Communication relations: a paradigm for parallel program design

Guy-René Perrin
Laboratoire d'Informatique, Université de Franche-Comté, 25030 Besançon Cédex, France

Jean-Pierre Finance
CRIN, BP 239, 54506 Vandoeuvre-les-Nancy Cédex, France

Communicated by M. Sintzoff
Received January 1988
Revised February 1991

Abstract


The design of parallel programs needs to specify the way parallel components cooperate. This has to be progressively derived as systematically as possible from abstract specifications. In this paper we introduce convenient specification tools and a parallelism model for such development expressions. The innovation we propose is the concept of communication relations, which can be considered

• on the one hand, as a powerful parallelism abstraction to capture the semantics of cooperation between communicating processes,
• on the other hand, as a paradigm for systematic parallel program design.

This concept of communication relations is defined in the first part of this paper. In the second part we show how this concept can be used as a paradigm for parallel program design and we present a notation for the statements and a refinement technique to transform these statements. Four fundamental rules underly this technique, which is then applied to the development of two examples. Very simple examples have been chosen, but they are suggestive enough to show the main steps of a methodological approach.

Introduction

The literature is very rich in parallel programming, see Hoare [16], Keller [18], and Milner [26, 27] on parallelism models; Chen and Yeh [9], Halpern and Moses

Correspondence to: G.-R. Perrin, Laboratoire d'Informatique, Université de Franche-Comté, 25030 Besançon Cédex, France. E-mail: perrin@loria.crin.fr
and Lamport [21] on specification; Apt et al. [1] and Pnueli [30] on proof systems; and Chandy and Misra [7], Kung [20], and [31] on algorithms; etc.


Our purpose is to make a concrete contribution to this topic. The main difficulty in inventing parallel programs is to specify the way parallel components cooperate and especially how they are synchronized. These features have to be progressively derived as systematically as possible from abstract specifications. The challenge is then to introduce relevant specification tools and a convenient parallelism model for such development expressions, according to appropriate techniques.

The innovation we put forward here is the concept of communication relations, which has to be considered

- on the one hand, as a powerful parallelism abstraction to capture the semantics of cooperation between communicating processes,
- on the other hand, as a paradigm for systematic parallel program design.

Let us consider some mathematical notions and the following abstract functional definition:

\[ y = f(x), \]  

where \( x \) and \( y \) are possibly infinite sequences of values. This definition means that for any occurrence of index \( k \) the value \( f(x[k]) \) is assigned to \( y[k] \) (cf. the map operator on lists in [4] or the variable definitions in LUCID [2]). Imagine two functions \( g \) and \( h \) are known, such that \( f = g \circ h \). The sequence \( y \) can then be rewritten as

\[ y = g \circ h(x). \]  

To return to programs, we can express a program as a set of unordered definitions of variables, assuming that we know how to define denotational semantics in functional terms. Therefore, to target the previous mathematical notions, we define a variable as a possibly infinite sequence of values. A variable definition, also called an equation, of the form

\[ y = f(x) \]

allows us to substitute throughout the program the right expression \( f(x) \) for its name \( y \). Such an expression uses variable names or what we call data. Data are either arguments in the program or variable occurrences.
For example, let us consider the following definitions:

\[ y = g(t), \]
\[ s = h(x), \]
\[ t = s, \]

where \( t, s, x, \) and \( y \) are variables. According to the substitution rule we can define the denotational semantics of this program as \( g \circ h \), i.e. as the previous \( f \).

From an operational point of view, this program only seems to be a more explicit form for a compositional notation \( y = g(h(x)) \), which was not explicitly defined in our mathematical notation. Hence, it should be interpreted as a recurrence which maps any \( (y[k-1], s[k-1], t[k-1]) \) to

\[
\begin{align*}
(y[k-1], h(x[k \text{ div } 3]), t[k-1]), & \quad \text{if } k = 0 \mod 3, \\
(y[k-1], s[k-1], s[k-1]), & \quad \text{if } k = 1 \mod 3, \\
g(t[k-1]), s[k-1], t[k-1]), & \quad \text{if } k = 2 \mod 3.
\end{align*}
\]

Concerning parallel interpretations, which is our primary topic here, it is rather common to associate a process network with the system (3) (see for example [17]). The processes

\[
G :: y[k+1] = g(t[k]) \quad \text{for any } k, \\
H :: s[k+1] = h(x[k]) \quad \text{for any } k,
\]

communicate messages via a FIFO medium (which implements the relation \( t = s \)), in the following network:

![Process network](image)

This simple presentation can serve as a guide for parallel programming; it defines a classical paradigm (introducing intermediate variables and folding definitions) supported by a parallelism model.

Communication relations are much more expressive abstractions. As an introduction to this concept, let us consider this trivial generalization of system (3):

\[
\begin{align*}
y &= g(t), \\
s &= h(x), \\
t &= s,
\end{align*}
\]

where the symbol \( \subset \) may be interpreted in various ways to define a channel between two processes: either the communication traces [28] are ordered [33] or \( t \) is any extracted sequence of \( s \) (for example: \( t = s \) when \( s \not< 0 \), in a LUSTRE notation [5]), etc.
According to a development process which must be rationalized, such an expression avoids confusing purely functional definitions and some operational aspects associated with parallelism interpretations. Moreover, it leads to a fruitful development tactic. To deal with more sophisticated communication or synchronization situations, we propose the following straightforward generalization:

\[
y = g(t), \\
s = h(x), \\
(s, t) \in \mathcal{R},
\]

(5)

where \(\mathcal{R}\), called a communication relation, is any relation expressed by a first-order logic formula over variable histories. As do many researchers (see, for example, Finance and Jaray [11], Perrin [29] or Caspi et al. [5]), we consider that these histories may handle value sequences and the sequences of instants when the values are assigned.

One aim of this paper is to convince the reader that such an expression is a suitable tool for parallel program development. It is an element of a design tactic which is based on the following steps:

- definition of variables and functions, as a formal specification of the problem,
- from this specification, introduction of definitions and communication relations by stepwise refinements.

We shall propose in the next section an adequate modeling for these expressions. Practical aspects (notations, refinements, and development processes) are presented in Section 2. Finally, Section 3 is devoted to two examples, which are discussed beginning from a specification and ending with some concrete solutions in the form of communicating processes.

1. Parallel program modeling

A parallel program may be described as a process network, the elements of which interact by message communication. Figure 1 illustrates such a program, composed of two processes P1 and P2; \(s\) is a message stream for output and \(t\) is for input.

The definition of such a program consists of process definitions and specification of the relation between the streams \(s\) and \(t\), which can be considered as characteristic of the underlying parallelism model, for example, an asynchronous communication without loss, spontaneous generation, duplication, or reordering of messages (cf. the example given in the introduction).

![Fig. 1. A parallel program.](image)
To focus our attention on synchronization derivation, we consider a concurrent program modeling with a very strict rule for message passing: the communication is synchronous along a channel in a *rendez-vous* situation (see, for example, [33]). Then—in such a model—to express a program such as the one suggested in the previous example, we have to introduce a *filter* medium process $F$, which implements a *FIFO* policy. In fact, because the *FIFO* structure is to be shared by $P_1$ and $P_2$, this definition requires the introduction of a particular stream, called a *request stream*, as shown in Fig. 2; the filter process $F$ reacts to the *request sequence* $\delta$ by passing, at once or soon after, the appropriate messages to the process $P_2$, which may be considered as a part of the *environment* of $F$. $P_2$ and $F$ comprise a *reactive system*, as defined by Harel and Pnueli [15].

![Fig. 2. A reactive system.](image)

Such a process network is then an implementation of the equality $t = s$, i.e. of a trivial communication relation, in a parallelism model whose communication primitive is the *rendez-vous*. Note that a direct synchronization point between $P_1$ and $P_2$ should be another implementation of $t = s$, in some sense *overloaded* relative to its specification. One interest of specifying communication relations and deriving solutions in some parallelism modeling by a stepwise refinement is precisely to discuss synchronization tools in regard to the specified relation.

In the following, we present a parallel program model in which the interaction between processes is only defined by communication relations, based on variable histories.

### 1.1. Variables, data, and equations

The semantics of a *variable* are a possibly infinite sequence of values in a set specified by the variable type. If we note this set as $\text{ELT}$, a variable $x$ is then a mapping $x : \mathbb{N} \rightarrow \text{ELT}$, whose *domain* is finite or infinite. Each element of this sequence, called a *variable occurrence*, can be selected by an index in the domain of the variable.

We call *data* either the notation of some argument of a problem or some variable occurrence. It is defined by a statement, called henceforth a *data definition*, of the form

$$n \overset{\text{def}}{=} f(\ldots, x[k], \ldots),$$

or to specify some variable occurrence

$$y[i] \overset{\text{def}}{=} f(\ldots, x[k], \ldots).$$
Note that we admit recurrence data definitions, in which the same variable name occurs on each side of the symbol \( \text{def} \), assuming that classical calculability rules are satisfied.

An equation defines a variable \( y \) from a function, for example \( f \), depending on variables (for example \( x, \ldots \)) or data (for example \( n, \ldots \)) as follows:

\[
y == f(\ldots, x, n, \ldots).
\]

Such an equation means that for any occurrence of index \( k \), the value \( f(\ldots, x[k], n, \ldots) \) is assigned to \( y[k] \). In other words, the symbol "==" links two identical expressions whose occurrences may be substituted for each other. Of course, a function is some composition of operations that is supposed to be defined in its argument types.

A program is a set of data definitions and equations, possibly completed by the definition of communication relations.

**Examples**

1. The integer set is defined by

\[
x[0] \text{def} 0,
\]

\[
x[k] \text{def} x[k-1] + 1, \quad k > 0.
\]

2. The integer square set is defined by

\[
y == x \times x.
\]

3. The set of factorials is defined by

\[
f[0] \text{def} 1,
\]

\[
f[k] \text{def} f[k-1] \times k, \quad k > 0.
\]

4. \( \text{gcd}(a_0, \ldots, a_n) \) can be deduced from the following definitions:

\[
z_p[0] \text{def} a_p,
\]

\[
z_p[k-1] < z_p[k] \rightarrow z_p[k] \text{def} z_p[k-1] - z_{p-1}[k-1],
\]

\[
z_p[k-1] \geq z_p[k-1] \rightarrow z_p[k] \text{def} z_p[k-1],
\]

\[
k > 0, \quad p = 0, 1, \ldots, n.
\]

\( k - 1 \) stands for \( k \) minus 1 modulo \( n + 1 \).

1.2. Communication relations

Intuitively data definitions and equations model the processes of parallel interpretation. Communication relations model the interaction between the processes. Their definition generally requires the specification of abstract instants. To handle time in a simple and natural way, let us consider an infinite totally ordered (by a relation \(<\) and its widening \(\leq\)) set, called CHRONO.
Definition 1. Any growing mapping $\delta : D \subseteq \mathbb{N} \to \text{CHRONO}$ is called a *clock*.

Let ELT be any set. For any mapping $x : \mathbb{N} \to \text{ELT} \times \text{CHRONO}$, we note the projections:

- $\text{value}(x) : \mathbb{N} \to \text{ELT}$,
- $\text{instant}(x) : \mathbb{N} \to \text{CHRONO}$.

**Definition 2.** A *variable history* of a variable is a mapping $x : \mathbb{N} \to \text{ELT} \times \text{CHRONO}$ such that $\text{instant}(x)$ is a clock.

Note the analogy with the variables in the language LUSTRE [5], which are interpreted according to a synchronous parallelism model. However asynchronous operators (when or current) allow asynchronous definitions of variables, as in the model we present here.

**Definition 3.** A *communication relation* (of type ELT) is any binary relation $\mathcal{R}$ on the set $\Sigma$ of variable histories of type ELT, defined as follows:

$$(s, t) \in \mathcal{R} \quad \text{iff} \quad \forall j \in T \exists i \in S \text{ such that }$$

- $\text{value}(t)[j] = \text{value}(s)[i]$,
- $\text{instant}(t)[j] \geq \text{instant}(s)[i]$,
- $\tau(i, j)$,

where $T$ is the domain of $t$, $S$ is the domain of $s$, and $\tau$ is a predicate over terms of a multisorted algebra, which includes the following sorts:

- $\mathbb{N}$,
- the polymorphic set $\Sigma$ of variable histories of type ELT ($x, y, \ldots$ denote the variables in this set),
- the set of clocks ($\alpha, \beta, \ldots$ denote the variables in this set).

**Note.** $\tau$ is a characteristic of $\mathcal{R}$.

Let $\mathcal{R}$ be a communication relation associated with the predicate $\tau$. For any given variable history $s$ (of domain $S$), there exist infinitely many variable histories $t$ such that $(s, t) \in \mathcal{R}$. Given any variable history $s$ and any relation $\mathcal{R}$, the following definitions are designed to construct a particular history $t$ such that $(s, t) \in \mathcal{R}$.

**Definition 4.** Let $\delta$ be a clock (of domain $D$), and $s$ a variable history (of domain $S$). We define the following variable history $t$ (of domain $T = D$), which is obviously such that $(s, t) \in \mathcal{R}$:

$$\forall j \in T \quad \text{value}(t)[j] = \text{value}(s)[i],$$

$$\text{instant}(t)[j] = \max(\delta[j], \text{instant}(s)[i]),$$

where $i \in S$ is such that $\tau(i, j)$. 
Definition 5. If $\forall j \in T \ \text{instant}(t)[j] < \delta[j + 1]$, then $t$ is called a consumption history relative to the production history $s$ and to the request history $\delta$, for the communication relation $\mathcal{R}$. The function $\varphi : T \rightarrow S$ defined by $\varphi(j) = i$ is called the communication trace function of $t$ in $s$, relative to $\delta$.

Examples.
(1) We consider the following predicate:

\[
\text{equality}(i,j) = j \in T \land i \in S \land i = j.
\]

The communication relation associated with this predicate defines a process network such that the communication has neither loss nor spontaneous generation, nor duplication, nor reordering of messages, according to a request sequence, as shown in Fig. 2. This may be shown as in Fig. 3 with variable values along lines which represent time.

\[
\begin{array}{cccccc}
\text{value}(s)[1] & \text{value}(s)[2] & \ldots & \text{value}(s)[i] & \ldots \\
\delta[1] & \delta[2] & \ldots & \delta[i] & \ldots \\
\text{wait} & \text{wait} & \ldots \\
\text{value}(t)[1] & \text{value}(t)[2] & \ldots & \text{value}(t)[i] & \ldots \\
\end{array}
\]

Fig. 3.

(2) We consider the following predicate:

\[
\text{refresh}(i,j) = j \in T \land i \in S \land
\begin{align*}
(j &= 1 \lor \forall i' \in S \ (\text{refresh}(i', j - 1) \Rightarrow i' < i)) \land \\
\forall i' \in S \ (\text{instant}(s)[i'] \leq \text{instant}(t)[j] \Rightarrow i' \leq i).
\end{align*}
\]

The communication relation associated with this predicate defines a process network such that each element of the consumption history is the last element of the production history at the request instant, assuming that at least one producing operation occurs between two consuming ones, as shown in Fig. 4.

Note. These definitions may be generalized by considering an $(n + 1)$-ary relation $\mathcal{R}$ on $\Sigma$, defined as follows:

\[
(s_1, s_2, \ldots, s_n, t) \in \mathcal{R}
\]

iff $\forall j \in T \ \exists (i, k) \in (S_1 \cup S_2 \cup \cdots \cup S_n) \times [1, n]$ such that

- $\text{value}(t)[j] = \text{value}(s_k)[i]$,
- $\text{instant}(t)[j] \geq \text{instant}(s_k)[i]$,
- $\tau((i, k), j)$.

Such a communication relation defines the process network illustrated in Fig. 5.
1.3. Example

We illustrate the preceding notions with a very simple example. Let us consider the calculus of factorial \( n \), for any positive \( n \). We can imagine two processes \( F \) and \( G \) which each compute a part of the result. Assuming that the process \( F \) computes incrementally \( 1 \times 2 \times \cdots \times (lb - 1) \) and the process \( G \) computes \( n \times (n-1) \times \cdots \times (ub + 1) \), these processes may run in parallel while they preserve the global invariant:

\[
\prod_{p=1}^{lb-1} p \land g = \prod_{p=ub+1}^{n} p \land lb < ub.
\]

The final state is defined by \( lb \geq ub \), from which we can deduce:

\[
(lb = ub \land n! = f \times g \times lb) \lor (lb > ub \land n! = f \times g).
\]

The interactions between \( F \) and \( G \) can be deduced from the following observation: let us consider the increasing production history \( lb \) of \( F \), and a \( j \)th request instant \( \delta[j] \) of \( G \), for which the current value of \( ub \) is \( ub[j] \) (see Fig. 6), then we have:

\[
\text{value}(lb)[k] < \text{value}(ub)[j] \Rightarrow \text{value}(lb)[i] < \text{value}(ub)[j] \quad \forall i \leq k.
\]

Hence, any \( i \leq k \) preserves the invariant, and then any communication relation is correct.

This straightforward conclusion will be deduced more formally in the final section. Communications equality possibly induce some duplication of the calculations. The
communication random, defined below, minimizes this inefficiency. It defines a consumption history such that each consumed element is the last of the produced one at the request instant:

\[
\text{random}(i, j) = \exists j \in T \land i \in S \land \\
\forall i' \in S (\text{instant}(s)[i'] \leq \text{instant}(t)[j] \Rightarrow i' \leq i).
\]

Here is an intuitive running trace example whose formal proof will be given in the final section:

\[
\begin{align*}
&f = 1, \quad lb = 1 & g = 1, \quad ub = 5 \\
&\text{send}(lb, f) & \text{send}(ub, g) \\
&1 < 5: \quad f = 1, \quad lb = 2 & 1 < 5: \quad g = 5, \quad ub = 4 \\
&\text{receive}(5, g) & T \\
&\text{send}(lb, f) & I \\
&2 < 5: \quad f = 1 \times 2, \quad lb = 3 & 1 < 4: \quad g = 5 \times 4, \quad ub = 3 \\
&\text{send}(lb, f) & M \\
&3 < 5: \quad f = 1 \times 2 \times 3, \quad lb = 4 & \text{receive}(4, g) \\
&\text{send}(lb, f) & E \\
&4 = 4: & \text{compute}(f \times g \times 4) \\
&\text{receive}(4, f) & \text{compute}(f \times g)
\end{align*}
\]

1.4. Communication and synchronization

Communication relations specify the interactions between the processes computing the data or the variable definitions. They induce synchronization constraints of varying importance for the processes. Concerning process construction and possibly efficiency considerations, a few results from these synchronization effects seem very significant.

1.4.1. Communication automaton

With any communication relation, we associate an automaton which defines the transitions of a function network which is assumed to be reduced to the production and consumption operations satisfying this communication relation.
Example. A producer-consumer network modeling the equality communication relation is such that

\[ \text{number of productions} \geq \text{number of consumptions} \]

is an invariant property. The associated automaton is shown in Fig. 7.

Definition 6. A communication automaton is a graph labeled by the indexes in the production and the consumption histories: a jth production edge from the root is labeled $P_j$, an ith consumption edge is labeled $C_i$, such that $\varphi(j) = i$.

Example. The communication automaton of the random relation is shown in Fig. 8.
1.4.2. Partial order

**Definition 7.** We define a partial order as $\preceq$ on the communication relation set as:

$\text{com}_1 \preceq \text{com}_2$

iff

- the graph associated with $\text{com}_1$ is a subgraph of the graph associated with $\text{com}_2$,
- on the corresponding edges the labels of $\text{com}_1$ are inferior to the labels of $\text{com}_2$, i.e. $\forall j \varphi_1(j) \leq \varphi_2(j)$, where $\varphi_1$ and $\varphi_2$ are the communication traces of $\text{com}_1$ and $\text{com}_2$.

This order is a convenient framework to transform solutions by changing their synchronization constraints. We will use it in the following sections.

2. Parallel program design

2.1. The CBS notation

This section is devoted to the introduction of a notation to express successive statements beginning with a specification and ending with a program. This description is made by using a first-order language including:

- an extended notation of the predicate calculus,
- the definition of intermediate mathematical functions.

Data, variable, and function types are explicitly given in these definitions. Definitions may be either predicates or constructive functions. The development process leads progressively to refining predicates by constructive functions in very well-known ways, which are not our main concern.

On the contrary, refinements which lead to introducing a parallel program, i.e. a set of definitions, equations, and communication relations, will be the main subject considered here. The notation we propose to unify these successive expressions, called CBS for Communication-Based Statements, is defined by the following BNF grammar:

$$(\text{statement}) ::= \text{statement} (\text{id}) (\text{declarations})$$

$$(\text{declarations}) ::= [\text{data} (\text{data-list})] [\text{variables} (\text{variable-list})]$$

$$(\text{definition_part}) ::= \text{definitions} (\text{equation})\{(\text{equation})\} (\text{intermediate})$$

$$(\text{equation}) ::= \{(\text{guard})\} (\text{alternative})$$

$$(\text{alternative}) ::= (\text{id})(\text{link})(\text{expression}) (\text{(repetition)}) | (\text{id}) \text{ such that } (\text{formula}) (\text{(repetition)})$$

$$(\text{link}) ::= \overset{\text{def}}{=} | \overset{\text{id}}{=}$$

$$(\text{intermediate}) ::= \text{with} \{ (\text{head}) \text{ defined by } (\text{body}) \}$$
A statement begins with the word "statement", followed by its name, the list of its typed data and variables, its (definition_part) and possibly its (communication_part). The definition of a datum or a variable, depending on other data or variables, consists of either one or more functional expressions, or of one or more (possibly guarded) predicates: this (expression) or this (formula) may use some intermediate definitions, given in a clause indicated by the word "with" followed by the appropriate (body). If the data or the variables are vectors, their definition may depend on a component index in some (repetition) slice. The (link) between some identificator and its expression is:

- either a symbol "=" to define a variable by an equation,
- or a symbol "def" to define a datum,
- or a symbol "==" followed by some predicate identifier to express a communication relation.

Each communication relation is defined from a (predicate) in the language presented in the previous section (see examples following Definition 5 above).

**Example.** Section 3 will illustrate these points with complete examples. Here, we only present a statement corresponding to a sample step for the previous factorial problem.

```
statement factorial,
data n, fact: integer,
variables
   lb, ub, f, g : integer,
   lb', ub', f', g': integer,
definitions
   k > 0 → lb[k] def k,
   k > 0 → ub[k] def n - k + 1,
   f = \prod_{p=1}^{lb-1} p,
   g = \prod_{p = ub+1}^{n} p,
   lb' = lb,
   ub' = ub,
   f' = f,
   g' = g,
```
2.2. Refinement

Program development, and especially parallel program development, requires a stepwise refinement technique. In a general way, a program is said to refine a specification if it satisfies this specification. The satisfaction property refers to partial correctness.

Abstractly speaking, we will say that a statement $S_2$ in CBS is a refinement of a statement $S_1$ if for any resulting data or variable $x$ (with domain $X$) appearing in $S_1$, $x$ is defined in $S_2$ in the ways shown in Table 1.

Well-known refinement steps consist of:

- substituting variables for constants,
- adding intermediate data, variables, or functions,
- folding or unfolding the definitions according to the substitution rule in CBS.

Table 1

<table>
<thead>
<tr>
<th>Definitions in $S_1$</th>
<th>Definitions in $S_2$</th>
<th>Under the following conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \stackrel{\text{def}}{=} y$</td>
<td>$x \stackrel{\text{def}}{=} z$</td>
<td>$y = z$</td>
</tr>
<tr>
<td>$x == y$</td>
<td>$x == z$</td>
<td>$\forall k \in X \ y[k] = z[k]$</td>
</tr>
<tr>
<td>$x$ such that $P_1$</td>
<td>$x \stackrel{\text{def}}{=} y$</td>
<td>$x = y \Rightarrow P_1$</td>
</tr>
<tr>
<td></td>
<td>$x == y$</td>
<td>$(\forall k \in X \ x[k] = y[k]) \Rightarrow P_1$</td>
</tr>
<tr>
<td></td>
<td>$x$ such that $P_2$</td>
<td>$P_2 \Rightarrow P_1$</td>
</tr>
</tbody>
</table>

2.3. Paradigms and tactics for parallel program design

In this section we are concerned with particular situations in which these first steps are used, along with other steps, to develop parallel programs. These paradigms and their schemes characterize some tactics used in program derivations.

Rule 1. Let $S_1$ be:

$$y == g(z), \quad z == h(x),$$

where $x$, $y$, and $z$ are variables. Let $S_2$ be:

$$y == g(t), \quad s == h(x), \quad t == s,$$

then $S_2$ is a refinement of $S_1$. 

$$k > 0 \land lb'[k] = ub'[k] + 1 \Rightarrow \text{fact} \ \overset{\text{def}}{=} f'[k] \times g'[k],$$

$$k > 0 \land lb'[k] = ub'[k] \Rightarrow \text{fact} \ \overset{\text{def}}{=} f'[k] \times g'[k] \times lb'[k],$$

communications

equality($i,j$) = $j \in T \land i \in S \land i = j$. 

$$A \cdot = A \cdot + \text{fact} \cdot \cdot x A \cdot,$$
Proof. Rule 1 follows from the substitution rule and the equivalence of \( t = s \) and \( k > 0 \rightarrow t[k] \overset{\text{def}}{=} s[k] \).

**Rule 2.** Let \( S_1 \) be:

\[
y = g(t), \quad s = h(x), \quad t = s,
\]

where \( s, t, x, \) and \( y \) are variables. Let \( S_2 \) be:

\[
y = g(t), \quad s = h(x), \quad t^{\text{equality}} = s,
\]

then \( S_2 \) is a refinement of \( S_1 \).

Proof. This rule follows from the definition of the *equality* communication relation.

**Rule 3.** Let \( S_1 \) be:

\[
y = g(t_1), \quad s = h(x), \quad t_1^{\text{com1}} = s.
\]

Let \( S_2 \) be:

\[
y = g(t_2), \quad s = h(x), \quad t_2^{\text{com2}} = s,
\]

then \( S_2 \) is a refinement of \( S_1 \) if \( \text{com2 as}< \text{com1} \).

Proof. From the definition of the partial order \( \text{as}< \), any communication trace for \( S_2 \) is a particular communication trace for \( S_1 \). We can deduce that if \( t_1 \) satisfies some property, so does \( t_2 \).

Note that the reverse transformation is generally false. Nevertheless, some particular properties of the variables, combined with those of the communication relations, allow a kind of *weakening*, in which a less synchronous communication relation is substituted for a given one. This is, in fact, one of the very fruitful development tactics we present below.

**Rule 4.** Let \( S_1 \) be:

\[
y = \text{expr}, \quad r \overset{\text{def}}{=} y[p^*],
\]

where \( y \) is a variable, and \( r \) and \( p^* \) are data. Let \( S_2 \) be:

\[
x_k = \text{expr}, \quad k \in [1, n],
\]

\[
r \overset{\text{def}}{=} x_k[p^*],
\]

where \( x \) is a variable of type vector\((n)\), then \( S_2 \) is a refinement of \( S_1 \).

Proof. Trivial.
This refinement rule leads to particular parallel solutions in which an $n$-tuple of variables has the same definition, modeling $n$ abstract identical concurrent processes.

Two main development tactics can then be proposed for parallel programming. They are illustrated in the next section with two examples. The first tactic consists of applying Rules 1 and 2 to introduce an asynchronous function network via equality communication relations. Then by weakening these relations we can derive other solutions which must be proved. The second one consists of applying Rule 4 as soon as possible to introduce variables of type vector. The application of these rules lead to identical concurrent processes. These solutions can be simple vectorized programs. In some situations, these concurrent processes can run asynchronously; these new solutions must then be proved.

3. Examples

3.1. Factorial

From the following statement, defining the factorial of some positive $n$, we introduce a few reasoning steps to illustrate the main paradigms and tactics in parallel program development.

\begin{verbatim}
statement factorial-1,
  data n, fact: integer,
  definitions
  fact = \prod_{p=1}^{n} p.
\end{verbatim}

By introducing a variable $x$, such that its $n$th occurrence is equal to “fact”, we specify a recurrent process for the product calculation.

\begin{verbatim}
statement factorial-2,
  data n, fact: integer,
  variables x: integer,
  definitions
  x[1] = 1, 
  p > 0 \rightarrow x[p+1] = x[p] \times (p + 1), 
  fact = x[n].
\end{verbatim}

This recurrent definition can be generalized by introducing subdomains, $\text{fact} = x_1 \times x_2 \times \cdots \times x_p$, such that every $x_i$ computes a part of the product. For example, we can use the following property:

$$\forall b \in [1, n] \quad \text{fact} = \prod_{p=1}^{b} p \times \prod_{p=b+1}^{n} p = \prod_{p=1}^{b-1} p \times \prod_{p=b+1}^{n} p \times b.$$
By substitution of the variables \( lb \) (for *lower bound*) and \( ub \) (for *upper bound*) for the constant \( b \), we obtain statement factorial-3, which satisfies the global invariant described in Section 1.3:

**statement** factorial-3,
**data** \( n, \text{fact: integer}, \)
**variables** \( lb, ub, f, g : \text{integer}, \)
**definitions**
\[
\begin{align*}
  k > 0 & \rightarrow lb[k] \overset{\text{def}}{=} k, \\
  k > 0 & \rightarrow ub[k] \overset{\text{def}}{=} n - k + 1, \\
  f & = \prod_{p=1}^{lb-1} p, \\
  g & = \prod_{p=ub+1}^{n} p, \\
  k > 0 & \land lb[k] = ub[k] + 1 \rightarrow \text{fact} \overset{\text{def}}{=} f[k] \times g[k], \\
  k > 0 & \land lb[k] = ub[k] \rightarrow \text{fact} \overset{\text{def}}{=} f[k] \times g[k] \times lb[k].
\end{align*}
\]

Factorial-3 is an actual statement from which a tactic to develop a parallel solution can be applied. This tactic consists of three steps. In the first one independent processes are discovered or introduced by duplicating some variables in new equations. In the second step *equality* communication relations are substituted for these equations. In the last one some other communication relations are substituted for these *equality* relations.

For this example, the pairs of variables \((lb,f)\) and \((ub,g)\) are independent; they can lead to natural distributed processes. On the contrary, a process computing the resulting "fact" needs these four variables. Therefore, because of the substitution rule in CBS (Refinement Rule 1), four new copies of these variables can be introduced, preserving the correctness property.

**statement** factorial-4,
**data** \( n, \text{fact: integer}, \)
**variables**
\[
\begin{align*}
  lb, ub, f, g & : \text{integer}, \\
  lb', ub', f', g' : \text{integer},
\end{align*}
\]
**definitions**
\[
\begin{align*}
  k > 0 & \rightarrow lb[k] \overset{\text{def}}{=} k, \\
  k > 0 & \rightarrow ub[k] \overset{\text{def}}{=} n - k + 1, \\
  f & = \prod_{p=1}^{lb-1} p, \\
  g & = \prod_{p=ub+1}^{n} p, \\
  lb' & = lb, \\
  ub' & = ub,
\end{align*}
\]
\[ f' = f, \]
\[ g' = g, \]
\[ k > 0 \land lb'[k] = ub'[k] + 1 \Rightarrow \text{fact} f'[k] \times g'[k], \]
\[ k > 0 \land lb'[k] = ub'[k] \Rightarrow \text{fact} f'[k] \times g'[k] \times lb'[k]. \]

By applying one of the main techniques, substitute *equality* communications relations for the new equations (Refinement Rule 2), we obtain statement factorial-5, presented in Section 1.4:

**statement** factorial-5,

**data** \( n, \text{fact} : \text{integer} , \)

**variables**

\( lb, ub, f, g : \text{integer} , \)
\( lb', ub', f', g : \text{integer} , \)

**definitions**

\[ k > 0 \Rightarrow lb[k] \overset{\text{def}}{=} k , \]
\[ k > 0 \Rightarrow ub[k] \overset{\text{def}}{=} n - k + 1 , \]
\[ f = \prod_{p=1}^{n} p , \]
\[ g = \prod_{p=ub+1}^{n} p , \]
\[ lb' \overset{\text{equality}}{=} lb , \]
\[ ub' \overset{\text{equality}}{=} ub , \]
\[ f' \overset{\text{equality}}{=} f , \]
\[ g' \overset{\text{equality}}{=} g , \]
\[ k > 0 \land lb'[k] = ub'[k] + 1 \Rightarrow \text{fact} f'[k] \times g'[k] , \]
\[ k > 0 \land lb'[k] = ub'[k] \Rightarrow \text{fact} f'[k] \times g'[k] \times lb'[k] . \]

**communications**

\[ \text{equality}(i, j) = j \in T \land i \in S \land i = j . \]

Such a statement leads to a rather poor parallel solution in which the processes \( f \) and \( g \) each compute half of the final product. For example, for \( n = 6 \), we have:

<table>
<thead>
<tr>
<th>( k )</th>
<th>( f )</th>
<th>( lb )</th>
<th>( lb' )</th>
<th>( g )</th>
<th>( ub )</th>
<th>( ub' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>20</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>60</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>5</td>
<td>5</td>
<td>120</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>120</td>
<td>6</td>
<td>6</td>
<td>120</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
For \( k = 3 \), we have \( lb'[k] = ub'[k] \), then \( \text{fact} = 2 \times 20 \times 3 \).

A significantly more asynchronous solution could appear as more satisfactory. But note that the introduction of more asynchronous communication relations such as \textit{random} ones, substituted for \textit{equality} relations, would be a mistake. This is proved by the following case: if \( lb'[1] = lb[5] = 5 \) and \( ub'[1] = ub[4] = 2 \), then no \( k > 0 \) satisfies the boolean conditions of the guards.

Nevertheless, a variant can be proposed by grouping the definitions of \( lb, f, \) and "fact", on the one hand, and the definition of \( ub, g, \) and "fact" on the other:

\begin{verbatim}
statement factorial-6,
data n, fact: integer,
variables
  lb, ub, f, g: integer,
  lb', ub', f', g': integer,
definitions
  k > 0 -> lb[k] \( \overset{\text{df}}{=} \) k,
  k > 0 -> ub[k] \( \overset{\text{df}}{=} \) n - k + 1,
  f = \( \prod_{i=1}^{n} \) p,
  g = \( \prod_{p=ub+1}^{n} \) p,
(ub', g') \( \overset{\text{random}}{=} \) (ub, g),
  k > 0 \& lb[k] = ub'[k] + 1 \rightarrow \text{fact} \( \overset{\text{df}}{=} \) f[k] \times g'[k],
  k > 0 \& lb[k] = ub'[k] \rightarrow \text{fact} \( \overset{\text{df}}{=} \) f[k] \times g'[k] \times lb[k],
(lb', f') \( \overset{\text{random}}{=} \) (lb, f),
  k > 0 \& lb'[k] = ub[k] + 1 \rightarrow \text{fact} \( \overset{\text{df}}{=} \) f'[k] \times g[k],
  k > 0 \& lb'[k] = ub[k] \rightarrow \text{fact} \( \overset{\text{df}}{=} \) f'[k] \times g[k] \times lb'[k],
communications
  random(i, h) \equiv j \in T \& i \in S \& \forall i' \in S \ (\text{instant}(s)[i'] \leq \text{instant}(t)[j] \Rightarrow i' \leq i).
\end{verbatim}

\textbf{Proof.} In the first step of the proof we note that if \textit{equality} communication relations are substituted for the \textit{random} ones, factorial-6 is equivalent to factorial-5. Hence, because of the semantics of the \textit{equality} communication relation, the grouping of variables does not affect the definitions.

The rest of the proof establishes that at least one of the conditions in the four guards is true for some \( k > 0 \). Hence let us consider, for example, the variables \( lb \) and \( ub' \) appearing in the first two guards. Variable \( lb \) defines a strictly increasing sequence of integers, and \( ub' \) a decreasing one. Let \( k^* \) be the least occurrence \( k \) such that \( lb[k] \geq ub'[k] \). From the definition of \( lb \) we then have:

\begin{equation}
  k^* - 1 < ub'[k^* - 1]
\end{equation}

and \( lb[k^*] \geq ub'[k^*] \).
If \( \text{lb}[k^*] = \text{ub}'[k^*] \), the second guard condition is satisfied. If \( \text{lb}[k^*] = \text{ub}'[k^*] + 1 \), the first one is satisfied. It is necessary to develop the case where \( \text{lb}[k^*] > \text{ub}'[k^*] + 1 \), i.e.

\[
k^* - 1 > \text{ub}'[k^*].
\]

(2)

From (1) and (2) we deduce:

\[
\text{ub}'[k^*] < k^* - 1 < \text{ub}'[k^* - 1],
\]

(3a)

i.e.

\[
\text{ub}'[k^*] < \text{ub}'[k^* - 1] - 1.
\]

(3b)

Let \( k^- \) and \( k^{**} \) be such that

\[
\text{ub}'[k^* - 1] = \text{ub}[k^-] \quad \text{and} \quad \text{ub}'[k^*] = \text{ub}[k^{**}].
\]

From (3) and the definition of \( \text{ub} \), we obtain:

\[
k^{**} > k^- + 1.
\]

(4)

Then the index interval \([k^- + 1, k^{**}]\) is not empty and from the definition of the predicate \( \text{random} \) we deduce:

\[
\forall k \in [k^- + 1, k^{**}] \quad \text{lb}'[k] = \text{lb}[k^* - 1],
\]

(5a)

i.e.

\[
\text{lb}'[k] = k^* - 1.
\]

(5b)

The \( \text{lb}'[k] \) are then equal in this index interval, and the \( \text{ub}[k] \) decrease with a step equal to 1.

From the following two lemmas we deduce that \( \exists k \in [k^- + 1, k^{**}] \) such that \( \text{ub}[k] = \text{lb}'[k] \). Therefore, in this case the last guard condition is satisfied.

**Lemma 1.** \( \text{ub}[k^- + 1] \geq k^- - 1 \).

**Proof.** From (3a) we have \( k^- - 1 < \text{ub}[k^-] \), i.e. \( k^- - 1 < n - k^- + 1 \) from the definition of \( \text{ub} \). From this same definition, we have \( \text{ub}[k^- + 1] = n - k^- \). Thus, \( \text{ub}[k^- + 1] > k^- - 2 \). \( \square \)

**Lemma 2.** \( \text{ub}[k^{**}] < k^- - 1 \).

**Proof.** From (3a) it follows that \( \text{ub}[k^{**}] = \text{ub}'[k^*] < k^- - 1 \). \( \square \)

This last statement is a refinement of the first one, which is obtained by weakening the initial property:

\[
\exists k > 0 \quad (\text{lb}[k] = \text{ub}[k] + 1 \lor \text{lb}[k] = \text{ub}[k]).
\]
The new one is:

\[
\exists k > 0 \quad (lb'[k] = ub'[k] + 1 \lor lb'[k] = ub'[k] \lor lb'[k] = ub[k] + 1 \lor lb'[k] = ub[k]).
\]

This statement leads to a solution in terms of communicating processes, whose trace, which is given in Section 1.3, is an instance of computation.

3.2. Function root

Let \( f \) be a real continuous function defined on the closed interval \([a, b]\) and assume that \( f(a).f(b) \leq 0 \), i.e. \( f \) has at least one root in \([a, b]\). The problem is to design an algorithm for finding a root of \( f \).

We focus our attention on the development of a few ways to achieve different solutions from the initial statement of the problem. These development approaches display the fundamental paradigms of parallel reasoning. They are organized in four stages:

1. problem formulation,
2. the first approach,
3. the second approach,
4. transformation techniques.

3.2.1. The problem formulation

**Step 1. Define the problem terms.** First, let us specify the problem terms, using the CBS notation.

<table>
<thead>
<tr>
<th><strong>statement</strong> root-1,</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data</strong></td>
</tr>
<tr>
<td>( a, b, x^* : ) point — the interval bounds and the result,</td>
</tr>
<tr>
<td>( f : ) point → real — the given function,</td>
</tr>
<tr>
<td>( \epsilon : ) real — the required precision,</td>
</tr>
<tr>
<td><strong>definitions</strong></td>
</tr>
<tr>
<td>( x^* ) such that</td>
</tr>
<tr>
<td>( \exists a^<em>, b^</em> : ) points ( a \leq a^* &lt; b^* \leq b \land )</td>
</tr>
<tr>
<td>( f(a^<em>).f(b^</em>) \leq 0 \land</td>
</tr>
</tbody>
</table>

**Step 2. Construct the first definition.** From this statement, the idea is to define a sequence of intervals \([a[p], b[p]]\) whose size is decreasing. A point \( x[p] \) is chosen partitioning each interval \([a[p], b[p]]\). This example illustrates a very common way to derive computable functions.
statement root-2,
data
\[ x^\ast : \text{point}, \]
\[ p^\ast : \text{integer}, \]
\[ f : \text{point} \to \text{real}, \]
\[ \varepsilon : \text{real}, \]
variables \(a, b, x: \text{point},\)
definitions
\[ x \text{ such that } \forall p > 0 \quad x[p] \in [a[p], b[p]], \]
\[ a[1] \overset{\text{def}}{=} \text{initial value}, \]
\[ b(1) \overset{\text{def}}{=} \text{initial value} \]
\[ p > 0 \to a[p + 1], b[p + 1] \text{ such that} \]
\[ (a[p + 1], b[p + 1] \subseteq a[p], b[p]) \land \]
\[ (a[p + 1] = x[p] \lor b[p + 1] = x[p]) \land \]
\[ (f(a[p + 1]) \cdot f(b[p + 1]) \leq 0), \]
\[ x^\ast \overset{\text{def}}{=} x[p^\ast], \]
\[ p^\ast \text{ such that } |a[p^\ast] - b[p^\ast]| < \varepsilon. \]

Proof. Statement root-2 refines root-1. This can be easily established by proving:
(a) partial correctness, which follows from
\[ \forall p > 0 \quad (x[p] \in [a[p], b[p]], f(a[p]) \cdot f(b[p]) \leq 0), \]
(b) termination, which follows from
\[ \forall p > 0 \quad (a[p + 1], b[p + 1] \subseteq a[p], b[p]). \]

Introducing independent computations or defining vectorized computations are two common ways to develop parallel programs. Hence we can illustrate these two tactics.

3.2.2. The first approach
This approach leads to computations which are as independent as possible, by avoiding function calls. This can be accomplished by introducing intermediate variables.

Step 3. Describe the intermediate results.

statement root-3,
data
\[ x^\ast : \text{point}, \]
\[ p^\ast : \text{integer}, \]
\[ f : \text{point} \to \text{real}, \]
\[ \varepsilon : \text{real}, \]
variables
\[ a, b, x : \text{point}, \]
\[ fa, fb : \text{real}, \]
Proof. We apply the substitution rule in CBS. □

Step 4. Refine the specifications. This step refines the definition of the variable \( x \) by constructing a function which returns a variable that verifies root-3.

**statement** root-4,

**data**
\( x^* \): point,
\( p^* \): integer,
\( f \): point \( \rightarrow \) real,
\( \epsilon \): real,

**variables**
\( a, b, x \): point,
\( f_a, f_b, f_x \): real,

**definitions**
\( x = = a \) function defining some point in \( ]a, b[ \),
\( f_x = = f(x) \),
\( a[1] \) \( \equiv \) initial value,
\( b[1] \) \( \equiv \) initial value,
\( p > 0 \land f_a[p].f_x[p] \leq 0 \)
\( \rightarrow a[p + 1] \equiv a[p], \ b[p + 1] \equiv x[p], \)
\( p > 0 \land f_x[p].f_b[p] \leq 0 \)
\( \rightarrow a[p + 1] \equiv a[p], \ b[p + 1] \equiv b[p], \)
\( f_a[1] \) \( \equiv \) \( f(a[1]) \),
\( f_b[1] \) \( \equiv \) \( f(b[1]) \),
\( p > 0 \land f_a[p].f_x[p] \leq 0 \)
\( \rightarrow f_a[p + 1] \equiv f_a[p], \ f_b[p + 1] \equiv f_x[p], \)
\( p > 0 \land f_x[p].f_b[p] \leq 0 \)
\( \rightarrow f_a[p + 1] \equiv f_a[p], \ f_b[p + 1] \equiv f_b[p], \)
\( x^* \) \( \equiv \) \( f_x[p^*] \),
\( p^* \) such that \( |a[p^*] - b[p^*]| < \epsilon \).
This statement could easily be implemented as a nondeterministic sequential algorithm. To derive a parallel one, we introduce some intermediate variables, defined by trivial equations (Refinement Rule 1), and then we establish equality communication relations between them (Refinement Rule 2). Moreover, when they use the same variables, some previous definitions in root-4 are grouped to define abstract functions.

Step 5. Construct a first parallel solution.

\begin{verbatim}
statement root-5,
data
x*: point,
p*: integer,
f*: point \rightarrow real,
\epsilon: real,
variables
a, b, x, y: point,
fa, fb, s, t: real,
definitions
x = inter(a, b, fa, fb, t),
s = val(f, y),
equality
\begin{align*}
  t &= s, \\
  y &= x,
\end{align*}
\begin{itemize}
  \item \text{equality} \quad x* \overset{\text{def}}{=} x[p*],
  \item p* such that |a[p*] - b[p*]| < \epsilon,
\end{itemize}
with
\begin{itemize}
  \item \xi = inter(\alpha, \beta, \varphi, \psi, \tau) defined by
    \begin{align*}
      \xi &= a \text{ function defining some point in }]a, \beta[, \\
      \alpha[1] &\overset{\text{def}}{=} \text{initial value}, \\
      \beta[1] &\overset{\text{def}}{=} \text{initial value}, \\
      \varphi[1] &\overset{\text{def}}{=} \text{initial value}, \\
      \psi[1] &\overset{\text{def}}{=} \text{initial value}, \\
      p > 0 \land \varphi[p], \tau[p] \leq 0 &\Rightarrow \alpha[p + 1] \overset{\text{def}}{=} \alpha[p], \quad \beta[p + 1] \overset{\text{def}}{=} \xi[p], \\
      \varphi[p + 1] &\overset{\text{def}}{=} \varphi[p], \quad \psi[p + 1] \overset{\text{def}}{=} \tau[p], \\
      p > 0 \land \tau[p], \psi[p] \leq 0 &\Rightarrow \alpha[p + 1] \overset{\text{def}}{=} \xi[p], \quad \beta[p + 1] \overset{\text{def}}{=} \beta[p], \\
      \varphi[p + 1] &\overset{\text{def}}{=} \tau[p], \quad \psi[p + 1] \overset{\text{def}}{=} \psi[p], \\
    \end{align*}
  \item \sigma = val(\phi, \gamma) defined by \sigma = \phi(\gamma),
\end{itemize}
communications
\begin{align*}
  \text{equality}(i, j) &= j \in T \land i \in S \land i = j.
\end{align*}
\end{verbatim}

Proof. In root-4, the definition of x depends on fx, i.e. recursively on x itself. Then we introduce a variable t, on which x depends. On the other hand, x is used to define f(x) via a new variable s. By establishing t = s, the substitution rule proves
the correctness of this statement (Refinement Rules 1 and 2), where \( y \) is the formal argument of the function computing \( s \).

\[ \text{Fig. 9.} \]

**Step 6. Derive a first CSP program.** We can continue with this development by representing the solution in a given language. Here the target language is CSP [16]. The only difficulty is to represent the communication relations. In a general way, it is quite obvious that an unbounded FIFO data structure can implement the predicate equality. It is defined by the following CSP program (see the example of the filter \( F \) in Fig. 9):

\[
F::
\begin{align*}
[c: (1 .. \infty)] & \text{ point; } \\
[h, l: \text{ integer}] & \text{ := 0; } \\
*[\text{INTER?]c(l) \rightarrow l := l + 1} \\
& \quad [l > h; \text{ VAL!c(h)} \rightarrow h := h + 1] \\
& \quad ] \\
& ]
\end{align*}
\]

In the particular case we are concerned with, we can reduce this implementation to a simple rendez-vous. This leads to a trivial concurrent algorithm composed of two processes which run alternatively. It is expressed by the following CSP program.

\[
[\text{INTER} \parallel \text{VAL}]
\]

**INTER::**

\[
\begin{align*}
*[\text{VAL?t} & \rightarrow \\
& \quad \text{ definition of the current interval} \\
& \quad [] \text{ VAL!x} \quad \text{ a point of the current interval} \\
& \quad \rightarrow \text{ skip} \\
& \quad ]
\end{align*}
\]

**VAL::**

\[
\begin{align*}
*[\text{INTER?y} & \rightarrow s := f(y); \text{ INTER!s} \\
& \quad ]
\end{align*}
\]
Note that this program terminates in a deadlock state. This is not our main point in this example, so from now on we shall omit it.

This ends the development of a first solution.

3.2.3. The second approach

Step 7. Introduce n points in the current interval. Let us return to root-2. The idea is to duplicate point \( x \) and define a new vector of \( n \) points \( x_1, x_2, \ldots, x_n \) (Refinement Rule 4). We obtain the following statement:

\[
\text{statement root-6,}
\text{data}
\]
\[
\begin{align*}
x^* : & \text{point,} \\
p^* : & \text{integer,} \\
f : & \text{point } \rightarrow \text{real,} \\
\varepsilon : & \text{real,}
\end{align*}
\]

\[
\text{variables}
\begin{align*}
a, b : & \text{point,} \\
x : & \text{vector}(n) \text{ of points,}
\end{align*}
\]

\[
\text{definitions}
\begin{align*}
x & = \text{a function defining some vector}(n) \text{ of points in }]a, b[,} \\
a[1] & \overset{\text{def}}{=} \text{initial value,} \\
b[1] & \overset{\text{def}}{=} \text{initial value,} \\
p > 0 \land f(a[p]) \cdot f(x[p]) \leq 0 \\
& \quad \rightarrow a[p+1] \overset{\text{def}}{=} a[p], \quad b[p+1] \overset{\text{def}}{=} x[p], \\
p > 0 \land f(x[k]) \cdot f(x_{k+1}[p]) \leq 0 \\
& \quad \rightarrow a[p+1] \overset{\text{def}}{=} x_k[p], \quad b[p+1] \overset{\text{def}}{=} x_{k+1}[p], \quad k \in [1, n-1], \\
p > 0 \land f(x[n]) \cdot f(b[p]) \leq 0 \\
& \quad \rightarrow a[p+1] \overset{\text{def}}{=} x_n[p], \quad b[p+1] \overset{\text{def}}{=} b[p], \\
x^* \overset{\text{def}}{=} x_i[p^*], \\
p^* \text{ such that } |a[p^*] - b[p^*]| < \varepsilon.
\end{align*}
\]

The following steps of this development describe intermediate variables and establish communication relations (Refinement Rules 1 and 2). These steps may be carried out in several ways. We focus on two developments which differ in their intermediate variables.

Step 8. Construct a vectorized solution. The first way is to define:

- an abstract function \( \text{inter}' \), identical to the previous \( \text{inter} \), but which receives a vector of \( n \) reals and returns a vector of \( n \) points,
- \( n \) instances of the function \( \text{val} \), which return the value of \( f \) at \( n \) points of the current interval.

The result specified in root-6 is then defined by the following CBS notation, illustrated in Fig. 10.
statement root-7,
data
\[ x^* : \text{point}, \]
\[ p^* : \text{integer}, \]
\[ f : \text{point} \rightarrow \text{real}, \]
\[ \varepsilon : \text{real}, \]
variables
\[ a, b : \text{point}, \]
\[ x, y : \text{vector}(n) \text{ of points}, \]
\[ fa, fb : \text{real}, \]
\[ s, t : \text{vector}(n) \text{ of reals}, \]
definitions
\[ x = \text{inter}'(a, b, fa, fb, t), \]
\[ s_k = \text{val}(f, y_k), \quad k \in [1, n], \]
\[ t_k \overset{\text{equality}}{=} s_k, \quad k \in [1, n], \]
\[ y_k \overset{\text{equality}}{=} x_k, \quad k \in [1, n], \]
\[ x^* \overset{\text{def}}{=} x_i[p^*], \]
\[ p^* \text{ such that } |a[p^*] - b[p^*]| < \varepsilon, \]
with \ldots
communications
\[ \text{equality}(i, j) = j \in T \land i \in S \land i = j. \]

Note that the communication relations between the set of processes VAL-\(k\), \(k \in [1, n]\), and the process INTER could be defined as a single one, which would be specified by a predicate \(\tau\) over \(S_1 \times S_2 \times \cdots \times S_n \times T\).
Step 9. Derive asynchronous solutions. Another way to develop solutions from Step 7 is to snap the vectors in order to get asynchronous algorithms. Indeed, we can define a new function inter\" whose input is a sequence of successive values of f and whose output is a sequence of vectors of points. From the definition of sequence x in Step 7, we can deduce that the resulting sequence of the function \text{winter} is composed of points that belong to intervals whose size is decreasing. Hence, to compute some pth result of any kth instance of the function val, we can substitute an input datum y_k[q] for y_k[p], for any q \geq p.

Then, to link the function value with a convenient point, the pair (x, f(x)) must be given. Moreover, to ensure the property of decreasing size of the intervals [a[p], b[p]], the function inter\" must only consider the pairs (x, f(x)) such that x is inside the current interval. The functions inter\" and val\' may be defined as follows in this context:

\begin{verbatim}
data \phi : point \rightarrow \text{real},
variables \alpha, \beta, \gamma : point,
\xi : vector(n) of points,
\varphi, \psi : real,
\sigma, \tau : point \times \text{real},
— for the two required components of this last variable we denote by
— \tau 1 (respectively \tau 2) the point (respectively the real) component,

\xi = = \text{inter}"(\alpha, \beta, \varphi, \psi, \tau) defined by
\xi = = \text{a function defining some vector(n) of points in } ]\alpha, \beta[,
\alpha[1] \text{ def initial value},
\beta[1] \text{ def initial value},
\varphi[1] \text{ def initial value},
\psi[1] \text{ def initial value},

p > 0 \land (\alpha[p] < \tau 1[q] < \beta[p]) \land \varphi[p] \land \tau 2[q] \leq 0\quad\rightarrow\quad\alpha[p+1] \text{ def } \alpha[p], \quad \beta[p+1] \text{ def } \tau 1[q],
\varphi[p+1] \text{ def } \varphi[p], \quad \psi[p+1] \text{ def } \tau 2[q],

p > 0 \land (\alpha[p] < \tau 1[q] < \beta[p]) \land \tau 2[q] \leq 0\quad\rightarrow\quad\alpha[p+1] \text{ def } \tau 1[q], \quad \beta[p+1] \text{ def } \beta[p],
\varphi[p+1] \text{ def } \tau 2[q], \quad \psi[p+1] \text{ def } \psi[p],

\nu = = \text{val}'(\phi, \gamma) defined by \nu = - (\gamma, \phi(\gamma)).
\end{verbatim}

Then the result is defined by the following CBS notation, illustrated in Fig. 11.

\begin{verbatim}
statement root-8,
data
x* : point,
p* : integer,
f : point \rightarrow \text{real},
\varepsilon : \text{real},
\end{verbatim}
variables

\( a, b : \text{point}, \)
\( x, y : \text{vector}(n) \) of points,
\( fa, fb : \text{real}, \)
\( s, t : \text{vector}(n) \) of \( \text{point} \times \text{real}, \)

definitions

\( x = \text{inter}''(a, b, fa, fb, t), \)
\( s_k = \text{val}'(f, y_k), \quad k \in [1, n], \)

\( t = s, \)

\( y_k = x_k, \quad k \in [1, n], \)
\( x^* \overset{\text{def}}{=} x_i[p^*], \)
\( p^* \) such that \( |a[p^*] - b[p^*]| < \epsilon, \)

with . . .

communications

\( \text{merge}((i, k), j) \)
\( \equiv j \in T \land k \in [1, n] \land i \in S_k \land \\
(i = 1 \lor \exists j' < j \quad \text{merge}((i - 1, k), j')) \land \\
((j = 1 \land \forall k' \in [1, n] \land \forall i' \in S_{k'}) \lor \\
(\text{instant}(s_k)[i] \leqslant \text{instant}(s_k')[i']) \lor \\
(j > 1 \land \forall k', k'' \in [1, n] \land \forall i' \in S_{k'} \land \forall i'' \in S_{k''} \\
(\text{merge}((i''', k'''), j - 1) \land \text{instant}(s_k'')[i'''] < \text{instant}(s_k)[i]) \\
\Rightarrow \text{instant}(s_k)[i] \leqslant \text{instant}(s_k')[i']) \\
)\),
\[ \text{refresh}(i, j) \]
\[ = j \in Y \land i \in X \land \]
\[ (j = 1 \lor \forall i' \in X \ (\text{refresh}(i', j - 1) \Rightarrow i' < i)) \land \]
\[ \forall i' \in X \ (\text{instant}(x)[i'] \leq \text{instant}(y)[j] \Rightarrow i' \leq i). \]

**Comments.** The communication relation associated with the predicate `refresh` defines reactive systems which are suitable implementations of the condition “\( q \equiv p \)” above. The communication relation associated with the predicate `merge` defines a reactive system which merges *equality* relations for all VAL\(_k\)-INTER subsystems (see Fig. 12).

![Diagram](attachment:fig12.png)

Fig. 12.

Note that this last \((n + 1)\)-ary communication relation is specified by the following application:

\[ \varphi : \text{dom}(S) \to (\text{dom}(R_1) \cup \text{dom}(R_2) \cup \cdots \cup \text{dom}(R_n)) \times [1, n]. \]

This characterizes a tactic to *dissociate* independent calculations. Note that this tactic differs from the previous one (Step 8), based on an application from \( \text{dom}(S) \) to \( \text{dom}(R_1) \times \text{dom}(R_2) \times \cdots \times \text{dom}(R_n) \).

**Step 10. Obtain a second CSP program.** As a follow-up to this reasoning process, we introduce an implementation of each of these communication relations by the manipulation of a data structure. This is an example of applying the well-known paradigm: represent objects and relations in some target model.

**Representation of the communication relation “refresh”.** It is quite trivial to deduce from the second part of the predicate:

\[ \forall i' \in X \ (\text{instant}(x)[i'] \leq \text{instant}(y)[j] \Rightarrow i' \leq i) \]

that, for any \( k \) in \([1, n]\), the needed data structure is composed of a variable of type point and two counters for indices in the sequences. Indeed, every computation of an \( x(k) \) value increases the value of a counter \( \#p \), and the index \( \#c \) of any considered value \( y_k \) is the current value of \( \#p \). In fact, because of the first part of the predicate

\[ j = 1 \lor \forall i' \in X \ (\text{refresh}(i', j - 1) \Rightarrow i' < i) \]
the boolean value "\#c < \#p" may be substituted for these counters. The following CSP process defines the variable x, of type point, and the variable rs, of type boolean.

\[
F(k) :: \\
[z : \text{point}; \]
\[
rs : \text{boolean} := \text{false}; \]
\[
*[\text{INTER} ? z \rightarrow rs := \text{true} \]
\[
[]rs; VAL(k)! z \rightarrow rs := \text{false} \]
\[
].
\]

**Representation of the communication relation “merge”**. We can easily generalize the implementation of the predicate equality to deduce an unbounded FIFO data structure which implements the relation merge. It is defined by the following CSP program:

\[
F' :: \\
[c: (1..\infty) \text{ point} \times \text{real}; \]
\[
h, l: \text{integer} := 0; \]
\[
*[((k:1..n) \text{VAL}(k)?c(l) \rightarrow l := l + 1 \]
\[
][l > h; \text{INTEGER!}c(h) \rightarrow h := h + 1 \]
\[
].
\]

Then, our reasoning process continues with the following step:

\[
\text{[INTER } F' \parallel F(k:1..n) \parallel \text{VAL}(k:1..n)]}, \]

\[
\text{INTER ::} \]
\[
\ldots \]
\[
((k:1..n) F(k)!x(k); \]
\[
*[F'?t \rightarrow \text{—definition of the current interval [a, b]} \]
\[
\text{—and of the vector x} \]
\[
(k:1..n) F(k)!x(k) \]
\[
]. \]

\[
\text{VAL}(k) :: \]
\[
\ldots \]
\[
*[F(k)?y \rightarrow s := (y,f(y)); \]
\[
F'!s \]
\[
].
\]

The processes \(F'\) and \(F(k)\) are defined above. Note that we omit the termination aspect, as was mentioned.

**3.2.4. Transformation techniques**

**Step 11. Cancel process \(F'\)**. Our goal is now to simplify this program by applying rational transformations. These transformations consist of abstracting communication
processes, i.e. processes that express some communication relation. The first transformation substitutes a simple rendez-vous between the processes VAL\( (k) \) and INTER for the communication expressed as an unbounded FIFO medium, without introducing deadlocks. We obtain the following program:

\[
[\text{INTER} \parallel F(k:1..n) \parallel \text{VAL}(k:1..n)],
\]

\[
\text{INTER} ::
\]
\[
\ldots
\]
\[
[(k:1..n) F(k)!x(k);
*[(k:1..n) \text{VAL}(k)?t \to
\ldots \text{id} \ldots
\]
\]
\[
\text{VAL}(k) ::
\]
\[
\ldots
\]
\[
*\left[(F(k)?y \rightarrow s := (y,f(y));\right.
\]
\[
\text{INTER}!x
\]
\]

The processes \( F(k) \) are defined above.

**Step 12. Merge processes INTER and \( F(k) \).** Now we proceed to a transformation which preserves syntactical correctness [24] by merging processes INTER and \( F(k) \). We identify the local variables \( x(k) \) in INTER and the local variable \( z \) in each \( F(k) \). Then, the communication operation \( "F(k)!x(k)" \) is reduced to its consequence, i.e., to the assignation \( "rs := true" \), where \( rs \) is the local variable of \( F(k) \). These \( n \) boolean variables are then represented by a vector of boolean variables \((rs(1), rs(2), \ldots, rs(n))\). Then the first communication operation in INTER: \( "(k:1..n) F(k)!x(k)" \) can be reduced to the initialization statement: \( "rs(k) := true for any \( k \) in \([1, n]\)". Lastly, this communication operation in the iterative part is expressed as a nondeterministic alternative. We obtain the following program:

\[
[\text{INTER} \parallel \text{VAL}(k:1..n)],
\]

\[
\text{INTER} ::
\]
\[
[rs:(1..n) \text{ boolean} := true;
\ldots
*[(k:1..n) \text{VAL}(k)?t \to
\ldots (k:1..n) rs(k) := true
\]
\[
\text{VAL}(k)!x(k) \to rs(k) := false
\]
\]

\begin{verbatim}
VAL(k) :: 
  ... 
  ![INTER ?y \rightarrow s := (y, f(y)); 
    INTER!s 
  ].
\end{verbatim}

In this program, the definition of the vector \((rs(1), rs(2), \ldots, rs(n))\) of boolean variables is a way to implement the required communication relation (such an expression is introduced in an example in [1]). We argue that this definition is the critical point in the design of such an algorithm. These booleans are parasitic variables with regard to the initial problem. Indeed, this algorithm is fundamentally a set of producer-consumer systems:

- the process INTER computes a sequence of vectors \((x(1), x(2), \ldots, x(n))\) of points, a few of which are utilized by the processes \(VAL_1, VAL_2, \ldots, VAL_n\),
- conversely, each process \(VAL_k\) computes a sequence of pairs \((y, f(y))\) which are utilized by INTER.

It is interesting to note that another asynchronous solution is given by Eriksen and Staunstrup [10]. The main difference from the one presented here is the absence of boolean conditions. Such a solution may be derived from a variant of the communication relation "refresh", in which only the second part of the predicate is defined. This relation means that the condition \("q \geq p"\) is not satisfied. The actual condition is then a weakening of the previous one, and defines a sequence of intervals whose size is not increasing. The efficiency and the convergence of the solution may depend on this hypothesis.

4. Conclusion

We are concerned with the important gap between proposed programming languages and parallel programming methods. Meanwhile, the design of parallel programs implies extensive work. We have proposed the beginning of an answer to this question by presenting a few paradigms and tactics for parallel programming. These paradigms are founded on the concept of communication relations, which are an abstraction of the cooperation between communicating processes. The tactics are based on refinement rules applied to abstract statements. This point has been illustrated by developing the above two examples. In these examples, the communication relations have been quite obvious; this is not always the case. Elsewhere we have shown [29] many communication relations whose definitions require long and careful development. In a few cases, this definition is the essential point of constructing a program. In other cases, it may lead to different versions of a given algorithm.

The steps described above has been chosen to illustrate the paradigms and tactics for parallel program design. Various groupings of such paradigms may lead to the
development of more or less interesting solutions. They can also display checks or blind alleys in these designs. It seems imperative to propose such reasoning tools for writing correct and/or efficient parallel programs.

References


