such that the assignments to underlying variables remain unaffected (here we add the *observe* section). The theorem states that every property of the underlying program is a property of the transformed program.

Refinement by guards strengthening: Guards strengthening states that the guard p of a statement τ in a UNITY program can be replaced by $p \wedge q$ without loss of any (safety or liveness) property provided the following hold:

$$q \; \texttt{unless} \; \neg p \qquad\qquad p \;\mapsto\; q$$

One important point is to precise *where* (i.e. in which program) these two properties should hold. The theorem has been stated for UNITY axiomatic semantics with the two properties being proved on the program without the statement τ [CM88,Sin93]. However, it is proved in [Cha97b] that the theorem is valid for UNITY *operational* semantics when both properties are proved on the complete program *after* the strengthening.

Refinement by substitution: The refinement by substitution states that if a program F satisfies `invariant` I, then I can be substituted by *true* in the program, giving a new program which refines F.

We now detail the proof for the transformation of the second statement guard. The case of the fourth statement guard is exactly the same.

We strengthen the guard of the second statement of each program $Node_i$ with the conjunction $c_i < t_i$ to obtain the programs $Node''_i$ where the second statement is:

$$Excl_i := true \;\texttt{if}\; Req_i \wedge \neg Excl_i \wedge c_i < c_{i-1} \wedge c_i < t_i$$

P^i denotes the system where this transformation is applied to the nodes between 0 and i, the other nodes remaining unchanged:

$$P^i \;=\; \langle [\!] j : 0 \leqslant j \leqslant i :: Node''_j \rangle \;[\!]\; \langle [\!] j : i < j < N :: Node_j \rangle$$

The system P^i is derived from the system P^{i-1} by strengthening the guard of the second statement of $Node_i$ with $c_i < t_i$. This transformation is a refinement provided the following properties hold in P^i:

$$c_i < t_i \;\texttt{unless}\; \neg Req_i \vee Excl_i \vee c_i \geqslant c_{i-1} \;\texttt{in}\; P^i \tag{11}$$

$$Req_i \wedge \neg Excl_i \wedge c_i < c_{i-1} \;\mapsto\; c_i < t_i \;\texttt{in}\; P^i \tag{12}$$

The safety property (11) is easy and derives directly from the program text. Due to (8), t_i is a non-decreasing counter. Therefore, the only way to falsify the left-hand-side of the *unless* is to change the value of c_i. This is only possible with the statements three and four of program $Node''_i$, i.e. with either $Excl_i$ or $\neg Req_i$. In both cases, the right-hand-side of the *unless* is *true* and (11) is satisfied.

The most interesting property is the liveness property (12). Intuitively, the property comes from the fact that c_i cannot change before the right-hand-side of the *leads-to* is *true*, and from the liveness of observation.

As we have said before, c_i cannot change as long as Req_i is *true* and $Excl_i$ is *false*, i.e. before the guard of the second statement is *true*. More precisely, the program P^i satisfies the property:

$$\forall l :: Req_i \wedge \neg Excl_i \wedge c_i < c_{i-1} \wedge c_i = l \;\texttt{unless}\; c_i < t_i \tag{13}$$

On the other hand, since c_{i-1} is a non-decreasing counter, the theorem (10) asserts that:

$$\forall k :: c_{i-1} = k \;\mapsto\; t_i \geqslant k \tag{14}$$

Applying the PSP (Progress-Safety-Progress) theorem[h] to (13) and (14), we get:

$$\begin{array}{c} \forall k, l :: c_{i-1} = k \wedge Req_i \wedge \neg Excl_i \wedge c_i < c_{i-1} \wedge c_i = l \\ \mapsto \\ (t_i \geqslant k \wedge Req_i \wedge \neg Excl_i \wedge c_i < c_{i-1} \wedge c_i = l) \vee c_i < t_i \end{array} \tag{15}$$

Now, we simply derive (12):

$$\begin{array}{ll} & Req_i \wedge \neg Excl_i \wedge c_i < c_{i-1} \\ \Rightarrow & \quad \{\text{predicate calculus}\} \\ & \langle \exists k, l :: k > l \wedge c_i = l \wedge c_{i-1} = k \wedge Req_i \wedge \neg Excl_i \wedge c_i < c_{i-1} \rangle \\ \mapsto & \quad \{(15), \text{disjunction theorem}\} \\ & \langle \exists k, l :: (\underline{k > l} \wedge \underline{c_i = l} \wedge \underline{t_i \geqslant k} \wedge Req_i \wedge \neg Excl_i \wedge c_i < c_{i-1}) \vee c_i < t_i \rangle \\ \Rightarrow & \quad \{\text{predicate calculus}\} \\ & c_i < t_i \end{array}$$

This completes the proof of (12).

As the two conditions (11) and (12) are satisfied, we deduce the refinement relation:

$$\forall i : 0 < i < N :: P^{i-1} \sqsubseteq P^i$$

Due to the transitivity of refinement, we obtain:

$$P^0 \;\sqsubseteq\; P^{N-1}$$

The system P^0 is exactly the system P. In the system P^{N-1}, the second statement of each node is guarded with $Req_i \wedge \neg Excl_i \wedge c_i < c_{i-1} \wedge c_i < t_i$, i.e. with $Req_i \wedge \neg Excl_i \wedge c_i < t_i \wedge (c_i < t_i \Rightarrow c_i < c_{i-1})$. Since all c_i are non-decreasing counters, and thanks to the theorem (9), we know that $t_i \leqslant c_{i-1}$ (and consequently $c_i < t_i \Rightarrow c_i < c_{i-1}$) is invariant in P^{N-1}. Therefore, the conjunction $c_i < t_i \Rightarrow c_i < c_{i-1}$ can be removed from the guards, and the second statements of nodes are equivalent in P^{N-1} and P'.

Applying the same reasoning to the fourth statements of nodes, we finally get the refinement relation:

$$P \;\sqsubseteq\; P'$$

[h]PSP Theorem [CM88]: $\frac{p \mapsto q \;,\; r \;\texttt{unless}\; b}{p \wedge r \;\mapsto\; (q \wedge r) \vee b}$

5 Conclusions

In this paper, we have introduced the observation relation as an abstraction of point-to-point communication in distributed systems. 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. Starting with a solution where communication was instantaneous, we have derived a solution where communication is done through observations, which is a rational abstraction of usual underlying networks. The interesting point in this validation is the fact that the proof is achieved entirely through refinement and that the refinement steps are proved with the observation proof rules.

We claim that observation can provide a way to describe communication in distributed algorithms while avoiding most of the drawbacks resulting from more traditional descriptions:

- Observation provides a more abstract description. It allows the representation of communication between components without describing explicitly the manner in which this communication is actually achieved.
- The resulting communication description is more consonant with UNITY (and UNITY-like) formalisms. Observation can be seen as a weakening of UNITY shared-variable paradigm. Moreover, the observation relation is well integrated into the UNITY logic. Observation can be seen as a new temporal operator, related to traditional UNITY operators through additional proof rules. This embedding of communication within UNITY makes refinement-based proofs much easier.
- Lastly, observation allows to deal *only* with value transmission aspects, neglecting any synchronization concerns.

However, communication, as important as it is, is only one aspect of distributed computing. Other aspects, such as true concurrency have been studied with a similar approach [Cha97a]. Others, such as synchronization or fault tolerance are currently being investigated.

References

[AS83] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *Computing Surveys*, 15(1):3–43, March 1983.

[CFM+98] Michel Charpentier, Mamoun Filali, Philippe Mauran, Gérard Padiou, and Philippe Quéinnec. Tailoring UNITY to distributed program design. In José Rolim, editor, *Int'l workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'98)*, volume 1388 of *Lecture Notes in Computer Science*, pages 820–832, Orlando, April 1998. Springer-Verlag.

[Cha97a] Michel Charpentier. A UNITY mapping operator for distributed programs. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *4th Int'l Symposium of Formal*

Methods Europe (FME'97), volume 1313 of *Lecture Notes in Computer Science*, pages 665–684. Springer-Verlag, September 1997.

[Cha97b] Michel Charpentier. *Assistance à la Répartition de Systèmes Réactifs*. Thèse de doctorat, Institut National Polytechnique de Toulouse, France, November 1997.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[CP98] Michel Charpentier and Gérard Padiou. Specification and verification of the ATMR protocol using UNITY. *Parallel Processing Letters*, 8(4):421–445, 1998.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[Gar96] Vilay K. Garg. *Principles of Distributed Systems*. Kluwer Academic Publisher, 1996.

[Gol92] David Moshe Goldschlag. *Mechanically Verifying Concurrent Programs*. PhD thesis, University of Texas at Austin, May 1992.

[GP89] Rob Gerth and Amir Pnueli. Rooting UNITY. In *5th Int'l Workshop on Software Specification and Design*, pages 11–19, May 1989.

[Hoa84] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1984.

[Mis94] Jayadev Misra. A logic for concurrent programming. Technical report, The University of Texas at Austin, April 1994.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[San91] Beverly A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, April–June 1991.

[Sin93] Ambuj K. Singh. Program refinement in fair transition systems. *Acta Informatica*, 30(6):503–535, May 1993.

[Siv97] Paolo A. G. Sivilotti. *A Method for the Specification, Composition, and Testing of Distributed Object Systems*. PhD thesis, California Institute of Technology, December 1997.