Tailoring UNITY to Distributed Program Design*

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

IRIT / INPT-ENSEEIHT
2, rue Charles Camichel, BP 7122
F-31071 Toulouse cedex 7, France
e-mail: {charpov,mauran,padiou,queinnec}@enseeiht.fr, filali@irit.fr

Abstract. As a general framework, UNITY does not offer any specific facility for the design of distributed systems. For such systems, distribution aspects must be represented at a low level, resulting into intricated models and proofs. To provide a more abstract view of distributed systems, we propose two extensions to UNITY. The first one is an *observation relation* which is integrated in UNITY semantics to provide an abstract communication mechanism. The second one is a *mapping operator* which accounts for the true parallelism of distributed systems. The paper illustrates, through different examples, how these extensions can be used to help the design of distributed systems in UNITY.

1 Introduction

UNITY is intended to be a general purpose formalism. It aims at supporting the development of correct and efficient (parallel) programs. The UNITY formalism is designed so that this goal can be fulfilled in two stages:

- the development of a correct abstract program;
- the transformation by refinement of this abstract program into a concrete one, that integrates the specific constraints and features of a given architecture and environment. Efficiency and complexity concerns are dealt with at this stage.

The UNITY formalism is built on this decomposition of the programming task. A major contribution of the formalism consists in focusing on the essential features of a program while reasoning about its correctness. Thus, the aspects of programming related to the mapping of a program to an architecture are postponed to a second stage. Also, the description of the control flow is reduced to a mere fair selection mechanism over a set of assignments. Thus still, proof rules are designed to allow the programmer to easily go back and forth between the program and its specification. Lastly, the preferred approach to program development is refinement, which allows program designers to remain within an homogeneous framework, and avoids any translation between ad-hoc formalisms.

* International workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'98), April 1998, LNCS 1388, pages 820–832.

A drawback of this approach which favors a simple description of abstract programs is that it does not provide any specific facility to deal with distributed systems: describing the system involves lots of details (specification of loci of control and of control flow, implementation constraints, ...). Although the UNITY formalism is expressive enough to cope with the specification and proof of a low-level system, this specification might not be very handy.

If we want to facilitate the use of UNITY for the design of distributed programs, we are faced with a classical dilemma: how to obtain more expressiveness without hampering reusability and generality? This paper aims at providing an answer to this dilemma by introducing two constructs in the UNITY framework to model fundamental distributed features:

- the *observation* relation is an abstract statement of asynchronous and unreliable communications in a distributed system;
- the *product* is a program composition operator closely related to union which allows simultaneous executions of statements by different processes.

2 Designing Distributed Systems in UNITY

At the physical level, a distributed system consists in a set of processes exchanging messages through a communication medium. Several distributed systems and languages such as Mach, Chorus or SR [1, 15, 2] describe communication by means of channels or ports. A channel provides a point-to-point link between a sender and a receiver. This notion is an elementary mechanism to describe communicating processes. It provides an asynchronous unidirectional communication protocol through an unbounded buffer.

In UNITY, a parallel system is described as a union of programs. If these programs share no variables, they are actually distributed processes. Channels are introduced to enable communication between these processes. To represent a channel, the most suitable abstract data type is the sequence. The append operation implements a message sending and a head extraction simulates a message reception. A pair of processes $(\mathtt{Sender}, \mathtt{Receiver})$ communicating through a channel $\mathtt{C}$, can be described by:

$$\mathtt{DS} = \mathtt{Sender} \,[\!]\, \mathtt{Receiver}$$

Program $\mathtt{Sender}$	**Program** $\mathtt{Receiver}$
Declare[1]	**Declare**
$\quad\mathtt{C} : \mathtt{sequence(T)};$	$\quad\mathtt{C} : \mathtt{sequence(T)};$
$\quad\mathtt{em} : \mathtt{T}$	$\quad\mathtt{rec} : \mathtt{T}$
Assign	**Assign**
$\quad\mathtt{C} := \mathtt{C}; \mathtt{em}$	$\quad\mathtt{rec}, \mathtt{C} := \mathtt{head(C)}, \mathtt{tail(C)}\ \mathtt{if}\ \neg\mathtt{empty(C)}$
$[\!]\ \cdots$	$[\!]\ \cdots$
End	**End**

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

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 [9]. This model can be easily described in UNITY: as for channels, a sequence represents a port. The send primitive on an output port is translated into an append operation and the receive primitive on an input port is translated into a head extraction.

Both approaches share a common feature: communication is described by shared sequence variables. However, this model does not provide any abstraction with respect to the physical model. In this paper, we propose a more abstract view of communication, fitted to distributed architectures[2].

3 Extending UNITY for Distributed Programs

In this section, we propose two extensions to UNITY. The first extension is an abstraction of communication called *observation*. The second extension is the definition of a mapping operator for distributed architectures called *product*. Both extensions aim at a better description of distributed programs (e.g. by allowing concurrent evolution of communications and transitions, and simultaneous execution of statements by different processes) and at easier proofs (by abstracting communications, and by reducing proofs of a composed-by-product program to proofs of a composed-by-union program).

3.1 Communication by Observation

Observation Relation. The observation relation describes the relationship between a *source* variable and an *image* variable in a program. It is defined as a temporal operator based on any operational semantic model of UNITY.

Given two variables v and $`v$ of a program $F$ and an operational semantic model $\mathcal{O}$, the relation $`v \prec v$ (read $`v$ *observes* v) is defined as follows:

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

$\mathit{Clock}(C)$ 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.

[2] In [4], Chandy and Misra propose a more ambitious approach. They define sufficient constraints (called monotonicity and commutativity) on the use of shared variables so that such variables could eventually be refined into channels. Provided these constraints are met, their proposition provides a high level of abstraction, as the designer of a distributed system is allowed to reason with the type semantics of these variables.

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

The observation relation has its own properties and properties related to other UNITY logical operators. For instance, the observation relation is reflexive, antisymmetric and transitive; the image of a stable predicate is stable and eventually becomes true if the predicate becomes true. More properties and their proofs can be found in [7].

Observation Mechanism. The observation *mechanism* is part of a program description. A new section, named *observe*, is introduced. It is made of assumed observation *relations* between some variables of the program. This mechanism is used to represent communication between processes.

To consider the *observe* section of a program at the semantic level, a new operational semantic model is defined. It allows image variables to change their value while a program assignment is executed, in accordance with the 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 $\sigma_t \xrightarrow{s} \sigma_{t+1} \;\equiv\; \langle \forall x : x \in F.NImg :: \sigma_{t+1}.x = (s.\sigma_t).x \rangle^3$;
- $\mathcal{O}_{observe}.F$ is the set of the computations σ such that:

  $$\langle \forall\, x \in F.Img : x \prec y \in F.observe :: \langle \exists C : Clock(C) :: \langle \forall t :: \sigma_t.x = \sigma_{C.t}.y \rangle \rangle \rangle;$$
- $\mathcal{O}bs.F = \mathcal{O}_{assign}.F \cap \mathcal{O}_{observe}.F$.

On the one hand, $\mathcal{O}_{assign}$ describes the states of the variables 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.

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

Communication Representation. Communication between processes is achieved by stating, in the *observe* section of a UNITY process, some observation relations with their sources located in other processes.

The composition by union is extended to handle UNITY programs with an *observe* section. The different *observe* sections are gathered to become the *observe* section of the composed program: $(F[\!]G).observe \;=\; F.observe \cup G.observe$

The variables of a program using an observation-based communication model can be read by all processes but can only be assigned by *one* process (including its observation mechanism). Such variables are called *distributed variables* [14], *read variables* [13], or *owned variables* [11].

Observations and Histories. The use of auxiliary variables to prove the correctness of programs is a well-known technique. Among these auxiliary variables, history variables play an important role.

In [13], the history of a program variable x is defined as the sequence variable which records the successive different values of x. Within the Unity formalism, such a variable $\widehat{x}$ satisfies the following predicates:

- **Initially** $\widehat{x} = [x]$
- $x, \widehat{x} = X, H$ `next` $x, \widehat{x} = X, H \vee (x \neq X \wedge \; \widehat{x} = H; x)$

In the following, $x \; \mathcal{H} \; \widehat{x}$ denotes the history relation between two variables x and $\widehat{x}$. When a variable x and a variable $\widehat{x}$ are used, the relation $x \; \mathcal{H} \; \widehat{x}$ is assumed.

In order to reason about programs and reuse the properties of histories and observations, we establish links between these two relations. We use the following rules:

- source and image histories: if we consider the extracted list relation noted $\sqsubseteq$ and defined by: $\begin{cases} l \sqsubseteq nil & = \; (l = nil) \\ nil \sqsubseteq l & \\ h; t \sqsubseteq h'; t' & = \; ((t = t') \wedge (h \sqsubseteq h')) \; \vee \; h; t \sqsubseteq h' \end{cases}$
then we have:
$$i \prec s \Rightarrow \texttt{invariant} \; \widehat{\imath} \sqsubseteq \widehat{s}$$
Such a relation allows to reason about finite sequences instead of the infinite sequences of program executions.
- image evolution: this property describes the next value of an image when the source history is a suffix of the image history[4]:
$$i \prec s \implies (i = I) \; \wedge \; (\widehat{s} = \widehat{\imath}@L) \; \texttt{next} \; (i = I) \; \vee \; \texttt{member}(i, L) \; \vee \; (i = s)$$

3.2 Mapping of Distributed Programs

The distributed application built from the processes described by programs P_i is represented by the program $\langle [\!] i :: P_i \rangle$. However, due to the semantics of UNITY, the statements are mutually exclusive, even when they come from distinct processes P_i. When

[4] The @ operator denotes the list concatenation.

mapping such a program to a distributed architecture, different statements from different UNITY processes are located on different real processors. Then, in the physical world, mutual exclusion is generally not guaranteed between these statements.

With observations, communication is strictly concurrent with internal computations and needs no additional transitions. Therefore, this communication model is compatible with the concurrency due to the mapping to a distributed architecture. In this section, a new composition operation is defined. It stems from the composition by union but handles simultaneous execution of statements of different processes. Therefore, this composition operator maps correctly on distributed architectures. The idea of the new composition is to represent the possible simultaneity of two statements by additional synchronous statements. Each additional statement takes place for two or more assignments executing at the same time.

Definition of the Product Operator. A new operator for program composition, called *product* and denoted by $[\![\!]\!]$, is defined.

A composite program $\langle [\!]\, i :: P_i \rangle$ is said to be *well distributed* if the P_i do not share any variables. The observation mechanism is the only means of communication.

We now assume that every UNITY program has the statement *fair* $:= \neg$*fair*. The boolean variable *fair* is not considered to be a variable of any process. It is used to break the weak fairness of UNITY as simultaneous execution of statements must not be *required* to occur infinitely often.

Given two UNITY programs F and G, possibly extended with an *observe* section, and such that $F [\!] G$ is well-distributed, the product $F [\![\!]\!] G$ is defined as follows:

- $(F [\![\!]\!] G).\mathit{assign} = (F [\!] G).\mathit{assign}$
  $\cup \{(s_F \| s_G)\ \texttt{if}\ \mathit{fair}$[5] $\mid s_F \in F.\mathit{assign} \wedge s_G \in G.\mathit{assign}\}$
- the other sections of $F [\![\!]\!] G$ are the same as in $F [\!] G$.

Because $F [\!] G$ is well-distributed, the programs F and G do not share any variable outside the *observe* sections. Therefore, the synchronous statements $s_F \| s_G$ are correct (non conflicting) UNITY statements.

Converting Union into Product. The composition by product allows simultaneity of execution in the description of a distributed application. On the one hand, using this form of composition may facilitate the mapping step. On the other hand, to prove properties on the program $F [\![\!]\!] G$, all the additional statements of the form $s_F \| s_G$ must a priori be considered. It seems interesting to have a way to introduce product operations at the end of the design and without making all the additional statements explicit. This is achieved by characterizing, for a well-distributed program composed by union with an observation-based communication model, the UNITY properties that still hold when the composition by product is used.

Let us consider F and G, two UNITY programs with *observe* sections such that $F [\!] G$ is well-distributed (the product $F [\![\!]\!] G$ is defined). P and Q are state predicates over the variables of F and G. Then, the following theorems hold:

[5] The notation "$(s_F \| s_G)$ *if fair*" means that the conjunction "$\wedge$ *fair*" is added to the guards of s_F and s_G.

$$\frac{\texttt{invariant}\ P\ \texttt{in}\ F [\!] G}{\texttt{invariant}\ P\ \texttt{in}\ F [\![\!]\!] G} \tag{1}$$

$$\frac{P \mapsto Q\ \texttt{in}\ F [\!] G\ \wedge\ Q \text{ local to } F}{P \mapsto Q\ \texttt{in}\ F [\![\!]\!] G} \tag{2}$$

$$\frac{P \mapsto Q\ \texttt{in}\ F [\!] G\ \wedge\ \texttt{stable}\ Q\ \texttt{in}\ F [\!] G}{P \mapsto Q\ \texttt{in}\ F [\![\!]\!] G} \tag{3}$$

These theorems extend to composition between N processes. Theorem (1) states that *all* the invariants are preserved when converting from composition by union into composition by product. Theorem (2) states that $P \mapsto Q$ is preserved if the predicate Q is local to one process (all the free variables of the predicate Q are variables of a single process). Finally, Theorem (3) states that $P \mapsto Q$ is also preserved when the predicate Q is stable.

The three theorems state that all the safety *invariant* properties are preserved and that the liveness *leads-to* properties not preserved must involve a *global* and *transient* state, and therefore are not significant in distributed application descriptions (since processes cannot detect them).

A more detailed description of the mapping and the proofs of the theorems can be found in [5].

4 Programming Examples

In this section, we illustrate the observation relation through examples. The first example shows how the observation can be used to express communication patterns, especially, we show how the "famous" alternating bit protocol can be stated. The second example illustrates the use of superposed observations. The third example shows how the observation mechanism can be used to refine a "centralized" communication into a distributed one.

4.1 Communication Patterns

In this section, we show how the alternating bit protocol can be derived with observations. First we propose a centralized solution: the computation is performed on data available at the sender node as well as at the receiver node. Based on this solution, we propose another solution where computations are performed on data available either at the sender node or at the receiver node. Interactions between nodes occur only through observations. In the following, we present each solution as a communication pattern and we state the relevant safety properties.

The $\circ$ Pattern. The aim of the $\circ$ program is to express that the emitting process cannot produce a new value unless it is sure that the current one has been received. A model of such a behavior is (along with the program, the safety properties are stated through the history and the *next* relations):

```
Program o (tick : T → T)
Declare
  em, rec : T;
Observe
  rec ≺ em;
Assign
  em := tick(rec)
End
```

$$
\begin{aligned}
&\mathtt{o}(\mathtt{tick},\mathtt{em},\widehat{\mathtt{em}},\mathtt{rec},\widehat{\mathtt{rec}}) = \\
&\qquad \mathtt{em}\,\mathcal{H}\,\widehat{\mathtt{em}} \wedge \mathtt{rec}\,\mathcal{H}\,\widehat{\mathtt{rec}} \wedge \\
&\qquad \mathtt{rec} \prec \mathtt{em} \wedge \\
&\qquad \forall\, \mathtt{E},\mathtt{R} :: (\mathtt{em} = \mathtt{E} \wedge \mathtt{rec} = \mathtt{R}) \;\; \mathtt{next} \;\; (\mathtt{em} = \mathtt{E} \vee \mathtt{em} = \mathtt{tick}(\mathtt{R}))
\end{aligned} \tag{4}
$$

Properties: The main safety property of the o pattern states that the sequence of the received values is a prefix of the sequence of the emitted values; we show the stronger property:

$$\mathtt{o}(\mathtt{tick},\mathtt{em},\widehat{\mathtt{em}},\mathtt{rec},\widehat{\mathtt{rec}}) \Rightarrow \mathtt{invariant}\; \widehat{\mathtt{em}} = \widehat{\mathtt{rec}} \;\vee\; \widehat{\mathtt{em}} = \widehat{\mathtt{rec}};\mathtt{tick}(\mathtt{rec}) \tag{5}$$

The oo Pattern.

Unlike the o program, where the assignment involves the two variables em and rec which are supposed to be on different nodes, in the oo program the assignment involves the variables em and ack which are supposed to be on the same node. Moreover, thanks to the observation properties, the oo program can be considered as a model for the alternating bit protocol.

```
Program oo (tick : T → T)
Declare
  em, ack, rec : T;
Observe
  rec ≺ em;
  ack ≺ rec;
Assign
  em := tick(ack)
End
```

Actually, when composed with a clock function, a sequence of values is transformed in such a way that the usual assumptions on a communication medium are modeled, i.e., loss, duplication and no reordering.

Properties:

– each received value has been previously sent. As for the o pattern we show[6]:

$$
\begin{aligned}
&\mathtt{oo}(\mathtt{tick},\mathtt{em},\widehat{\mathtt{em}},\mathtt{rec},\widehat{\mathtt{rec}},\mathtt{ack},\widehat{\mathtt{ack}}) \\
&\quad \Rightarrow \mathtt{invariant}\; \widehat{\mathtt{em}} = \widehat{\mathtt{rec}} \vee \widehat{\mathtt{em}} = \widehat{\mathtt{rec}};\mathtt{tick}(\mathtt{rec})
\end{aligned} \tag{6}
$$

[6] The expression of the oo(...) predicate is similar to that of the o(...) predicate.

– when the sender has been acknowledged, the last value received is the last value sent:

$$\begin{aligned} &\mathtt{oo}(\mathtt{tick}, \mathtt{em}, \widehat{\mathtt{em}}, \mathtt{rec}, \widehat{\mathtt{rec}}, \mathtt{ack}, \widehat{\mathtt{ack}}) \\ &\quad \Rightarrow \mathtt{invariant}\ \widehat{\mathtt{em}} = \widehat{\mathtt{ack}} \Rightarrow \widehat{\mathtt{em}} = \widehat{\mathtt{rec}} \end{aligned} \tag{7}$$

Note: An equivalent expression of the oo pattern is obtained by replacing the assignment $\mathtt{em} := \mathtt{tick}(\mathtt{ack})$ by $\mathtt{em} := \mathtt{tick}(\mathtt{em})\ \mathtt{if}\ \mathtt{em} = \mathtt{ack}$.

The wave Pattern. The wave paradigm is an important one for distributed computing. Actually, many termination algorithms use it to collect the states of a distributed computation. In this section, we express this paradigm as a communication pattern. Note that it can be considered as a generalization of the oo pattern.

Program $\mathtt{wave}\ (\mathtt{tick} : \mathtt{T} \rightarrow \mathtt{T})$
Declare
  $\mathtt{em} : \mathtt{T};$
  $\mathtt{ack},\ \mathtt{rec} :$ **array** $[0..N-1]$ **of** $\mathtt{T}$
Observe
  $\langle \wedge\ i :: \mathtt{rec}[i] \prec \mathtt{em} \rangle;$
  $\langle \wedge\ i :: \mathtt{ack}[i] \prec \mathtt{rec}[i] \rangle$
Assign
  $\mathtt{em} := \mathtt{tick}(\mathtt{em})$ **if** $\langle \wedge\ i :: \mathtt{em} = \mathtt{ack}[i] \rangle$
End

Property: We only state the property that allows us to inherit oo properties:

$$\begin{aligned} &\mathtt{wave}(\mathtt{tick}, \mathtt{em}, \widehat{\mathtt{em}}, \mathtt{rec}, \widehat{\mathtt{rec}}, \mathtt{ack}, \widehat{\mathtt{ack}}) \\ &\quad \Rightarrow \langle \wedge i :: \mathtt{oo}(\mathtt{tick}, \mathtt{em}, \widehat{\mathtt{em}}, \mathtt{rec}[i], \widehat{\mathtt{rec}}[i], \mathtt{ack}[i], \widehat{\mathtt{ack}}[i]) \rangle \end{aligned} \tag{8}$$

4.2 Superposed Communication

Reusability is an important programming method. In the UNITY framework, superposition can be considered as a mechanism to allow such a programming style. The following examples illustrate the superposition idea applied to observations. More precisely, we show how the communication patterns of the previous section can be reused through superposition in order to build new ones. Since the aim of this section is just to illustrate reusability of patterns, we only state the relevant properties.

The s_oo Pattern. First, it should be noted that the usual alternating bit protocol can be easily derived: actually the communication of the control bit and data between the sender and the receiver can be modeled as $(\mathtt{bit\_rec}, \mathtt{data\_rec}) \prec (\mathtt{bit\_em}, \mathtt{data\_em})$ which can be interpreted as the superposition of the data communication to the control communication. Of course, such a superposition inherits the properties of the underlying program. In this section, we study another superposition to the oo pattern: the s_oo pattern. We add the variables oc and c and the observation $\mathtt{oc} \prec\!\cdot\ \mathtt{c}$ is superposed to $\mathtt{ack} \prec \mathtt{rec}$. The program associated to the s_oo pattern is the following:

Program $\texttt{s\_oo}$ $(\texttt{tick} : \texttt{T} \rightarrow \texttt{T})$
Declare
  $\texttt{em}, \ \texttt{ack}, \ \texttt{rec} : \ \texttt{T}; \ \texttt{oc}, \ \texttt{c} : \ \texttt{C};$
Observe
  $\texttt{rec} \ \prec \ \texttt{em};$
  $(\texttt{ack}, \texttt{oc}) \ \prec \ (\texttt{rec}, \texttt{c})$
Assign
  $\texttt{em} \ := \ \texttt{tick}(\texttt{ack})$
End

Properties: The properties of the oo pattern are naturally inherited. Moreover, if the variable c is a non-decreasing counter, the s_oo pattern has the following property:

$$\begin{array}{l}\texttt{invariant}\\ (\exists P_1, P_2 :: (\texttt{em}, \widehat{\texttt{ack}, \texttt{oc}}, \texttt{c}) = (P_1; (\texttt{em}_1, \texttt{ack}_1, \texttt{oc}_1, \texttt{c}_1))@(P_2; (\texttt{em}_2, \texttt{ack}_2, \texttt{oc}_2, \texttt{c}_2)) \\ \qquad \wedge (\texttt{em}_1 = \texttt{ack}_1) \wedge (\texttt{em}_2 = \texttt{ack}_2) \wedge (\texttt{em}_1 \neq \texttt{em}_2)) \\ \quad \Rightarrow \texttt{c}_1 \leqslant \texttt{oc}_2\end{array}$$

This property is useful since the knowledge of two different observations ($\texttt{em}_1 \neq \texttt{em}_2$) such that their data values are the same ($\texttt{oc}_1 = \texttt{oc}_2$) implies the existence of a *global* state where $\texttt{c} = \texttt{oc}_1$ (since c is a counter, we have $\texttt{invariant oc} \leqslant \texttt{c}$, then from $\texttt{oc}_1 \leqslant \texttt{c}_1$, we conclude $\texttt{oc}_1 = \texttt{c}_1$).

The s_wave Pattern. In a similar way to the s_oo pattern, we add the variables oc and c and we superpose the observations $\langle \wedge i :: \texttt{oc}[i] \prec\cdot \texttt{c}[i] \rangle$ (respectively) to $\langle \wedge i :: \texttt{ack}[i] \prec \texttt{rec}[i] \rangle$. The program associated to the s_wave pattern is the following:

Program $\texttt{s\_wave}$ $(\texttt{tick} : \texttt{T} \rightarrow \texttt{T})$
Declare
  $\texttt{em} : \ \texttt{T};$
  $\texttt{ack}, \ \texttt{rec} : \ \ \textbf{array} \ [0..N-1] \ \ \textbf{of} \ \texttt{T}$
  $\texttt{oc}, \ \texttt{c} : \ \ \textbf{array} \ [0..N-1] \ \ \textbf{of} \ \texttt{C}$
Observe
  $\langle \ \wedge \ i \ :: \ \texttt{rec}[i] \ \prec \ \texttt{em} \ \rangle;$
  $\langle \ \wedge \ i \ :: \ (\texttt{ack}[i], \texttt{oc}[i]) \ \prec \ (\texttt{rec}[i], \texttt{c}[i]) \ \rangle$
Assign
  $\texttt{em} \ := \ \texttt{tick}(\texttt{em}) \ \textbf{if} \ \langle \wedge \ i \ :: \ \texttt{em} \ = \ \texttt{ack}[i] \rangle$
End

Properties: Properties of the s_wave pattern can be derived from those of the s_oo pattern, in the way that the properties of the wave pattern are derived from the oo pattern (8).

In [8], we show how the basic ideas of a variant of Mattern's termination algorithm [12] can be stated in terms of these patterns.

4.3 Using Observation to Refine Communication

The example we consider is a mutual exclusion algorithm similar to the algorithm described by E.W. Dijkstra in [10]. 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 and its predecessor is necessary. In this section, we give two descriptions of the system. In the first one, each node has an instantaneous vision of its predecessor and in the second one, the nodes use observations to communicate. In [8], we prove that the second system refines the first one.

We now describe this system with a UNITY program. This program P is a composition by union of N identical programs $\mathtt{Node}_i$:

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

Each node has the following behaviour:

Program $\mathtt{Node}_i$
Declare $\mathtt{Req}_i, \mathtt{Excl}_i : \mathtt{bool};\ \mathtt{c}_i : \mathtt{int};$
Initially $\mathtt{c}_i = i \wedge \neg\mathtt{Excl}_i \wedge \neg\mathtt{Req}_i$
Assign - - requesting the critical section access
$\mathtt{Req}_i := \mathtt{true}$
$[\!]$ - - Entering the critical section when requesting and being on the edge
$\mathtt{Excl}_i := \mathtt{true}\ \mathtt{if}\ \mathtt{Req}_i \wedge \neg\mathtt{Excl}_i \wedge \mathtt{c}_i < \mathtt{c}_{i-1}$
$[\!]$ - - leaving the critical section, passing the edge to the next
$\mathtt{Excl}_i, \mathtt{Req}_i, \mathtt{c}_i := \mathtt{false}, \mathtt{false}, \mathtt{c}_i + N\ \mathtt{if}\ \mathtt{Excl}_i$
$[\!]$ - - passing the edge to the next when no request is pending
$\mathtt{c}_i := \mathtt{c}_i + N\ \mathtt{if}\ \neg\mathtt{Req}_i \wedge \mathtt{c}_i < \mathtt{c}_{i-1}$
End

The second and fourth statements of the node i use the value $\mathtt{c}_{i-1}$ 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 $\mathtt{Node}'_i$
Declare $\mathtt{Req}_i, \mathtt{Excl}_i : \mathtt{bool};\ \mathtt{c}_i, \mathtt{t}_i : \mathtt{int};$
Observe $\boxed{\mathtt{t}_i \prec \mathtt{c}_{i-1}}$
Initially $\mathtt{c}_i = i \wedge \neg\mathtt{Excl}_i \wedge \neg\mathtt{Req}_i$
Assign $\mathtt{Req}_i := \mathtt{true}$
$[\!]$ $\mathtt{Excl}_i := \mathtt{true}\ \mathtt{if}\ \mathtt{Req}_i \wedge \neg\mathtt{Excl}_i \boxed{\wedge\ \mathtt{c}_i < \mathtt{t}_i}$
$[\!]$ $\mathtt{Excl}_i, \mathtt{Req}_i, \mathtt{c}_i := \mathtt{false}, \mathtt{false}, \mathtt{c}_i + N\ \mathtt{if}\ \mathtt{Excl}_i$
$[\!]$ $\mathtt{c}_i := \mathtt{c}_i + N\ \mathtt{if}\ \neg\mathtt{Req}_i \boxed{\wedge\ \mathtt{c}_i < \mathtt{t}_i}$
End

The only difference between $\mathtt{Node}_i$ and $\mathtt{Node}'_i$ is that $\mathtt{c}_i < \mathtt{t}_i$ replaces $\mathtt{c}_i < \mathtt{c}_{i-1}$ in the second and fourth statement guards.

The distributed system is the composition by union of the programs $\mathtt{Node}'_i$:

$$P' = \langle [\!] i : 0 \leqslant i < N :: \mathtt{Node}'_i \rangle$$

A result known as *guard 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 that $p \mapsto q$ and q $\mathtt{unless}$ $\neg p$ hold.

Traditionally, the theorem is stated with the two properties being proved on the program without the statement τ [4, 16]. However, it is proved in [6] that the theorem is valid for the operational logic of UNITY when both properties are proved on the complete program *after* the strengthening.

By induction, we can strengthen the second statement of each node, and then similarly the fourth statement with the term $\mathtt{c}_i < \mathtt{t}_i$. For each strengthening, the *unless* and *leads-to* conditions are easily proved using observation theorems about increasing counters. In the resulting program, the statements are guarded with the same condition applying both on the sources ($\mathtt{c}_i < \mathtt{c}_{i-1}$) and on the images ($\mathtt{c}_i < \mathtt{t}_i$). Again, the theorems about the observation of increasing counters allows to prove an invariant stating that the condition on the sources is weaker than the condition on the images: $\mathtt{invariant}\ \mathtt{c}_i < \mathtt{t}_i \Rightarrow \mathtt{c}_i < \mathtt{c}_{i-1}$. Therefore, we can just keep the condition on the images. The inductions and the applications of observation theorems are detailed in [8].

The refinement relation being proved, the mutual exclusion property and the fairness of the algorithm can be proved on program P without considering the communication part of the system.

Moreover, the system P' is well-distributed. The mutual exclusion property (*invariant*) and the fairness of the algorithm (*leads-to* with local right-hand side) satisfy the conditions stated in section 3.2. Therefore they also hold in the composition by product of the programs $\mathtt{Node}'_i$.

5 Conclusion

This paper presents two extensions to UNITY for distributed systems. A set of distributed algorithms are then described using these extensions. These examples illustrate how to refine a centralized UNITY description into a distributed one. UNITY's expressiveness is increased as interprocess communication is described at a high level and the mapping on nodes is easier. Due to space constraints, proofs have been omitted. Interested readers can refer to the full report [8].

Observations seem well suited for the expression of self-stabilizing algorithms. In [3], programs are transformed by adding so called *asynchronous delays* and the impact on stabilization properties is analyzed. These delayed variables can be specified as observations. We are studying how these transformations could be applied to programs with observations and stated as refinements as in section 4.3.

Besides, causality, which is a useful property of distributed systems, is still difficult to describe. Currently, the causal ordering of events is described by means of explicit counters (implementing (vector) clocks), which remains a low level description. A more abstract mechanism is needed to handle this crucial problem.

References

1. M. Accetta, R. Baron, D. Golub, R.F. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for UNIX development. In *Summer USENIX Conf.*, pages 93–112, 1986.
2. G.R. Andrews, R.A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.
3. A. Arora and M.G. Gouda. Delay-Insensitive stabilization. In S. Ghosh and T. Herman, editors, *Self-Stabilizing Systems*, volume 7 of *International informatics series*, pages 95–109. Carleton university press, 1997.
4. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
5. M. Charpentier. A UNITY mapping operator for distributed programs. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Fourth International Symposium of Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 665–684. Springer-Verlag, September 1997.
6. M. Charpentier. *Assistance à la Répartition de Systèmes Réactifs*. Thèse de doctorat, Institut National Polytechnique de Toulouse, France, November 1997.
7. M. Charpentier, M. Filali, P. Mauran, G. Padiou, and P. Quéinnec. Abstracting communication to reason about distributed algorithms. In Ö. Babaoğlu and K. Marzullo, editors, *Tenth International Workshop on Distributed Algorithms (WDAG'96)*, volume 1151 of *Lecture Notes in Computer Science*, pages 89–104. Springer-Verlag, October 1996.
8. M. Charpentier, M. Filali, P. Mauran, G. Padiou, and P. Quéinnec. Tailoring UNITY to distributed program design. Technical report 97-55-R, Institut de Recherche en Informatique de Toulouse, France, November 1997. 21 pages. Available as http://www.enseeiht.fr/Recherche/Info/Logiciel/mvr/publis/RR_97_55_R.ps.
9. M. Charpentier and G. Padiou. Specification and verification of the ATMR protocol using UNITY. In D. Méry, editor, *International workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'97)*, pages 26–36, University of Geneva, Switzerland, April 1997.
10. E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.
11. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.
12. F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.
13. J. Misra. *A Logic for Concurrent Programming*. Technical Report ("New UNITY"), Department of Computer Science, University of Texas at Austin, 1994.
14. M. Raynal. *Algorithmique du parallélisme : le problème de l'exclusion mutuelle*. Dunod, 1984.
15. M. Rozier and J. L. Martins. The Chorus distributed operating system: some design issues. In Y. Paker, J.-P. Banâtre, and M. Bozyigit, editors, *Distributed Operating Systems*, pages 261–287. Springer Verlag, January 1986.
16. A.K. Singh. Program refinement in fair transition systems. *Acta Informatica*, 30:503–535, May 1993.