A UNITY Mapping Operator for Distributed Programs*

Michel Charpentier

INPT-ENSEEIHT / IRIT
2, rue Charles Camichel
31071 Toulouse cedex, FRANCE
e-mail: charpov@enseeiht.fr

Abstract. When describing a distributed application within the UNITY framework, one usually uses the composition by union to express the interaction between processes. Unfortunately, the semantics of union is not well-suited to express the possible true parallelism between the different parts of a program. However, the different processes of a distributed application do not share any memory. Thanks to this particularity and the definition of an abstract communication model, the composition by union can be modified to fit the parallelism of distributed programs. This paper gives a set of theorems that characterize the UNITY properties of a union program that remain valid when the processes of the program are composed with the new operator. Therefore, this operator can be used to formalize the mapping of programs to distributed architectures.

1 Introduction

The work presented here is related to our current project called DADA for *Distributed Algorithms Design Assistant*. The DADA project aims to help the design and the proof of asynchronous distributed programs, that is to say programs sharing neither memory nor a common clock. This project involves a three way research:

- distribution formalization: we are working on a well-formalized definition of distribution to be incorporated into the UNITY framework. We have developed a new communication abstraction based on the *observation* relation [4, 6] described in Sect. 4. This observation relation extends the UNITY logic and appears to be helpful to describe and prove distributed algorithms [5];
- automatic proof of parallel programs: the kind of proofs we have to design are very difficult and tedious and need the help of automatic tools. For that purpose, we have developed a UNITY-based environment working on a subset of the language [7, 8]; the environment allows us to automatically check the consistence of a specification with its corresponding implementation.

* International Symposium of Formal Methods Europe (FME'97), September 1997, LNCS 1313, pages 665–684.

– program refinement: automatic tools are not always able to prove the properties of a large or complex program. Our idea, in this case, is to refine a centralized (or not fully distributed) program description towards another expression well-suited to distribution. We investigate such transformations, able to introduce distribution into a program without loosing its correctness [10].

In this context, the reminder of the paper deals with distributed programs which are described, together with their properties, within the UNITY formalism extended with observation. More precisely, we investigate the consequences of union semantics on the mapping of programs to distributed architectures. This leads to the definition of a new composition operation which appears well-suited to formalize the mapping step in the design of a distributed application.

First, we introduce the UNITY framework and some specific notations in Sect. 2. Section 3 gives an example showing the difficulties arising when mapping programs composed by union to distributed architectures. Then, we present our communication model and the logical relation it is based on in Sect. 4. A new composition operation, called product, is defined in Sect. 5. We investigate more precisely the relationship between union and product in Sect. 6. This is achieved by considering the transformation from union into product and the converse transformation. The theorems presented in this part are then proved in Sect. 7.

2 The UNITY Framework

2.1 Introduction

This section describes briefly the UNITY formalism. A more comprehensive presentation can be found in the given references, especially [3, 13].

The UNITY framework consists of two parts:

– a programming notation semantically based on fair transition systems;
– a specification language based on a linear temporal logic, able to express the safety properties and the progress properties of the former systems.

Different semantic models have been proposed [2, 3, 9, 15, 17] both for the language and for the logic. These models are related to each other but are not equivalent. We now make precise the choice used for this work.

2.2 The Language

A UNITY program comprises three main parts, namely the declaration of the state variables (*declare*), their initial values (*initially*) and the description of the system transitions (*assign*).

For a given program F, $F.declare$ contains the declaration of the state variables of the program along with their types. We are not interested in types in this paper and we consider, for the sake of simplicity, $F.declare$ simply as the *set* of state variables. In particular, we use set operations like $\in$, $\cup$ or $\cap$.

F.init is used to constrain the initial values of the state variables. In [3, 13], *F.init* assigns an initial value to some of the variables. Following [9, 15, 17], we prefer to see *F.init* as a *predicate* stating the acceptable initial states. This is useful to state initial conditions like $x \geqslant 0$ or $x < y$ without assigning explicit values to variables. We suppose that *F.init* is satisfiable.

The *F.assign* part is the set of the program statements. It describes all the valid transitions of the system with multiple guarded assignments, separated by the $[\!]$ mark. These assignments are of the following form:

$$\begin{aligned} \bar{v} := {} & \bar{e}_1 \ \ \texttt{if} \ \ g_1 \\ \sim {} & \bar{e}_2 \ \ \texttt{if} \ \ g_2 \\ & \ldots \\ \sim {} & \bar{e}_n \ \ \texttt{if} \ \ g_n \end{aligned}$$

where $\bar{v}$ is a list of references to be assigned, $\bar{e}_i$ is a list of values and $\sim$ acts as an *"else if"* operator. A UNITY statement must be deterministic: when several guards are simultaneously true, the right-hand side values must be the same. More precisely, the statement must satisfy $\langle \forall i, j :: g_i \wedge g_j \Rightarrow \bar{e}_i = \bar{e}_j \rangle$. Moreover, the *assign* section of a program is always supposed to contain, in addition to the statements explicitly described, the *skip* statement that has no effect on a state [13].

The $\|$ mark denotes the simultaneous assignment of several variables: $x := a \| y := b$ is the same as $x, y := a, b$. More generally, if $\bar{v}$ and $\bar{w}$ contain no common references, $\bar{v} := \bar{e}$ $\texttt{if}$ $p \parallel \bar{w} := \bar{f}$ $\texttt{if}$ q represents:

$$\begin{aligned} \bar{v}, \bar{w} := {} & \bar{e}, \bar{f} \ \ \texttt{if} \ \ p \wedge q \\ \sim {} & \bar{e}, \bar{w} \ \ \texttt{if} \ \ p \wedge \neg q \\ \sim {} & \bar{v}, \bar{f} \ \ \texttt{if} \ \ \neg p \wedge q \end{aligned}$$

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

Therefore, an operational semantic model for UNITY is a characterization of all the possible computations given the program text. For program F, we denote the set of possible computations by $\mathcal{O}.F$. A computation σ is a sequence of states σ_i. For an expression e, the notation $\sigma_i.e$ represents the value of the expression e in the state σ_i. Several models $\mathcal{O}$ were proposed [9, 17] and we define another in Sect. 4.3 in accordance with our communication model.

2.3 The Logic

The UNITY framework also provides a linear temporal logic to describe and prove some properties of programs. This logic is based on the operators *unless*, *stable*, *constant*, *invariant*, *leads-to* ($\mapsto$) and *until*[1]. Because they are fundamental properties of systems [16], we only consider *invariant* and *leads-to* relations.

[1] In a purely operational semantic model, where only computations are considered, the operator *ensures* is removed from the theory presented in [3].

However, we also give the operational definition of *stable* which is useful to state properties about *leads-to* in Sect. 6.

Given a program F, its operational semantics $\mathcal{O}.F$ and state predicates p and q, the *stable*, *invariant* and *leads-to* properties are defined as follows:

$$
\begin{array}{ll}
\texttt{invariant}\ p\ \ \texttt{in}\ \ F & \equiv \langle \forall \sigma : \sigma \in \mathcal{O}.F :: \langle \forall t :: \sigma_t.p \rangle\rangle \\
p\ \mapsto\ q\ \ \texttt{in}\ \ F & \equiv \langle \forall \sigma : \sigma \in \mathcal{O}.F :: \langle \forall t :: \sigma_t.p \Rightarrow \langle \exists t' : t' \geqslant t :: \sigma_{t'}.q \rangle\rangle\rangle \\
\texttt{stable}\ p\ \ \texttt{in}\ \ F & \equiv \langle \forall \sigma : \sigma \in \mathcal{O}.F :: \langle \forall t :: \sigma_t.p \Rightarrow \langle \forall t' : t' \geqslant t :: \sigma_{t'}.p \rangle\rangle\rangle
\end{array}
$$

2.4 Program Composition by Union

The Unity logic provides an operator, called *union* and denoted by $[\!]$, to compose different programs. The state variables of programs F and G are gathered to become the variables of the program $F[\!]G$. The statements of $F.assign$ and $G.assign$ are also collected into $(F[\!]G).assign$ and any initial state of $F[\!]G$ must be a valid initial state both for F and for G. Formally:

$$
\begin{array}{ll}
(F[\!]G).declare & = F.declare\ \cup\ G.declare \\
(F[\!]G).init & = F.init\ \wedge\ G.init \\
(F[\!]G).assign & = F.assign\ \cup\ G.assign
\end{array}
$$

This composition by union is used to:

– write programs in a modular way;
– represent the different parts of a distributed or concurrent application.

The distributed application built from the processes described by programs P_i is represented by the program $\langle [\!] i :: P_i \rangle$. By analogy, each part P_i of the union is called in this case a Unity *process.*

No memory is shared between the different processes of a distributed program. Therefore, different Unity processes must not share any state variable. In this case, the composite program is said to be *well distributed*:

$$\langle [\!] i :: P_i \rangle \text{ is } \textit{well distributed} \ \equiv\ \langle \forall i, j : i \neq j :: P_i.declare\ \cap\ P_j.declare = \emptyset \rangle$$

This definition of *well distribution* may look too restrictive. It eliminates all kinds of shared variables instead of classifying them carefully by specifying how they can be accessed. For instance, this classification can rely on type extension [12] or on predicate transformers [17]. We must outline that the problem of interaction between programs is not the matter of this paper. Instead, we are interested in mapping, and from the mapping point of view, distributed programs have no shared variables.

However, well distributed programs need a communication mechanism to exchange information. We use a communication mechanism based on observation which is described in Sect. 4.

2.5 Mapping Programs to Architectures

The UNITY formalism was intended to abstract program design from architecture. Therefore, the formalism was not designed for one specific architecture. Instead, UNITY programs can be implemented on many architectures, but a mapping step is necessary.

This mapping step is an informal part of a program description. It says how the program can be executed on a specific architecture. Different mappings for different architectures are described in [3]. In particular, for distributed architectures, the mapping must partition the set of statements and the set of variables among processors, so that any statement is on the same processor as the variables it uses. The mapping must also choose a control flow for each processor with respect to the weak fairness constraint. Lastly, the mapping must define a communication mechanism, for instance by assigning variables of type sequence to communication channels.

3 Presentation of the Problem

Describing distributed programs with the composition by union of UNITY involves considering programs of the form:

$$P \;=\; \langle [\!]\, i :: P_i \rangle$$

Remember that the composition by union only gather the statements from the processes P_i into the *assign* part of the program P. Therefore, due to the semantics of UNITY, these statements are mutually exclusive, even when they come from distinct processes P_i.

When mapping such a program on a distributed architecture, two statements from two different UNITY processes go on two different real processors. Therefore, no mutual exclusion is guaranteed between these two statements: different sites are always likely to execute one statement each, exactly at the same time. Moreover, there is no way to prevent processes not sharing any clock to do so.

By transgressing this mutual exclusion constraint, a distributed program may behave in a slightly different way from its UNITY model as shown in the following example. Consider two processes and their UNITY descriptions F and G:

Program F		Program G	
Declare	$x : int$	Declare	$y : int$
Initially	$x = 0$	Initially	$y = 0$
Assign	$x := 1$	Assign	$y := 1$

The concurrent computation of these two processes is then described by the composition $F [\!] G$, which is only a shorthand for:

Program $F [\!] G$	
Declare	$x, y : int$
Initially	$x, y = 0, 0$
Assign	$x := 1$
$[\!]$	$y := 1$

If processes F and G are mapped on a distributed architecture, the computation

$$(x, y) : (0,0) \to (1,1) \to (1,1) \to \dots \tag{1}$$

should be valid, since it corresponds to the case of the two processes executing their statement at the same time. But this behaviour is not allowed for $F [\!] G$ because the two statements are mutually exclusive in this program.

This differences between the operational semantics of the distributed application and that of the UNITY program is a serious problem because one may prove within the UNITY framework properties that will be falsified on a distributed architecture. For example, the property $x + y = 0 \;\mapsto\; x + y = 1$ can be proved for the program $F [\!] G$, but is falsified by the computation (1) of the processes F and G in a distributed context.

Therefore, it appears that the traditional mapping of programs composed by union is not well-suited to distributed architectures. Such a mapping does not take into account all the concurrency between the different processes of a distributed application. In this paper, we define precisely another kind of mapping, able to represent simultaneous computation of statements from different processes. This mapping relies on the separation of variables (idea of well-distributed programs) and the choice of an abstract communication model (observation).

4 Observation-based Communication

4.1 Presentation

In the UNITY model, communication between processes relies on shared variables. Generally, variables of type sequence are used to describe distributed applications. These sequences correspond to a low-level model of communication based on message exchanges. However, this communication model makes difficult both the description and the proof of algorithms: 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 gave in Sect. 2.4 the definition of a well-distributed program. In such a program, processes share no variables. Therefore, we were led to define another communication abstraction more consonant with UNITY principles. This abstraction, called *observation*, weakens the shared variable model to adapt it to distribution. It is based on a kind of weak consistency between variables. It can be seen as a relation (extending the logic) or as a mechanism (extending the model) relying on the relation.

4.2 Observation Relation

The observation relation describes the relationship between a *source* variable and an *image* variable in a program. It takes the form of a temporal operator based on any operational semantic model of UNITY. This operator has its own properties (in particular, it is an order[2]) and properties related to other UNITY logical operators.

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

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

$Clock(C)$ means that C is an abstract clock, that is 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 monotonously increasing;
3. $\langle \forall k :: \langle \exists t :: C.t > k \rangle\rangle$: an abstract clock eventually increases.

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

A more comprehensive presentation of the observation relation and its properties can be found in [6, 5].

4.3 Observation Mechanism

The observation *mechanism* is part of a program description. It is made of assumed observation *relations* between some variables of the program. In the next section, this mechanism will be used to represent communication between processes.

A new section, named *observe*, is introduced into UNITY programs. It consists of a set of observation relations, the truthfulness of which the program environment is responsible for. There are constraints on these observation relations so that the mechanism maps straightforwardly on distributed architectures: image variables are supposed to be assigned by the observation mechanism alone. In particular, they cannot appear on the assignments left-hand side. Moreover, no variable can be image in more than *one* observation relation, even though a variable can be both image and source in two different relations. For example, $x \prec y,\ y \prec z$ is a correct *observe* section, but $x \prec y,\ x \prec z$ is not.

When satisfying these hypotheses, a UNITY program with an *observe* section can be translated into a standard program without observations [6]. However, this effective fulfillment of the observation relations must not be given explicitly if we want the mechanism to remain abstract. Therefore, we have to consider the *observe* section of a program at the semantic level.

This can be achieved by considering the environment part in the operational semantics of a program. Such semantic models are used in [9, 17]. They are based

[2] The relation $\prec$ is not strictly antisymmetric. We can only deduce an invariant of the program, that is to say: $x \prec y\ \mathtt{in}\ F \wedge y \prec x\ \mathtt{in}\ F \Rightarrow \mathtt{invariant}\ x = y\ \mathtt{in}\ F$.

on an interleaving between the program transitions and the environment transitions, the latter possibly being restricted with logical specifications. However, the observation mechanism cannot rely on such semantic models. Due to the observation relation definition, an image variable and its source must be able to change their value *at the same time*, that is to say on the same state transition. That means a program transition and an environment transition take place simultaneously. This is impossible in these interleaved semantic models[3].

Moreover, in these models, the environment is constrained at the *transition* level, not at the *computation* level, and only safety is considered. This is not enough to describe observation relations.

Therefore, a new operational semantic model is defined that allows image variables to change their value while a program assignment is executed and in accordance with the observation relations. The computation of the next state in a sequence of states must therefore take the observation relations into account.

This new semantic model, called $\mathcal{O}bs$, is defined as follows:

- $F.Img$ is the set of the image variables of F, that is to say the variables in the left-hand side of (exactly) one observation relation:

$$F.Img \;=\; \{x \in F.declare \mid \langle \exists y : y \in F.declare :: x \prec y \in F.observe \rangle\}$$

- $F.NImg$ is the set of the variables of F *not* image in any *observe* relation, that is to say the variables assigned by the program statements. The variables of the program F are then 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:
  - $\sigma_0.F.init$,
  - $\langle \forall t :: \langle \exists s : s \in F.assign :: \sigma_t \xrightarrow{s} \sigma_{t+1} \rangle\rangle$ and
  - $\langle \forall s : s \in F.assign :: \langle \forall t :: \langle \exists t' : t' > t :: \sigma_{t'} \xrightarrow{s} \sigma_{t'+1} \rangle\rangle\rangle$.

  where $\sigma_t \xrightarrow{s} \sigma_{t+1} \;\equiv\; \langle \forall x : x \in F.NImg :: \sigma_{t+1}.x = (s.\sigma_t).x \rangle$[4];
- $\mathcal{O}_{observe}.F$ 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 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 with any value. On the other hand, $\mathcal{O}_{observe}$ describes the states of the images determined by the observation relations of the *observe*

[3] The $[\![$ operator would not even be able to later introduce this simultaneity because the state changes due to the observation mechanism would not be described with UNITY assignments (cf Sect. 5).

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

section. The operational model $\mathcal{O}bs$ is made of the computations satisfying both constraints.

Note that the definition of the observation relation as a logical operator, given in Sect. 4.2, is based on *any* operational semantic model of UNITY. In particular, the operator and its properties can be used on a UNITY program with an *observe* section within the semantic model $\mathcal{O}bs$.

4.4 Communication Representation

Communication between processes can be described with the observation mechanism presented above in a more abstract way than with explicit messages sequences.

This is achieved by stating, in the *observe* section of a UNITY process, some observation relations with their sources located in other processes. The processes remain well-distributed if they were, but each process now has an approximated view of the other processes. We choose the observation relations between two UNITY processes (one that observes and one that is observed) to be located in the *observe* section of the process that observes. In other words, the images are local to processes and the sources are remote-located[5].

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 composite program:

$$(F[\!]G).observe \;=\; F.observe \cup G.observe$$

A given variable is supposed not to be the image of different observation relations from different *observe* sections so that the new *observe* section remains consistent. Note that any well-distributed program automatically satisfies this assumption.

The variables of a well-distributed program using an observation-based communication model can be seen by all processes but can only be assigned by *one* process (including its observation mechanism). Such variables are sometimes called *distributed variables* [14], or *output variables* [17], or *private variables* [17], or *read variables* [12], or *owned variables* [11].

Finally, due to the definition of the observation relation, the internal computations of processes and the communication are strictly concurrent: the images are altered by the observation mechanism and may change their value at the same time (that is to say during the same transition) as any ordinary assigned variable.

[5] That means we have to state, in a program F, relations like $x \prec\!\cdot\, y$ with possibly $y \notin F.declare$. This is only a matter of syntax. In fact, we consider the relation $x \prec\!\cdot\, y$ to be part of the description of the program $F[\!]G$ and the relation is not handled outside this program.

4.5 Example

Let us consider the following two UNITY processes:

Program F		Program G	
Declare	$x : int$	Declare	$y, z : int$
Initially	$x = 0$	Initially	$z = 0$
		Observe	$y \prec x$
Assign	$x := x + 1$ if $x < 10$	Assign	$z := y$

The variable x from the program F is increased from 0 to 10 by increments of one. The program G communicates with F through observation and has an image y of x. With respect to the semantic model $\mathcal{O}bs$, the variable z is monotonously increasing and bounded with 10 in the program $F \| G$. Properties such as $true \mapsto z = 10 \texttt{ in } F \| G$ and $z \prec x \texttt{ in } F \| G$ can also be proved.

A more comprehensive example can be found in [5] where a mutual exclusion algorithm based on logical clocks is fully described with observation. The proofs of its fundamental safety and liveness properties are also given.

5 A New Program Composition Operation

The communication between UNITY processes composed by union can be represented with the observation mechanism. As we saw previously, the communication is then strictly concurrent with the internal computations and need no additional transitions. Therefore, this communication model is compatible with the concurrency due to the mapping to a distributed architecture. However, the problem introduced in Sect. 3 still remains: the mapping must still forbid assignments to be executed at the same time by different processes.

In this section, a new composition operation is defined. It stems from the composition by union but handles simultaneous execution of statements by different processes. Therefore, this composition operator maps correctly on distributed architectures.

5.1 Principles

Due to the atomicity of UNITY statements, perfect simultaneity is the only case of overlapping between statements[6]. Indeed, two atomic statements take place at two different moments or at the same time but there is no other possibility. The composition by union does not handle the later case.

The idea of the new composition is to represent this possible simultaneity with additional synchronous statements. Each additional statement takes place for two or more assignments executing at the same time.

[6] If a partial overlapping need to be considered, that means the granularity chosen for the UNITY model is not the right one.

For example, the *assign* section of the program of Sect. 3 could be:

$$\begin{array}{ll} \text{Assign} & x := 1 \\ [\!] & y := 1 \\ [\!] & x, y := 1, 1 \end{array}$$

The computations of this program into the UNITY model and the computations of the two processes $F : x := 1$ and $G : y := 1$ concurrently are the same, even when F and G execute one statement each at the same time. Therefore, any distributed mapping of the two processes will satisfy all the properties proved on the program above.

5.2 Definition of the Product Operator

The ideas introduced above are now formalized. A new operator for program composition, called *product* and denoted by $[\![\!]\!]$, is defined.

In addition to the *skip* statement introduced in Sect. 2.2, we now assume that every UNITY program has the statement $fair := \neg fair$. The boolean variable *fair* is never declared and is not considered to be a variable of any process. It is only used for product composition. It cannot appear elsewhere in the program or in its properties. If such a variable already exists, it must be renamed. When reasoning on program computations, we always consider that the variable *fair* is part of the program state: a state of a program F is a mapping from $F.declare \cup \{fair\}$.

We can now formally define the product composition. 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:

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

Because $F[\!]G$ is well-distributed, the programs F and G share no variable outside the *observe* sections. Therefore, the synchronous statements $s_F \| s_G$ are correct UNITY statements (that is to say non conflicting).

The notation "$(s_F \| s_G)$ *if fair*" only means that the conjunction "$\wedge$ *fair*" is added to the guards of s_F and s_G. The variable *fair* is used to break the weak fairness of UNITY. Statements of the form $s_F \| s_G$ do not represent statements of the processes, but only the simultaneous execution of such statements. Therefore, they do not require any fairness. Maintaining fairness for such statements would correspond to consider that simultaneous execution *must* occur infinitely often, which is not consonant with the actual behaviour of an asynchronous distributed program. However, in UNITY, the weak fairness rule applies to *all* the statements of the *assign* section. This is the reason why we need to remove weak fairness for the statements introduced by product composition: due to the variable *fair*, no additional statement execution is forced.

Simultaneous execution of statements from different processes is now handled with the product operation by implicitly adding the corresponding assignments. The computations of the processes F and G in a distributed context and those of $F|||G$ in the UNITY model are the same.

As we will see, the additional statements need not be explicitly manipulated to reason about the program $F|||G$. As these additional statements are usually numerous, the theorems of Sect. 6 are very useful in hiding them from the user.

5.3 Simple Properties

In this section, simple properties of composition by product are stated. The reasoning in the next sections is made easier using these properties, in particular when generalizing theorems for any number of processes. The equalities are pure syntax: two programs are equal when they have the same set of statements.

$$
\begin{aligned}
F|||G &= G|||F \\
F|||(G|||H) &= (F|||G)|||H = F|||G|||H \\
F|||(G[]H) &= (F|||G)[](F|||H)
\end{aligned}
$$

The first two properties state that the composition by product is commutative and associative as was the composition by union. They rely on the commutativity and the associativity of $||$ when defined. The last property states that $|||$ distributes over $[]$.

6 Converting Union into Product

6.1 Introduction

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. But on the other hand, all the additional statements of the form $s_F||s_G$ must a priori be considered to prove properties on the program $F|||G$. These statements become numerous when the number of UNITY processes increases. All the possibilities of parallelism between two statements, between three statements, and so on, must be considered. A program with ten processes of ten statements each leads to ten billions additional statements.

Moreover, using the product composition when designing a UNITY program freezes the future mapping of the program: it precisely expresses which abstract UNITY process goes on which node. Since UNITY principles are to postpone such a mapping step, product operations should not appear too soon in the design of a program.

For both these reasons, 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 can be 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.

Referring to the example program of Sect. 3, we already know that some properties of $F \| G$ are not preserved in $F \|\| G$, but if the class of preserved properties is large enough, one may keep doing proofs on $\langle \| i :: P_i \rangle$ and then convert some unions into products depending on the desired mapping without loosing correctness.

6.2 Theorems

The theorems stated in this section show the preservation of some properties when converting from a composition by union into a composition by product. Only *invariant* and *leads-to* properties are considered, but similar results hold for *stable*, *constant* or *unless* and can be proved from the proof principles given in the next section.

In the following, F and G are 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:

$$\frac{\texttt{invariant } P \texttt{ in } F \| G}{\texttt{invariant } P \texttt{ in } F \|\| G} \tag{2}$$

$$\frac{P \mapsto Q \texttt{ in } F \| G \ \wedge \ Q \text{ local to } F}{P \mapsto Q \texttt{ in } F \|\| G} \tag{3}$$

$$\frac{P \mapsto Q \texttt{ in } F \| G \ \wedge \ \texttt{stable } Q \texttt{ in } F \| G}{P \mapsto Q \texttt{ in } F \|\| G} \tag{4}$$

Theorem (2) states that *all* the invariants are preserved when converting from composition by union into composition by product. Theorem (3) states that $P \mapsto Q$ is preserved as soon as the predicate Q is local to one process (that is to say all the free variables of the predicate Q are variables of a single process). Finally, Theorem (4) states that $P \mapsto Q$ is also preserved when the predicate Q is stable.

These theorems extend in case of a composition between N processes. For example, (3) becomes:

$$\frac{P \mapsto Q \texttt{ in } \langle \| i : 1 \leqslant i \leqslant N :: F_i \rangle \ \wedge \ \langle \exists i : 1 \leqslant i \leqslant N :: Q \text{ local to } F_i \rangle}{P \mapsto Q \texttt{ in } \langle \|\| i : 1 \leqslant i \leqslant N :: F_i \rangle}$$

The three theorems state that, when converting from union into product, 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).

Note that these theorems are theorems of the conversion from union into product and they are not composition theorems since properties of $F \|\| G$ are deduced from properties of $F \| G$ and not from properties of F and G separately.

One may wonder if new properties can be proved in a composition by product that were falsified in the corresponding composition by union. The answer is that the converse transformation (from product into union) is a program *refinement*. The following properties hold:

$$\frac{\texttt{invariant}\ P\ \texttt{in}\ F [\![\!]\!] G}{\texttt{invariant}\ P\ \texttt{in}\ F [\!] G} \tag{5}$$

$$\frac{P \mapsto Q\ \texttt{in}\ F [\![\!]\!] G}{P \mapsto Q\ \texttt{in}\ F [\!] G} \tag{6}$$

From (2) and (5), we deduce that the programs $F [\!] G$ and $F [\![\!]\!] G$ have *exactly* the same set of invariants, that is to say the same set of reachable states.

6.3 Example of Composition by Product

Let us consider a mutual exclusion algorithm for N nodes. The nodes are set along a logical ring, each having a predecessor and a successor. They handle counters so that, at any time, only *one* node has a counter smaller in value than the counter of its predecessor. This node may then enter the critical section. It releases access right to its successor by increasing its counter.

This algorithm requires communication between nodes so that a node knows about its predecessor. More precisely, each node *observes* the counter of its predecessor. The following Unity program describes the behaviour of a node:

Program $Node_i$
Declare $c_i, t_i : int$
$\quad req_i, excl_i : bool$
Initially $c_i = i \wedge \neg req_i \wedge \neg excl_i$
Observe $t_i \prec c_{i\ominus 1}$
Assign $req_i := true$
$[\!]$ $excl_i := true$ if $req_i \wedge c_i \leqslant t_i$
$[\!]$ $excl_i, req_i, c_i := false, false, t_i + 1$ if $excl_i \vee \neg req_i$

The variables req_i and $excl_i$ represent respectively a request for the critical section and the actual access to this section. The variable c_i is the counter of the node i and t_i is the image the node i has of the counter of its predecessor, the node $i \ominus 1$. The observation mechanism is responsible for updating the variable t_i.

The application made of N such nodes ($N > 1$) is described by the Unity program:

$$\langle [\!] i : 1 \leqslant i \leqslant N :: Node_i \rangle$$

This program satisfies usual properties of mutual exclusion algorithms:

invariant $\langle \forall i, j : 1 \leqslant i, j \leqslant N :: c_i \leqslant t_i \wedge c_j \leqslant t_j \ \Rightarrow\ i = j \rangle$	: frontier uniqueness
invariant $\langle \forall i, j : 1 \leqslant i, j \leqslant N :: excl_i \wedge excl_j \Rightarrow i = j \rangle$	: mutual exclusion
$\forall i : 1 \leqslant i \leqslant N :: c_i \leqslant t_i \mapsto c_{i\oplus 1} \leqslant t_{i\oplus 1}$	: frontier movement
$\forall i : 1 \leqslant i \leqslant N :: req_i \mapsto excl_i$	: algorithm fairness

The hypotheses of the theorems of Sect. 6.2 are satisfied by the program and its properties:

– each node only uses local variables outside its *observe* section and never assigns its image variable;
– the first two properties are *invariant* properties and are therefore automatically preserved;
– the last two properties are *leads-to* properties with a right-hand side local to a node ($i \oplus 1$ in the first case and i in the second case). Therefore, they are also preserved.

Since all the four properties are satisfied by the program composed by union, and since they are preserved, they still hold for the distributed program:

$$\langle [\![\!] i : 1 \leqslant i \leqslant N :: Node_i \rangle$$

7 Proofs

7.1 Important Lemma

The proof of the theorems of the preceding section relies on the following important lemma that shows a close relation between each computation of $\mathcal{O}bs.(F[\![\!]G)$ and a corresponding computation in $\mathcal{O}bs.(F[\!]G)$.

Given two UNITY processes F and G such that $F[\!]G$ is well-distributed and $s_F \| s_G$ `if` *fair* a statement of $(F[\![\!]G).assign$ with $s_F \in F.assign$ and $s_G \in G.assign$. Then, with respect to the notations introduced in Sect. 4.3:

$$\begin{aligned}&\langle \forall \sigma : \sigma \in \mathcal{O}bs.(F[\![\!]G) :: \langle \forall i : \sigma_i \xrightarrow{s_F \| s_G} \sigma_{i+1} :: \\ &\quad \langle \exists \sigma' : \sigma' \in \mathcal{O}bs.(F[\!]G) :: \langle \exists j :: \sigma'_j = \sigma_i \xrightarrow{s_F} \sigma'_{j+1} \xrightarrow{s_G} \sigma'_{j+2} = \sigma_{i+1} \rangle\rangle\rangle\rangle\end{aligned} \tag{7}$$

This lemma can be read as follows: in any computation of $F[\![\!]G$, the state transformation produced by executing an additional statement when *fair* is *true* is the same as the state transformation produced by executing, one after the other, the two parts of the statement in a particular computation of $F[\!]G$ (Fig. 1). Of course, when *fair* is *false*, the additional statement has no effect.

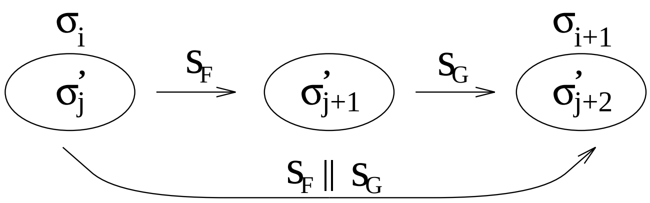

Fig. 1. Additional statements interpretation

To prove (7), one must show the existence of the sub-sequence $\sigma'_j \xrightarrow{s_F} \sigma'_{j+1} \xrightarrow{s_G} \sigma'_{j+2}$ in a computation of $\mathcal{O}bs.(F[\!]G)$. This is done in three steps:

- The separation of variables due to the well-distribution of F and G ensures:

$$\langle \forall \sigma_i :: (s_F \| s_G).\sigma_i = s_F.s_G.\sigma_i = s_G.s_F.\sigma_i \rangle$$

  The executions of s_F and s_G on a state σ_i are fully independent. The resulting state is the same, whether the statements are executed in any order or at the same time. This can be shown for instance by calculating the *weakest precondition* of $s_F \| s_G$, $s_F; s_G$ and $s_G; s_F$.
- Then, we assume $\langle \exists \sigma' : \sigma' \in \mathcal{O}bs.(F[\!]G) :: \langle \exists j :: \sigma'_j = \sigma_i \rangle \rangle$, that is to say σ_i is reachable in at least one computation σ' of $\mathcal{O}bs.(F[\!]G)$, and we show that the states with indexes greater than j can be chosen[7] for σ' so that $\sigma'_j \xrightarrow{s_F} \sigma'_{j+1} \xrightarrow{s_G} \sigma'_{j+2} = \sigma_{i+1}$. This is achieved by showing that the image variables can change their values concurrently with s_F and s_G in σ' in the same way they change concurrently with $s_F \| s_G$ in σ. If $`\bar{v}$ represents *all* the image variables in $F[\!]G$, then $\sigma'_j.`\bar{v} = \sigma_i.`\bar{v}$ (since $\sigma'_j = \sigma_i$). The computation $\sigma'$ is then chosen so that $\sigma'_{j+1}.`\bar{v} = \sigma'_j.`\bar{v}$ (the image variables do not change while executing $s_F$) and $\sigma'_{j+2}.`\bar{v} = \sigma_{i+1}.`\bar{v}$ (the image variables change while executing s_G in the same way they change while executing $s_F \| s_G$ in σ). These changes of the image variables are in accordance with the observation definition. Therefore, the desired computation σ' is built.
- In fact, the argument above proves the lemma for the *first* occurrence of an additional statement in σ. We conclude that the state σ_{i+1} is then reachable in a computation σ' and the same argument can be used again. Therefore, the proof of the lemma is extended by induction for all the occurrences of additional statements.

From the construction above, we see that σ' can be chosen in $\mathcal{O}bs.(F[\!]G)$ with the image variables not changing while executing s_F. This proves the stronger lemma:

$$\begin{array}{l}\langle \forall \sigma : \sigma \in \mathcal{O}bs.(F[\![\!]\!]G) :: \langle \forall i : \sigma_i \xrightarrow{s_F \| s_G} \sigma_{i+1} :: \\ \langle \exists \sigma' : \sigma' \in \mathcal{O}bs.(F[\!]G) :: \langle \exists j :: \sigma'_j = \sigma_i \wedge s_F.\sigma'_j = \sigma'_{j+1} \xrightarrow{s_G} \sigma'_{j+2} = \sigma_{i+1} \rangle\rangle\rangle\rangle \end{array} \quad (8)$$

which is used later in the proof.

7.2 Proof of the Theorems from Union into Product

Once the lemma is established, the theorems stated in Sect. 6.2 are easily proved. This lemma states that the additional statements have no effect but the bypass of exactly one state. Therefore, no new state becomes reachable and all *invariant* properties are preserved (Theorem (2)).

[7] Starting in a given state, the execution of a given statement can lead to different states depending on how images change.

Concerning *leads-to* properties, the hypothesis $P \mapsto Q$ `in` $F \| G$ states that, in any computation σ' in $\mathcal{O}bs.(F\|G)$, a P-state will always be followed by a Q-state. The only way not to respect this constraint for a computation σ in $\mathcal{O}bs.(F|||G)$ is to bypass the last Q-state (if their number is finite) or all Q-states from a point (if their number is infinite). In both cases, executing $s_F \| s_G$ must bypass a Q-state between two $\neg Q$-states to falsify the property (Fig. 2). Moreover, we saw that σ'

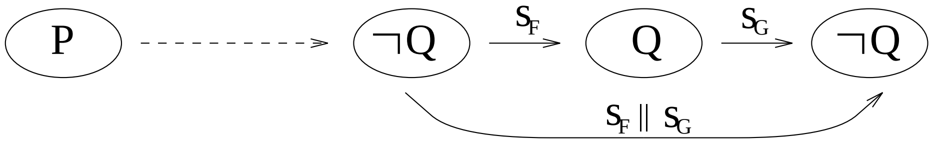

Fig. 2. Case of a *leads-to* property not being preserved.

can be chosen so that images do not change while executing s_F (Lemma (8)). If the predicate Q is local to process G (Theorem (3)), the transition $\neg Q \xrightarrow{s_F} Q$ is impossible (the case of Q local to F is proved with a symmetrical argument), and if the predicate Q is *stable* in the program $F \| G$ (Theorem (4)), the transition $Q \xrightarrow{s_G} \neg Q$ is impossible. That proves both theorems about *leads-to*.

7.3 Proof of the Theorems from Product into Union

The refinement property of the converse transformation (from product into union) is easy to prove since we took care of breaking the weak fairness of Unity for the additional statements. This allows to assert that, for two programs F and G:

$$\langle \forall \sigma' : \sigma' \in \mathcal{O}bs.(F\|G) :: \langle \exists \sigma : \sigma \in \mathcal{O}bs.(F|||G) :: \natural\sigma = \natural\sigma' \rangle\rangle \tag{9}$$

The operator $\natural$ is used to abstract from stuttering [1]. More precisely, $\natural\sigma$ is the same as σ except that all the maximal finite segments of identical states in σ are replaced by the single state in $\natural\sigma$. For a given computation σ', the corresponding computation σ is obtained by always executing the additional statements of the form "$s_F \| s_G$ *if fair*" when the variable *fair* is *false*.

We cannot assert that $\mathcal{O}bs.(F\|G) \subset \mathcal{O}bs.(F|||G)$ since a computation of $\mathcal{O}bs.(F\|G)$ only executing a finite number of times the *skip* statement when the variable *fair* is *false* is not in $\mathcal{O}bs.(F|||G)$: due to weak fairness, the computations of $\mathcal{O}bs.(F|||G)$ must have an infinite number of *skip* transitions starting from a $\neg$*fair*-state, corresponding to the execution of the additional statements.

Having proved (9), and since the computations σ and $\natural\sigma$ satisfy the same set of Unity properties, it is obvious that the Unity properties of $F|||G$ are valid properties of $F\|G$[8].

[8] This is a strong form of refinement. Weaker forms are possible. For example, a transformation can preserve all the Unity properties of a program without preserving

If the additional statements were obeying the weak fairness rule, (9) would not be true. Consider, for instance, the following programs F and G:

Program F			Program G	
Declare	$x : int$		Declare	$y : int$
Assign	$x := 1$		Assign	$y := 1$
$[\!]$	$x := 0$		$[\!]$	$y := 0$

Without removing the weak fairness, their product would contain the statement $x, y := 1, 1$. Therefore, the property $true \mapsto x+y = 2$ would be true for $F[\![\!]]G$ and not for $F[\!]G$. This is another justification for having introduced the boolean variable *fair*, even if it could be shown that only transient global *leads-to* properties are involved.

8 Conclusions

Because it forces mutual exclusion between statements, the composition by union cannot express the possible simultaneity when executing statements in different processes. Therefore, programs composed by union do not map in a straightforward way to distributed architectures.

However, synchronous statements can be described within the UNITY language, and this kind of statements can represent simultaneity. Therefore, these synchronous statements are added to the composition by union to define a new form of composition called *product*.

Moreover, we have shown that composing processes by product preserves most of the properties proved in a composition by union: all invariants are preserved and only some liveness properties about global and transient states are falsified. Thus, as shown with the example of Sect. 6.3, one may keep making proofs on the simpler union program without explicitly considering the additional statements.

Within our context of a design by stepwise refinement, two transformations may be distinguished when converting a centralized program into a distributed program:

- the localization of abstract processes among nodes;
- the increase of parallelism inherent to distribution.

The first point involves a mapping step which is not formally described in the UNITY framework, and the composition by union does handle correctly the second point. The criterion of *well-distribution* and the composition by product answer both these problems by formalizing the mapping and by enabling some true parallelism between statements of different nodes.

a single computation. This is due to the fact that two programs may have exactly the same set of UNITY properties and different computations, even when abstracting from stuttering [18].

In this work, we only consider UNITY programs using an observation-based communication model. It would be interesting to extend it to a more standard communication model like message-passing. However, this less abstract model would probably make the proofs more difficult.

The UNITY language was very appreciated for its ability to express synchronous statements. However, the formalism forces weak fairness for all statements. Thus, the definition of the product composition is more awkward than if the kind of fairness (including the complete absence of fairness), could be chosen for each statement, as it is the case for example in [11, 16].

Acknowledgements

Special thanks to G. Padiou and P. Quéinnec for their valuable comments on earlier drafts and their help in preparing the final version of this paper. The author is also grateful to the anonymous referees for careful reading of the manuscript and helpful comments.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. In *Third Annual IEEE Symposium on Logic in Computer Science*, pages 165–175. Computer Society Press, July 1988.
2. N. Brown and A. Mokkedem. On mechanizing proofs within a complete proof system for UNITY. In *Algebraic Methodology and Software Technology*, Concordia University of Montréal, 1995.
3. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
4. M. Charpentier, M. Filali, P. Mauran, G. Padiou, and P. Quéinnec. Observer pour répartir. In *Journées du GDR programmation*, Grenoble, France, November 1995.
5. 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.
6. M. Charpentier, M. Filali, P. Mauran, G. Padiou, and P. Quéinnec. Répartition par observation dans UNITY. Technical report 96-01-R, Institut de Recherche en Informatique de Toulouse, France, January 1996. 27 pages.
7. M. Charpentier, A. El Hadri, and G. Padiou. A UNITY-based algorithm design assistant. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, pages 131–145, Aarhus, Denmark, May 1995. BRICS Notes Series NS-95-2.
8. M. Charpentier, A. El Hadri, and G. Padiou. Preuve automatique dans un environnement de développement UNITY. *Technique et Science Informatiques (TSI)*, 15(1), January 1996.
9. P. Collette. *Design of Compositional Proof Systems Based on Assumption-Commitment Specifications. Application to* UNITY. Doctoral thesis, Faculté des Sciences Appliquées, Université Catholique de Louvain, June 1994.

10. M. Filali, P. Mauran, and G. Padiou. Raffiner pour répartir. In *Actes des quatrièmes Rencontres francophones du Parallélisme (RenPar'4)*, Villeneuve D'Ascq, France, 1992.
11. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.
12. J. Misra. Closures properties. In "New UNITY" [13], chapter 6.
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. B.A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, April–June 1991.
16. A.U. Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, September 1993.
17. R.T. Udink. *Program Refinement in* UNITY*-like Environments.* PhD thesis, Utrecht University, September 1995.
18. R.T. Udink and J.N. Kok. On the relation between UNITY properties and sequences 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. Springer-Verlag, 1993.