

Event Driven Distributed Program Construction

by J.-R. Abrial

August 2001

Version 5

Event Driven Distributed Program Construction

In this article, we shall illustrate the use of a certain *formal technique* for developing *distributed algorithms*. By “formal technique” we mean one consisting in: (1) describing rigorously the problem at hand, (2) elaborating gradually a solution to it, and (3) mechanically verifying that the proposed solution is correct. This technique uses a, so-called, “event driven” approach ([2] [3]) together with the B-Method as it is introduced in [1]. This technique is already mentioned and used elsewhere for the development of sequential programs [4], electronic circuits [5], and, more generally, complex systems involving software as well as hardware [6]. Since the theoretical background of this approach has already been amply presented in the mentioned works, we shall not discuss it here another time. We shall just present a number of illustrating examples.

All cases presented in this paper have been entirely mechanically treated with Atelier-B, the tool associated with the B-Method. For each of them, we shall give the number of proofs that have been discharged automatically or interactively by the prover of the tool. It should be reminded to the reader that in using Atelier-B, you do not have to write yourself what is to be proved: this is produced automatically by the tool (more precisely, by that part of it called the “proof obligation generator”). Such statements are generated according to the various mathematical theories that have been presented in [1]. In these examples, we shall thus follow the classical way of using that tool: (1) informally stating and solving a given problem, then (2) encoding it in B, and finally (3) submitting it to the tool. In what follows we shall thus hardly present any proof at all (we sometimes give some hints however) unless we feel that such proofs are needed in order to improve the informal understanding of our solution.

Example 1: A Simple Transmission Protocol

Our first example has already been presented in more general and complex terms in [7]. We only present here a simplified version, whose rôle is to give a first elementary illustration of our approach. This example is very classical: it is called the Stenning’s protocol [8]. This is an abstraction of the well known “alternating bit” protocol [9].

This protocol is supposed to transfer a file from one agent, the sender, to another one, the receiver. These agents are supposed to reside on *different sites*, so that the transfer is not made by a simple copy of the file, it is rather realized gradually by two distinct programs exchanging various kind of messages on a network. Such programs are working with different contexts and on different machines: the overall protocol is indeed *distributed program*.

What we are going to develop here is *not* directly the distributed program in question. We are rather going to construct a *model of its distributed execution*. In the context of this *model*, the file to transfer is formalized by means of a constant total function f from the interval $1..n$ to some set D . Formally:

$$\begin{array}{l} n \in \mathbb{N} \\ f \in 1..n \rightarrow D \end{array}$$

The file f is supposed to “reside” at the sender’s site. At the end of this protocol execution, we want the file f to be copied without loss nor duplication on the receiver’s site. On this site, the copied file, called g , is typed as follows:

$$g \in 1 \dots n \rightarrow D$$

The very global transfer action of the protocol can be abstracted by means of a *single* event called *trm* as follows:

$$\text{trm} \hat{=} \text{BEGIN } g := f \text{ END}$$

This event does not “exist” by itself, it is not part of the protocol: this is just a *time snapshot* that we would like hopefully to *observe*. In the “reality”, the transfer of the file *f* is not done in one shot, it is made gradually. But, at this very initial stage of our approach, we are not interested in this. In other words, *as an abstraction*, and regardless of what will happen in the details of the distributed execution of the protocol, its final action must result in the possibility to observe that the file *f* has indeed been copied in the file *g*.

It should be noted that, at this point, we are not committed with any particular protocol: this specification is thus, in a sense, the most general one corresponding to a given class, namely that of succeeding transfers. Some more elaborated specifications could have been proposed, in which the file might have only been partially transferred.

We are now going to *refine* the file transfer done in one shot by the previous *abstract* event *trm* acting “magically” on the receiver’s side. For this, we have a number of *concrete* events corresponding to the various *phases* of the protocol. They are aiming at transferring the file *piece by piece*. Of course, the abstract event *trm* should not disappear: it will have a concrete counterpart in which the same observation as in the abstraction would be possible.

These phases are informally behaving as follows: the sender has a local counter *s*, which records the “index” of the next datum to be send to the receiver (initially, *s* is set to 1). When a transmission does occur, the value of the index *s* is sent together with that of the datum *f(s)* (event *sndd*). The sender does not increment *s* nor does it send the next item immediately, it waits until it receives an *acknowledgement*. The receiver is also supposed to have its own local counter *r* initially set to 0. When receiving a pair “index-datum”, it compares the received index with *r* and accepts the datum if the index it receives is just equal to *r* + 1 (event *rcvd*). In this case, *r* is incremented and sent back as an acknowledgement (event *snda*). When the sender receives a number which is equal to its own counter *s* (event *rcva*), it consider that the acknowledgement is effective, increments *s* and proceeds with the next item, and so on.

The sender and the receiver are thus connected by means of two channels as indicated in Figure 1: the data channel and the acknowledgement channel. These channels are supposed to be *non-reliable*. More precisely, items in them can be reordered, duplicated, or even removed. In our model, the first default (reordering) is formalised by supposing that each channel contains a set of unordered items rather than some sort of FIFO list. The second one (duplication) is rendered by supposing that an item, once read, is not removed from the channel (hence it can be read again, thus simulating a duplication). Finally, the third default (removal) is realized by means of some “daemons” acting on the channels by removing some randomly choosen items. Clearly these daemons are not part of the protocol itself, they rather represent the possibly *hostile context* within which the protocol is supposed to behave.

In order to cope with the possible loss of messages in both channels, the sender keeps sending the same pair “index-datum” while waiting for a correct acknowledgement from the receiver. Similarly, the receiver, once it has received a correct pair from the sender, keeps on sending the same acknowledgement until it receives

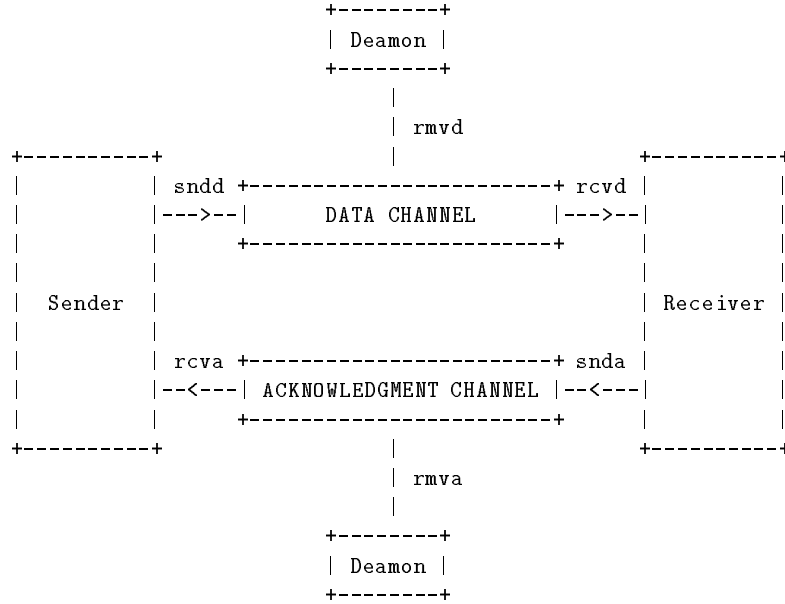


Fig. 1. The Transmission Protocol Environment

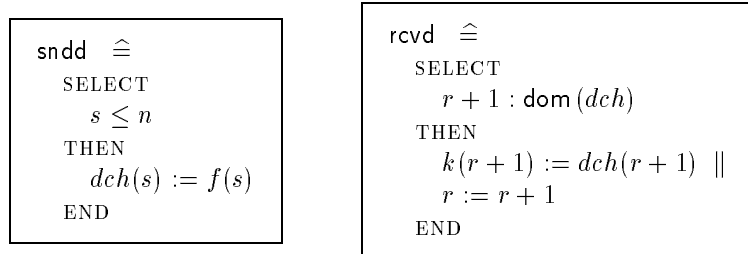
the next correct pair. We now proceed with the formal treatment of this refinement. The state variables are declared as follows (notice that the concrete result file is now called k):

$$\begin{array}{l}
 s \in 1 \dots n + 1 \\
 r \in 0 \dots n \\
 r \leq s \leq r + 1 \\
 k = (1 \dots r) \triangleleft f
 \end{array}$$

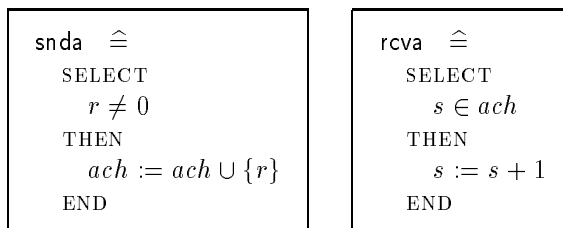
The first two invariants corresponds to the declarations of s and r . The third invariant states that the counter s is at most one more than the counter r . The fourth invariant states that the result file, which we call k in this refinement, corresponds exactly to the r first elements of the original file f . It remains now for us to formalize the channels. The data channel dch is, as we know, a set of pair “index-datum”. In fact this set of pairs happen to be a partial function from the interval $1 \dots n$ to D . More precisely, this function is included into f restricted to its s first elements. The acknowledgement channel is formalized by means of a set of indices included in the interval $1 \dots r$. Formally:

$$\begin{array}{l}
 dch \subseteq (1 \dots s) \triangleleft f \\
 ach \subseteq 1 \dots r
 \end{array}$$

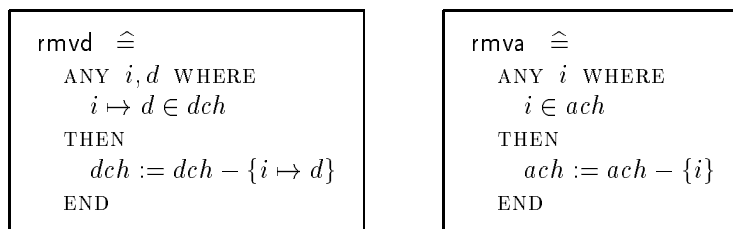
Next are the various events. They encode the informal behaviour of the protocol as described above. First, on the “data” events:



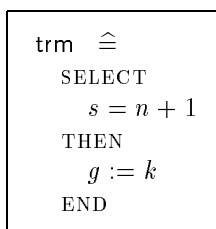
Then come the “acknowledgement” events.



Finally, we have the two deamons:



It remains now for us to re-consider our unique abstract event `trm`. Termination of the protocol is clearly achieved when s is equal to $n + 1$, that is when the last acknowledgement has been received by the sender. In that case, the concrete result file k should be equal to our initial file f (this is what we can reasonably suppose). Thus the concrete event `trm` is guarded by the condition $s = n + 1$ and its “action” part consists in copying in g , not the file f as in the abstraction, but rather the concrete file k that has been gradually transported on the receiver’s site. Formally:



It might seem strange to still have to make a *copy* in this concrete event. It should be noted however that this event is *not part of the protocol*: it is just an artefact used to make precise what the protocol does. Technically, we could have avoided this copy by having the concrete events working directly on the file g and having then the concrete version of the event `trm` doing nothing (but still being guarded by the condition $s = n + 1$). The price to pay however would have been to write an abstract version of the event `rcva` doing a completely non-deterministic and meaningless assignment in the file g . In the model which we have presented here, the event `rcva` has no abstract counterpart like the other events dealing directly with the protocol. Using one or

the other technique is rather a question of taste.

It is easy to prove that this event `trm` refines its abstraction. This amounts to proving $s = n + 1 \Rightarrow k = f$ (Hint: first prove that $t = n$ holds and then use the invariant $k = (1..r) \triangleleft f$).

Notice that the termination of the protocol is not guaranteed by this formal model. In other words, we are not sure to ever reach the conditions for the firing of the concrete event `trm`. This is so because we have not made any assumptions concerning any maximum number of duplications or removals in the channels. Easy finitary assumptions about these duplications and removals could have been made allowing us to prove that some natural number quantity is indeed decreasing whatever the event that is fired. In [7] a similar protocol is presented in which the retransmissions of data or acknowledgements are *bounded* so that the protocol can be proved to terminate, but sometimes, of course, with a file only partially transmitted. Its specification is thus slightly different from the one we have given here.

A last point is worth noticing. If the protocol terminates (that is, if the condition $s = n + 1$ holds), we would like to have all events directly concerned by the protocol, namely `snd`, `rcvd`, `rcva`, and `snda`, to have naturally “deadlocked”. Clearly the guards of the three first one do not hold any more. But unfortunately, the guard $r \neq 0$ of `snda` continues to hold. In other words, the receiver continues to indefinitely send its last acknowledgement to the sender. This is due to the fact that the receiver has no way to know that the sender has indeed received this last acknowledgement since there is no further item to be sent by this agent. Clearly, this is a weakness of this protocol. We leave it to the reader to improve this interesting situation.

The formal correctness of this model (invariant preservation and correct refinement of the event `trm`) requires 23 proofs all done automatically.

Example 2: Leader Election on a Ring-shaped Network

This second example originated in a paper by Le Lann in the seventies [10]. Here we have a possibly large (but finite) number of agents, just not two as in the previous example. These agents are disposed on different sites that are in communication with each others by means of unidirectional channels forming a ring as indicated in the following figure.

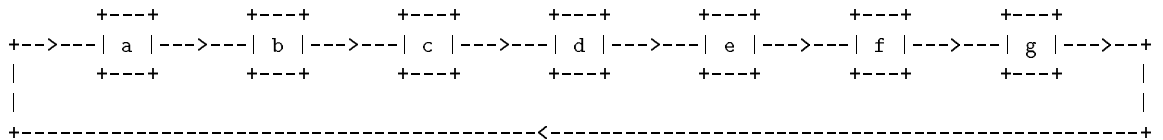


Fig. 2. A ring-shaped network

Each agent is thus able to send messages to its “right” neighbour and receive ones from its “left” neighbour. Such messages are not supposed to be transmitted immediately from one node to the next. In fact, we suppose that they can be “buffered” between the two, and also reordered or duplicated. Moreover each agent is supposed to execute the *same* piece of code. The distributed execution of all these programs should result in a *unique agent* being “elected the leader”. This decision, based on certain local criteria, should be made by the winning agent itself. Of course, it must be proved that no other agent can reach the same conclusion. The determination of such a privileged agent might be useful when the ring is started or re-initiated.

Since every agent executes the same code, the problem seems to be unsolvable: what kind of distinction between them could indeed introduce a certain difference in their, otherwise homogeneous, behaviour? Their position in the ring is certainly not such a distinction, since the very shape of the ring does not give the position of an agent any special distinction (no first, no last, no medium position, etc). In fact, the only attribute that makes one agent different from the others is its name: the agents are indeed supposedly named and named differently. But by itself, this difference in names still is an homogeneous property: there is, a priori, no “more” distinction than the distinction itself.

In order to possibly introduce a supplementary distinction in these distinct names, we must have a certain structure in the name set. The “simplest” one we can think of is that of the natural numbers. In other words, we shall suppose that the names of the agents form a finite set of natural numbers. Clearly then, there exists a possible identifying distinction between these names: the largest one (or the smallest one as well). Now the problem can be restated as follows. How can one agent figure out that it bears a name that happen to be the largest number of the collection of names of all agents in the ring? At this point, we have enough elements to start our formalization. We first define the constant set of agent names A and its maximum mx :

$$\begin{array}{l} A \subseteq \mathbb{N} \\ mx \in A \\ \forall n \cdot (n \in A \Rightarrow n \leq mx) \end{array}$$

As in the previous example, we define a single event, `elect`, which solves the problem in one shot by assigning the maximum mx of A to a variable of the model called w (for winner), formally:

$$\text{elect} \hat{=} \text{ BEGIN } w := mx \text{ END}$$

Here is a simple concrete procedure. To begin with, each agent has no choice but to send its own name to its right neighbour. An agent receiving a name from its left neighbour collects it and sends it further to its right neighbour. When an agent receives its own name, this “obviously” means that it has collected all the names (since its own name must have made a complete turn of the ring) and can then decide whether its name is indeed the maximum mx . Clearly, this very primitive procedure works but it is not very efficient. Moreover, it supposes that the channels do not reorder the messages. This second drawback can be circumvented by having each agent knowing the number n of distinct agents so that it can start the decision procedure, not, as before, when receiving its own name from its left neighbour, but only after receiving $n - 1$ distinct agent names that are distinct from its own. That would certainly also work (although in a very tedious way) but we want to avoid agents having to know the number n in question for obvious practical reasons (the ring could be quite often dynamically extended or shrunked).

The other procedure that has been proposed resembles the first one we mentioned above. But rather than transmitting systematically the just received name, the idea is only to transmit it, provided it is strictly greater than the current agent name. In case it is strictly smaller than the current name, then the received name cannot be the maximum we are looking for, consequently there is no point in retransmitting it since in no case could it be elected. Finally, in case the received name is the same as that of the current agent, then the agent in question is elected: its name is indeed the maximum mx . We have the “impression” that it works but *it certainly remains to be proved* in full generality (and in the presence of asynchronous channels, which may reorder or duplicate messages).

The first model we propose now is *not* one where we represent explicitly the channels between the nodes nor the corresponding “read” and “write” operations. We shall rather represent the actual state of the evolving

situation by means of a binary relation linking each agent name x with the names of the agents, which already have had x in their buffer of messages either in the past or just now. For instance if the name \mathbf{a} has moved successively to \mathbf{b} and \mathbf{c} , and is now waiting to be read by \mathbf{d} , then \mathbf{a} is related to \mathbf{b} , \mathbf{c} , and \mathbf{d} .

Now, clearly, \mathbf{a} must be already greater than \mathbf{b} and \mathbf{c} since otherwise it would not have been allowed to move until \mathbf{d} . Therefore, if we suppose that, moreover, \mathbf{a} is also greater than or equal to all names x situated in the outer “interval” from \mathbf{d} to \mathbf{a} in the ring (interval including \mathbf{d} and \mathbf{a}), then, clearly, \mathbf{a} is equal to the maximum mx . Now, when \mathbf{a} is related to itself, a situation that it discovers when it reads its own name, then the outer interval mentioned above reduces to the singleton $\{\mathbf{a}\}$, and so \mathbf{a} is indeed the maximum mx .

We notice that such a simple informal proof has been possible because we departed in our model from a close copy of the real environment of the futur distributed program: no FIFO channels, no buffers, etc. It is important to make a clear distinction between the activity of building models and that of building programs. In the former, we are aiming at proving. In the latter, we are aiming at executing. Such a difference in goals induces a difference in forms.

What we would like to do now is to completely mechanize this informal proof and then proceed further to have a final model corresponding to the real environment of the distributed program. The first thing we formalize is the concept of a *ring* and that of an *interval* on a ring. The ring is first defined by means of a function nx (for next) representing the connection between each agent and its, say, right neighbour. Clearly nx is a total function from the set A to itself. We also define a function itv (for interval), which, given two nodes, yields their interval as a set of nodes. Here is the typing of these constants:

$$\begin{array}{l} nx \in A \rightarrow A \\ itv \in A \times A \rightarrow \mathbb{P}(A) \end{array}$$

Next are more properties of the function itv concerning the interval from a node to itself and also the relationship between itv and nx :

$$\begin{array}{l} \forall x \cdot (x \in A \Rightarrow itv(x, x) = \{x\}) \\ \forall (x, y) \cdot (x, y \in A \times A \wedge x \neq y \Rightarrow itv(x, y) = \{x\} \cup itv(nx(x), y)) \end{array}$$

The last property we introduce is the one defining the ring characteristic property: every point x in the ring is such that the interval between $nx(x)$ and x is exactly the entire set A .

$$\forall x \cdot (x \in A \Rightarrow itv(nx(x), x) = A)$$

We now define the variable ps (for position) corresponding to the binary relation informally mentioned above. Initially, ps is set to nx . We type ps and also give its *main invariant property*, as informally stated above.

$$\begin{array}{l}
ps \in A \leftrightarrow A \\
mx \in \text{dom}(ps) \\
\forall (x, y) \cdot \left(\begin{array}{l} (x, y) \in ps \wedge \\ \forall z \cdot (z \in \text{itv}(y, x) \Rightarrow z \leq x) \\ \Rightarrow \\ x = mx \end{array} \right)
\end{array}$$

Next are the events: **elect** that was already present in the abstraction and the new event **progress**.

$$\begin{array}{l}
\text{progress} \hat{=} \\
\text{ANY } a, b \text{ WHERE} \\
(a, b) \in ps \wedge \\
b < a \\
\text{THEN} \\
ps := ps \cup \{a \mapsto nx(b)\} \\
\text{END}
\end{array}$$

$$\begin{array}{l}
\text{elect} \hat{=} \\
\text{ANY } a, b \text{ WHERE} \\
(a, b) \in ps \wedge \\
b = a \\
\text{THEN} \\
w := a \\
\text{END}
\end{array}$$

The invariant preservation and also the verification that the concrete event **elect** indeed refines its abstraction require 7 proofs among which 4 were easily performed interactively.

We now proceed with another refinement transforming the relation ps by associating an unordered “buffer” bf with each node in the ring. The variable ps will disappear, it will be “glued” by a certain predicate to the new function bf . This refinement consists then in “localizing” the global relation ps . Here are the formalities:

$$\begin{array}{l}
bf \in A \rightarrow \mathbb{P}(A) \\
\forall (x, y) \cdot (y \in A \wedge x \in bf(y) \Rightarrow (x, y) \in ps)
\end{array}$$

The events are modified as follows in a straightforward way:

$$\begin{array}{l}
\text{progress} \hat{=} \\
\text{ANY } a, b \text{ WHERE} \\
b \in A \wedge \\
a \in bf(b) \wedge \\
b < a \\
\text{THEN} \\
bf(nx(b)) := bf(nx(b)) \cup \{a\} \\
\text{END}
\end{array}$$

$$\begin{array}{l}
\text{elect} \hat{=} \\
\text{ANY } a, b \text{ WHERE} \\
b \in A \wedge \\
a \in bf(b) \wedge \\
b = a \\
\text{THEN} \\
w := a \\
\text{END}
\end{array}$$

A simple automatic proof is needed to validate the following transformations:

```

progress ≐
  ANY b WHERE
    b ∈ A
  THEN
    ANY a WHERE
      a ∈ bf(b) ∧
      b < a
    THEN
      IF a = b THEN
        w := a
      ELSIF b < a THEN
        bf(nx(b)) := bf(nx(b)) ∪ {a}
      END
    END
  END
END

```

```

elect ≐
  ANY b WHERE
    b ∈ A
  THEN
    ANY a WHERE
      a ∈ bf(b) ∧
      b = a
    THEN
      IF a = b THEN
        w := a
      ELSIF b < a THEN
        bf(nx(b)) := bf(nx(b)) ∪ {a}
      END
    END
  END
END

```

By putting now the two events together, we obtain the following “piece of code” to be performed in each node b . As can be seen the second ANY corresponds to the reading of the buffer $bf(b)$, whereas the assignment $bf(nx(b)) := bf(nx(b)) \cup \{a\}$ corresponds to writing a in the buffer $bf(nx(b))$.

```

elect ≐
  ANY b WHERE
    b ∈ A
  THEN
    ANY a WHERE
      a ∈ bf(b)
    THEN
      IF a = b THEN
        w := a
      ELSIF b < a THEN
        bf(nx(b)) := bf(nx(b)) ∪ {a}
      END
    END
  END
END

```

Notice that, as in the previous example, the termination is not guaranteed. It could, provided certain hypotheses hold concerning the limitation of duplications in the buffers. The formal development of this example required 19 proofs, among which 8 needed interactions.

Example 3: Synchronizing Processes Spread on a Tree-shaped Network

In this example we have a network, which is slightly more elaborate than in the previous case: it is a tree. At each node of the tree, we have a “process” performing a certain task, which is the same for all processes (the exact nature of this task is not important). The constraint we want this processes to observe is that they remain *synchronized*. In other word, no one of them should be able to “progress” too much with regards to the others. This example has been treated by many researchers. We have taken it from the following books [11], [13] but, clearly this is not the original presentation of it.

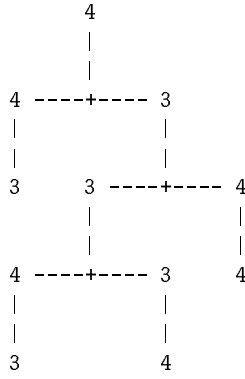


Fig. 3. Synchronized processes (the difference between any two counters is at most 1)

In order to formalize this synchronization constraint, we assign a counter to each node of the tree and we state that the difference between any two of these counters is at most equal to 1. Intuitively, each counter represents the “phase” within which each process is currently running. The constraint thus expresses that each process is at most on phase ahead of the others.

An additional constraint of our distributed algorithm states that each process can *read the counters of its immediate neighbours only*. Of course, each process is *only allowed to modify its own counter*. One of the most important aspect of our approach is that such localization constraints have not to be followed right from the beginning of the construction process. In the early phases of the process, we feel free to “magically” have access everywhere from any node. During the refining phases however, we shall gradually strengthen the guards of the events so that these localization constraints will be observed eventually. Again, this reveals an important distinction between a model and a program.

We now proceed with the first formalization. The network is defined from an abstract set N of nodes. The tree is defined by its root r which is a node, and also by the “father” function f defined on all nodes except r . Formally:

$$\begin{array}{l} r \in N \\ f \in N - \{r\} \rightarrow N \end{array}$$

Of course these two components alone are not sufficient to define a tree (for instance, a tree together with a ring can be defined in this way). But as we do not need them, we shall not introduce more properties for the moment. The only state variable is the node function c defining the counters (all initialized to 0). We type it and also formally state the basic synchronization property:

$$\begin{array}{l} c \in N \rightarrow \mathbb{N} \\ \forall (x, y) \cdot (x \in N \wedge y \in N \Rightarrow c(x) \leq c(y) + 1) \end{array}$$

Our only event, **progress**, makes explicit the conditions under which a node n can progress, this is obviously when its counter is not greater than any other counter: in this case only can it increment its counter without destroying the invariant. As can be seen (and as announced above), we have supposed that a given node n has

free access to all other nodes in the tree: again, this is because we are here in an abstraction where every access is possible.

```

progress  $\hat{=}$ 
  ANY  $n$  WHERE
     $n \in N \wedge$ 
     $\forall m \cdot (m \in N \Rightarrow c(n) \leq c(m))$ 
  THEN
     $c(n) := c(n) + 1$ 
  END

```

The problem with the guard of this event is that we have to compare the value $c(n)$ of the counter at node n , with that of *all* other counters $c(m)$. In order to limit such comparisons, the idea is to suppose that the value $c(r)$ of the counter at the root is always smaller than or equal to that of any other counters. We have thus $c(r) \leq c(m)$ for *all* nodes m . In that case, it is sufficient to reduce the guard of event **progress** to a mere comparison of $c(n)$ with $c(r)$: when the equality holds then we have $c(n) \leq c(m)$ for *all* nodes m and we can thus safely increment the counter of n .

In order to maintain the property $c(r) \leq c(m)$ for *all* nodes m , it is “obviously” sufficient (a property that we shall have to formally prove however) to have the counter of the “father” of each node m (except the root which has no “father”) being kept smaller than or equal to that of its “son”: $c(f(m)) \leq c(m)$ (the intuitive idea is that this property moves up the tree until the root r is reached). We thus introduce the following new invariant:

$$\forall m \cdot (m \in N - \{r\} \Rightarrow c(f(m)) \leq c(m))$$

This has the consequence that the counters are incremented by *waves* moving up from the leaves as shown in the following figure:

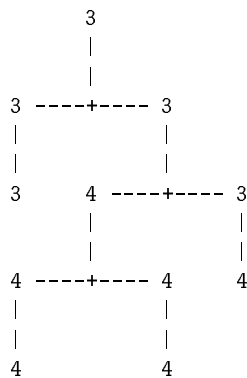


Fig. 4. A wave of 4 is climbing up the tree

In order to maintain our new invariant, we have to strengthen the guard of the event by ensuring that the counter of node n is strictly smaller than that of all its “sons”:

```

progress ≐
  ANY n WHERE
    n ∈ N ∧
    c(n) = c(r) ∧
    ∀m · (m ∈ N ∧ f(m) = n ⇒ c(n) < c(m))
  THEN
    c(n) := c(n) + 1
  END

```

Figure 5 below shows some typical transitions where the evolution of the moving wave can be seen. Notice

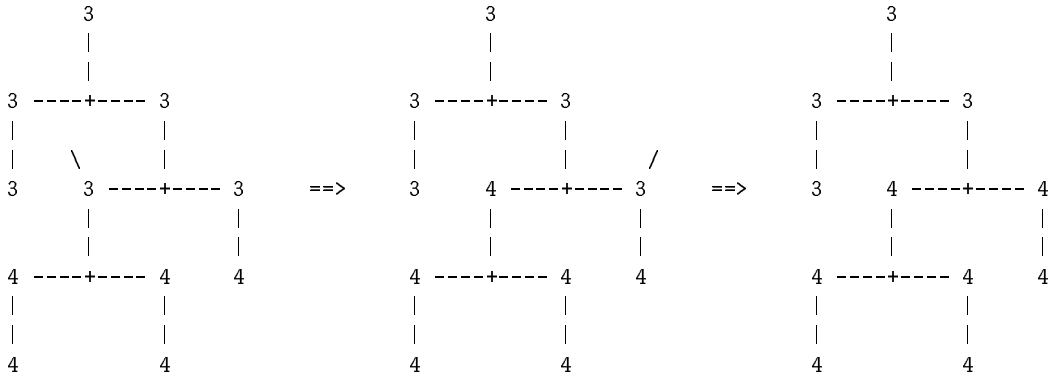


Fig. 5. Some transitions in the phase synchronization

that the comparison in node n are now limited to the sons of n (which is allowed), but still also with the root, which is not yet acceptable and will be solved in the next refinement.

We have said above that the new invariant concerning the non-augmentation of the counter from a node to its “father” was sufficient to prove the following property stating that the counter at the root is not greater than that of any other node:

$$\forall x \cdot (x \in N \Rightarrow c(r) \leq c(x))$$

As we said, this is clearly “obvious” by propagating our invariant property from the leaves up to the root. But, so far, *we have no way to prove it formally* unless we state a general *induction rule* which is the essence of the tree structure. This rule states that in order to prove a property P of all nodes of a tree it is sufficient to prove that it holds at the root and then at any node under the hypothesis that it holds at its “father”. This is typically a, so-called, “higher-order” rule. Within the framework of set theory, where we place ourselves in B , such an higher-order rule is very easily formalized by replacing the property P by a set S supposed to be included in the set N of nodes of the tree. The fact that a node n has the property P is simply formalized by saying that n is a member of S , as shown in the following formal rule:

$$\forall S \cdot \left(\begin{array}{l} S \subseteq N \wedge \\ r \in S \wedge \\ \forall n \cdot (n \in N - \{r\} \wedge f(n) \in S \Rightarrow n \in S) \\ \Rightarrow \\ N \subseteq S \end{array} \right)$$

Now, in order to prove our intended property, namely

$$\forall x \cdot (x \in N \Rightarrow c(r) \leq c(x))$$

that is, equivalently

$$N \subseteq \{x \mid x \in N \wedge c(r) \leq c(x)\}$$

we simply instantiate S in the above rule with the set

$$\{x \mid x \in N \wedge c(r) \leq c(x)\}$$

It then amounts to proving that the root r belongs to it, which is trivial, and then to prove the following (after some simplifications):

$$c(r) \leq c(f(n)) \Rightarrow c(r) \leq c(n)$$

But the following holds, according to our invariant

$$c(f(n)) \leq c(n)$$

This eventually proves our result by transitivity. It remains now to replace the test $c(r) = c(n)$ in the guard of the event **progress** by a more local test. The problem is to have the nodes made informed that the value of the counter at the root is indeed equal to their local counter. This is clearly the case when an incrementing wave has reached the root, where then all nodes have the same value, as shown in figure 6. The idea is to have then

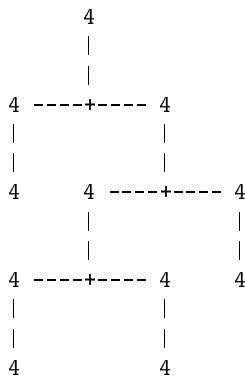


Fig. 6. All nodes are now perfectly synchronized

a second wave going down and gradually informing the nodes. For this, we need a second counter d at each node for holding that second descending wave as indicated in Figure 7. We thus declare the counter d together with its two properties: one is stating that in all nodes n (except the root) $d(n)$ is not greater than $d(f(n))$, and the other states that the counter d at the root is not greater than that of c . Formally:

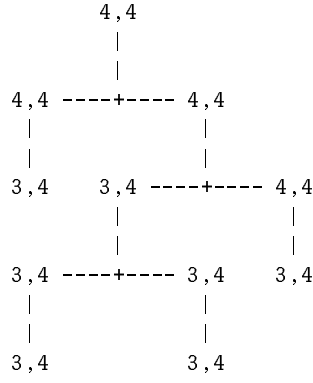


Fig. 7. A descending wave of 4 in the counters d

$$\begin{aligned}
 & d \in N \rightarrow \mathbb{N} \\
 & \forall n \cdot (n \in N - \{r\} \Rightarrow d(n) \leq d(f(n))) \\
 & d(r) \leq c(r)
 \end{aligned}$$

From our second invariant, we can easily deduce the following by induction

$$\forall n \cdot (n \in N \Rightarrow d(n) \leq d(r))$$

From this we immediately deduce that, provided both counters $c(n)$ and $d(n)$ are equal in a node n then the condition $c(n) \leq d(r)$ does hold, then also $c(n) \leq c(r)$ by transitivity. As we already have $c(r) \leq c(n)$ (this was the property we proved by induction in the previous refinement), we easily deduce that $c(r) = c(n)$ holds. In other words, we have been able to transform the remaining non-local condition $c(r) = c(n)$ of the guard of

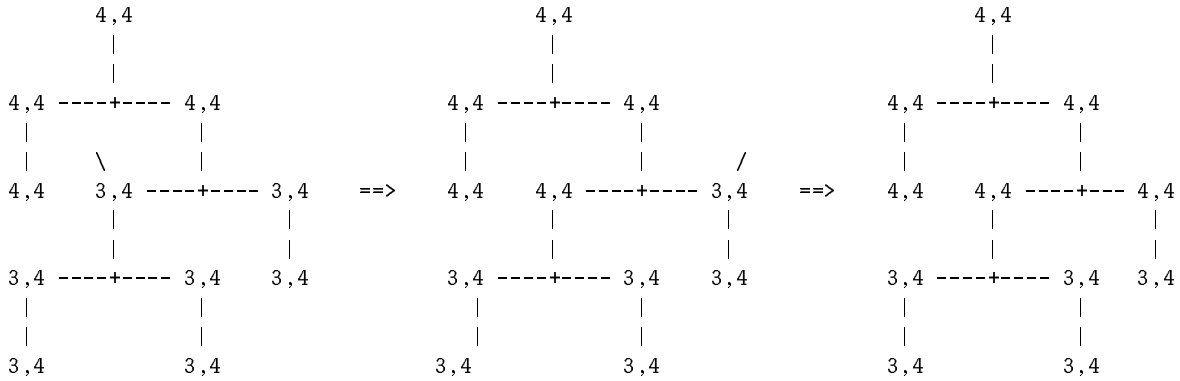


Fig. 8. Some transitions in the descending wave

event **progress** into the local condition $c(n) = d(n)$. Here is thus the new version of that event:

```

progress  $\hat{=}$ 
  ANY  $n$  WHERE
     $n \in N \wedge$ 
     $c(n) = d(n) \wedge$ 
     $\forall m \cdot (m \in N \wedge f(m) = n \Rightarrow c(n) < c(m))$ 
  THEN
     $c(n) := c(n) + 1$ 
  END

```

It remains now for us to introduce two more events for handling the second wave. We have an event for the standard node, `progress_2`, and one for the root called `root`:

```

progress_2  $\hat{=}$ 
  ANY  $n$  WHERE
     $n \in N - \{r\} \wedge$ 
     $d(n) < d(f(n))$ 
  THEN
     $d(n) := d(n) + 1$ 
  END

```

```

root  $\hat{=}$ 
  SELECT
     $d(r) < c(r)$ 
  THEN
     $d(r) := d(r) + 1$ 
  END

```

Figure 8 shows some transitions in the descending wave. This example required 21 proofs, among which 7 needed an interaction.

Example 4: Distributed Mutual Exclusion

This example is also a very classical ones. It has been studied by many authors, for instance in [11], but the solution proposed here is new (to the best of my knowledge). We, supposedly, have a number of processes running in parallel. From time to time some of them want to access a certain “resource” (whose exact nature is not important) in an *exclusive* way. We want to develop a solution that handles this constraint. To begin with, we shall not propose any solution at all. We just want to formally specify the problem so that several distinct solutions could eventually be realized.

The processes in question are all members of a certain fixed set P of processes. Each such process x is supposed to cycle indefinitely on the following three successive phases:

- x is in the, so-called, *non-critical* section: it is thus not using nor willing to use the resource,
- x is in the *pre-critical* section: this corresponds to the process willing to access the resource. It is thus competing with other processes, which are also in the pre-critical section waiting to be admitted in the critical section where the mentioned resource is supposed to be granted exclusively to a single process,
- x is in the *critical* section: it is using the resource.

We shall represent the transitions between these phases by means of the following three events:

- event `ask` corresponds to the transition *non-critical* \rightarrow *pre-critical*,
- event `enter` corresponds to the transition *pre-critical* \rightarrow *critical*,

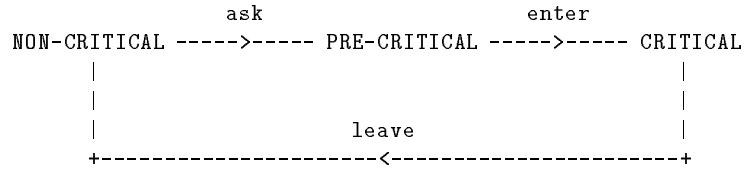


Fig. 9. The transitions between the 3 phases

- even **leave** corresponds to the transition *critical* \rightarrow *non-critical*.

There are three fundamental constraints in this problem, which are the following:

- a process, supposed to be blocked in the non-critical section, *must not block* the processes that are in the pre-critical section: this excludes solutions where each process is given a pre-defined ordered access to the resource whatever its situation,
- a process *must not wait for ever* in the pre-critical section: this encourages solutions where a certain dynamic re-ordering between the processes is realized,
- the critical section contains *at most one process*: this is the basic mutual exclusion property.

The problem is formalized by means of a set p containing the processes in the pre-critical section *as well as the set in the critical section*. The set of processes in the critical section is denoted by c . The critical section should have at most one member. Formally:

$$\begin{array}{l}
 p \subseteq P \\
 c \subseteq p \\
 \forall (x, y) \cdot (x \in c \wedge y \in c \Rightarrow x = y)
 \end{array}$$

We also define a *precedence* relation r possibly holding between members of the set P and members of the set of processes in p . When a pair x, y belongs to r this means that x must not enter the critical section before y (or that y should precede x in entering the critical section). The reason for having y to precede x is because last time x was in the critical section, y was already waiting in the pre-critical section. And, since then, y is still waiting. Consequently, if x re-enters the pre-critical section, it will be unfair to allow it to enter the critical section before y , since otherwise y could be indefinitely blocked in the pre-critical section. Clearly, the relation r should be a *strict partial order*: if z precedes y and y precedes x then z should precede x (otherwise we might have some risk of deadlock), and x cannot be preceded by itself. All this is formalized in what follows:

$$\begin{array}{l}
 r \in P \leftrightarrow p \\
 r \circ r \subseteq r \\
 r \cap r^{-1} = \emptyset
 \end{array}$$

The three events are as follows:

<pre> ask ≐ ANY x WHERE x ∈ P - p THEN p := p ∪ {x} END </pre>	<pre> enter ≐ ANY x WHERE x ∈ p ∧ c = ∅ ∧ x ∉ dom(r) THEN c := {x} END </pre>	<pre> leave ≐ ANY x WHERE x ∈ c THEN c := ∅ p := p - {x} r := (r - (P × {x})) ∪ ({x} × (p - {x})) END </pre>
--	---	--

The third guard, $x \notin \text{dom}(r)$, of event **enter** makes sure that no process does precede x for entering the critical section. Notice that we have a certain non-determinism here as there might exist several candidates not preceded by other processes (remember, r is only a *partial* order). The modification of the relation r upon leaving the critical section in event **leave** makes no process be preceded anymore by x in entering the critical section (since x has just been “served”), and, conversely, all processes present in the pre-critical section should now precede x for entering the critical section. The formal verification that these events maintain the invariant (in particular the strict partial order of the relation r) requires 14 proofs, among which 3 needed an easy interaction.

The development of a specific mutual exclusion algorithm (in subsequent refinements) may:

- implement the relation r by means of a more concrete relation, say h , that only needs to be *weaker* than r (thus h has a domain including that of r). Hence, when a process x is *not* in the domain of h , then it is not, a fortiori, in the domain of r : no process precedes x , which can then safely enter the critical section,
- implement the test $c = \emptyset$ without mentioning c (for certain classes of algorithms). For doing so, the idea is to have the concrete relation h , be a *total order*. And an extra invariant could stipulate that the process in c (if any) is *the smallest process* of p under h . In this case, when a candidate process of p , happens to be *the smallest process* of p under h then, clearly, the critical section c is empty, since h is *total*.

We shall now proceed with a specific refinement. We suppose that the processes are able to communicate with a unique *central agency*, which is responsible for choosing the candidate to be admitted in the critical section. We suppose however that the *channel* connecting the various processes to this agency is known to be possibly “unfair” (that is, a process could overpass some older processes in it). The agency has got a *door*, which could be open or closed, and also a “fair” *pool* of processes waiting to be admitted in the critical section. When open, the agency may read the channel and transfer a process from the channel into the pool. When closed, the agency is supposed to make its choice among the processes of the pool. The set p corresponding to the pre-critical section is now partitioned into two sets: (1) the set ch which is the channel, the local pool pl of the agency (notice that c is included in the pool). This is indicated in figure 10.

The challenge we face now is the following. In the abstraction, the choice was made (when the critical section c was empty) among the processes of the pre-critical section p . Here, the choice is made *only* among the requesting processes that are known to the agency: these are the members of the pool pl . As the channel *is not fair*, we wonder whether a *global fair choice* can nevertheless be realized by the agency, as stipulated in the abstraction. In fact, the agency must take a decision based on a *partial* information only, decision which has nevertheless to be *consistent* with the abstraction. A number of design decisions are envisaged (which have to be validated by a formal proof):

1. The processes enter the pre-critical section through the event **ask**, as in the abstraction. This event has the extra (concrete) effect of posting a request message (with the process name) in the channel ch .

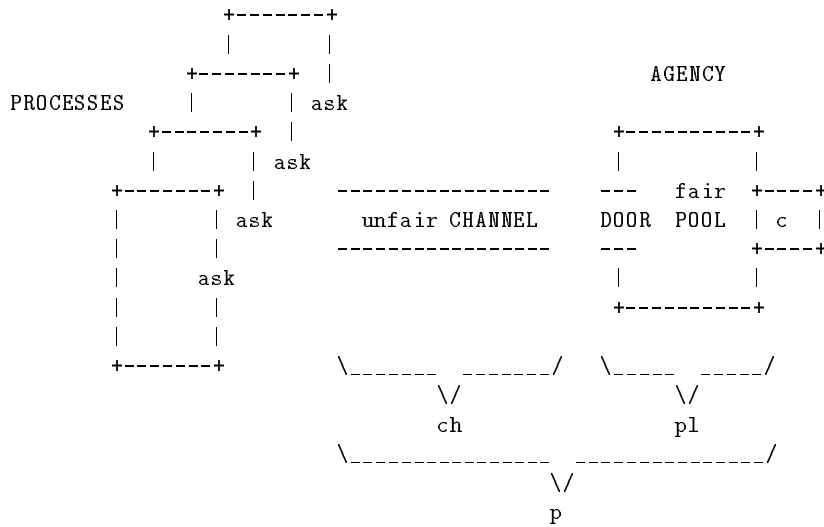


Fig. 10. Partitionning of the pre-critical section p

2. At the other side of the channel, the agency reads such request messages (in an *unfair fashion*) by means of the new event **request** (when the door is open). This event has the effect of removing a certain request message from the channel ch and putting it in the local pool pl of the agency.
3. From time to time, when the pool pl is not empty and when the critical section c is empty, the agency closes its door. This is performed by means of the new event **loop1**. Before closing its door, however, the agency *checks that there are no pending messages in the channel*
4. While the door is closed:
 - (a) the agency makes its choice (it might take a certain time),
 - (b) some processes can post more messages in the channel, but the agency do not read them as its door is now closed. Notice that the processes that have arrived in the channel while the door is closed were not there when the door was last open, since, as we have seen, the door is only made closed when the channel is empty (this is point 3 above).
5. Once the agency has made its choice, it re-opens its door. This is done by the event **enter**, which was already visible in the abstraction.
6. The agency uses the *same* precedence relation r as in the abstraction. This means, when a process x is leaving the critical section, that the agency has to guarantee some new precedence links between x and the members of p (this is stipulated in the abstraction). Such members of p are in:
 - (a) the pool pl : it is easy to give them precedence since the agency has full visibility over this pool,
 - (b) the processes mentioned in the request messages, which have been posted *while the door was closed*: it seems *very difficult* (impossible even) to give precedence to these processes since the agency *does not see them*: this is a challenge for the next refinement.

Formally, we thus have three new variables: the channel ch , the pool pl , and the door dr . The channel and the pool partition the pre-critical section p , which is still a variable of this refinement.

$$\begin{aligned}
ch &\subseteq P \\
pl &\subseteq P \\
dr &\in \{closed, open\} \\
p &= ch \cup pl \\
ch \cap pl &= \emptyset
\end{aligned}$$

When the door is closed then we have a number of extra invariant laws:

1. the critical section c must be empty, so that the agency can fill it,
2. the local pool pl of the agency is not empty, so that the agency can make its choice,
3. the processes mentioned in the request channel ch do not have precedence over any other processes: this is indeed a *very strong condition* that must be very seriously confirmed by the formal proof. Intuitively, this is because the processes in question all come from the non-critical section and were not there when the door was last open (remember, the door can only be closed provided the channel is empty).

$$\begin{aligned}
dr = closed &\Rightarrow c = \emptyset \\
dr = closed &\Rightarrow pl \neq \emptyset \\
dr = closed &\Rightarrow ch \cap \text{ran}(r) = \emptyset
\end{aligned}$$

Next is the new version of the event `ask` and the new events `request` and `loop1`:

```

ask ≐
  ANY x WHERE
    x ∈ P - p
  THEN
    p := p ∪ {x}
    ch := ch ∪ {x}
  END

```

```

request ≐
  ANY x WHERE
    x ∈ ch ∧
    dr = open
  THEN
    ch := ch - {x} ||
    pl := pl ∪ {x}
  END

```

```

loop1 ≐
  SELECT
    c = ∅ ∧
    ch = ∅ ∧
    pl ≠ ∅ ∧
    dr = open
  THEN
    dr := closed
  END

```

Next are the new versions of the events `enter` and `leave`:

```

enter ≐
  ANY y WHERE
    y ∈ pl ∧
    y ∉ dom(r) ∧
    dr = closed
  THEN
    c := {y} ||
    dr := open
  END

```

```

leave ≐
  ANY x WHERE
    x ∈ c
  THEN
    c := ∅ ||
    p := p - {x}
    pl := pl - {x}
    r := (r - (P × {x})) ∪ ({x} × (p - {x}))
  END

```

The formal verification requires 29 proofs, among which 4 needed an easy interaction.

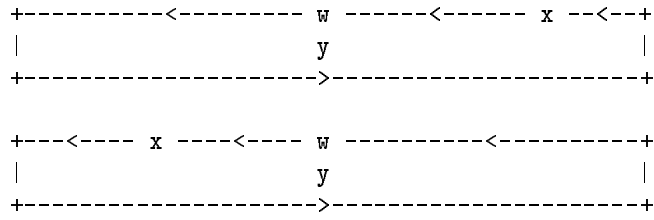
We now proceed with our next refinement. We give more shape to the process names by supposing that they are all natural numbers (we thus have the classical ordering relation on numbers). In this refinement, we shall implement the precedence relation r . For this, we introduce a new variable w (for winner), which is a process. It represents either the unique member of the critical section, or its *previous member*, in case the critical section is now empty. We require that, when w has precedence over some process (that is, when w is in the range of r), then it is indeed in the critical section. Formally:

$$\boxed{\begin{array}{l} wr \in P \\ c \subseteq \{w\} \\ w \in \text{ran}(r) \Rightarrow c = \{w\} \end{array}}$$

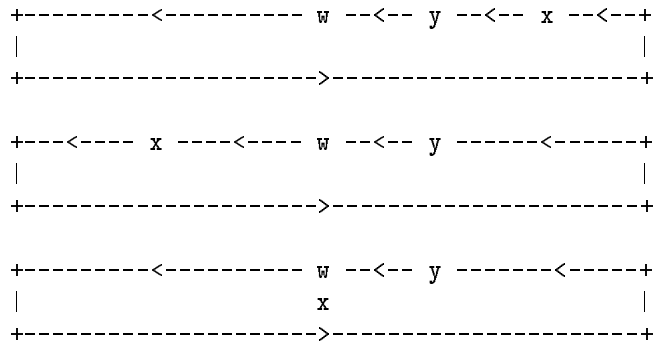
We now define a concrete precedence relation h as follows, together with the gluing invariant stating that h is *weaker* than the abstract precedence relation r :

$$\boxed{\begin{array}{l} h \in P \leftrightarrow P \\ \forall (x, y) \cdot \left(\begin{array}{l} (x, y) \in h \\ \Leftrightarrow \\ (y = w \Rightarrow x \neq w) \wedge \\ (y > w \Rightarrow y < x \vee x \leq w) \wedge \\ (y < w \Rightarrow y < x \wedge x \leq w) \end{array} \right) \\ r \subseteq h \end{array}}$$

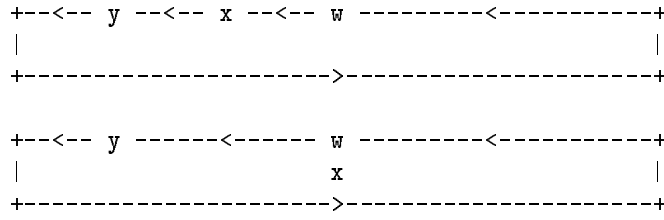
This corresponds to an arithmetic “ring structure”. In the following figure is the case where y is equal to w



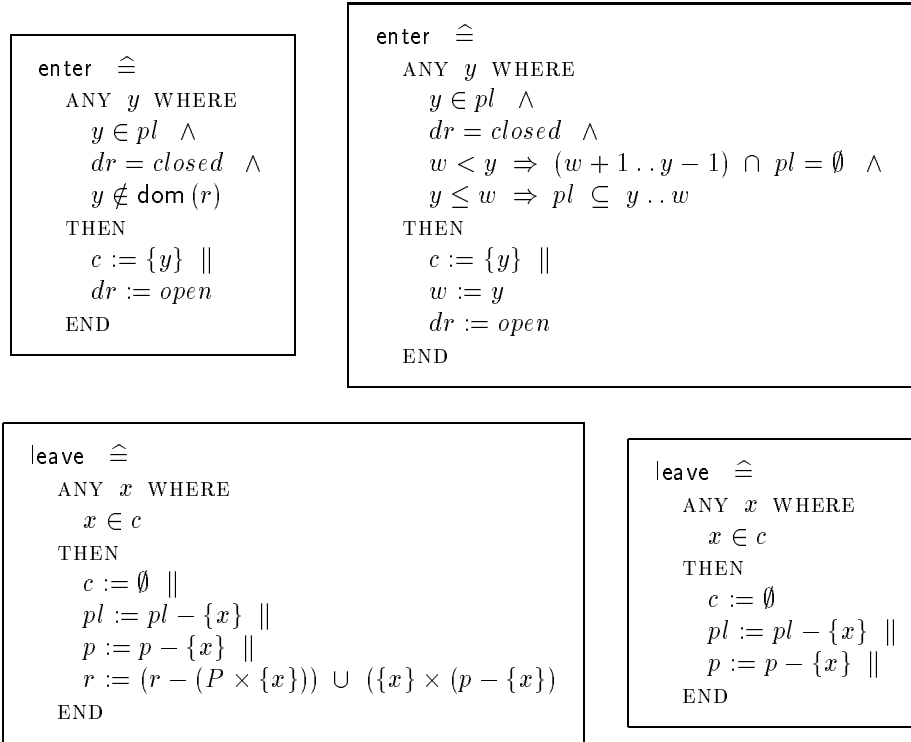
Then comes the case where y is greater than w .



Then finally, the case where y is smaller than w



As can be seen, the predicate $(x, y) : h$ is represented by y situated in front of x when one is going *counterclockwise* in the ring. Notice that when w is equal to y (that is when w is in the range of h) then it is the greatest element of the relation h . Conversely, when w is equal to x (that is when w is in the domain of h), then it is the smallest element of h . Events **enter** and **leave** are the only ones modified. We show each of them preceded by its abstraction so that the difference between the two can be clearly seen:



The guard of the new version of **enter** requires some intuitive explanations. In fact, it says that there is no elements of pl except y that lies “between” w and y . In other words, y is the “greatest” element of pl that is “smaller” than or equal to w , thus y is certainly the process with no process preceding it (Hint: remember the interpretation of h we have given above). Of course, when we write “between” or “smaller” or “natural ordering” it is relative to the counterclockwise ordering on the arithmetic ring structure.

The refinement of **leave** also requires some intuitive explanations as the modification of r has *completely disappeared*. This is because then w (which used to be the process in the critical section) now becomes automatically “smaller” than any other process (hint: look at the two diagrams above where x and w are identical, you will “see” that all other processes y are “greater” than x).

5. The winner gets this message by means of the new event **acknowledge**.

All this is pictured in figure 11. Another and last refinement completes the sequential treatment of the loop in the agency. We leave it to the reader to fill in the details of these two refinements. Their formal verification requires respectively 37 proofs (with 10 done interactively) and 29 (with 12 interactive). The overall development required 119 proofs among which 36 needed an interaction.

Example 5: Termination Detection

This example comes from [14], this is the classical Dijkstra-Scholten algorithm for distributed termination detection. We are only giving here the *essence* of this algorithm leaving the final details to the reader. We shall first state the problem of termination detection in very general terms, then express and prove the main idea of this algorithm.

We have a number of, so-called, *nodes* that can be either *active* or *sleeping*. Such nodes are in fact processes spread on a network. A termination detection algorithm is one that is able to detect that all nodes are sleeping. Notice that we do not require that it detects this fact *as soon as* it holds, only that it detects it *when* it holds.

This can be formalized in a straightforward manner. We are given a set N of nodes. A variable s (for sleeping) denotes the set of sleeping nodes (it is initialized to N : all nodes are thus initially sleeping). We also have a variable d (for detection) which can take one of the two distinct values ok and ko (it is initialized to ko). We state as an invariant that the condition $d = ok$ implies that the condition $s = N$ holds: when d is ok , all nodes are sleeping. Formally:

$$\begin{array}{l} s \subseteq N \\ d \in \{ok, ko\} \\ d = ok \Rightarrow s = N \end{array}$$

We have three events **awake**, **make_asleep**, and **detection**, which are self-explanatory:

```

awake  $\hat{=}$ 
  ANY  $n$  WHERE
     $n \in s$ 
  THEN
     $s := s - \{n\}$  ||
     $d := ko$ 
  END

```

```

make_asleep  $\hat{=}$ 
  ANY  $n$  WHERE
     $n \in N - s$ 
  THEN
     $s := s \cup \{n\}$ 
  END

```

```

detection  $\hat{=}$ 
  SELECT
     $s = N$ 
  THEN
     $d := ok$ 
  END

```

The formal verification of this first model requires 6 automatic proofs. What is not acceptable in this first model is that the detection termination has to observe *all* nodes (in the guard of event **detection**) to conclude that they are all sleeping. We would like to *localize such an observation to one node only*. This is the purpose of the algorithm.

In order to achieve this goal, it is clearly necessary to give more structure to the set of nodes. In fact we suppose that they form a graph g : this is the network. Moreover, we suppose that we have a special node r , which is always asleep. Finally, we suppose that all awakened nodes are forming a tree that *partially spans* the graph g and whose root is r . Note that this tree may also contain some sleeping nodes, but we shall see that in certain circumstances (when they are leaves of the tree), they will be removed from the tree. The tree is defined by means of the “father” function f . Formally:

$$\begin{aligned}
&g \in N \leftrightarrow N \\
&r \in s \\
&f \in N - \{r\} \rightarrow N \\
&f \subseteq g^{-1} \\
&N - s \subseteq \text{dom}(f)
\end{aligned}$$

The inductive property of the tree is defined as usual by means of the following rule:

$$\forall S \cdot \left(\begin{array}{l} S \subseteq N \wedge \\ r \in S \wedge \\ \forall n \cdot (n \in N - \{r\} \wedge f(n) \in S \Rightarrow n \in S) \\ \Rightarrow \\ \text{dom}(f) \subseteq S \end{array} \right)$$

The abstract event **awake** splits into two events **awake_1** and **awake_2**. In both cases, according to the abstraction, the chosen node n must be sleeping. In the first case, n is *not a node of the tree*, therefore it is certainly sleeping. Moreover, there exists supposedly a node m of the tree that is directly connected to n through the graph g . The tree is then extended by linking n to m , and n is awakened. In the second case, n is supposed to be a sleeping node of the tree, it is then just awakened.

$$\begin{aligned}
&\text{awake_1} \hat{=} \\
&\text{ANY } n, m \text{ WHERE} \\
&\quad n \in N - (\text{dom}(f) \cup \{r\}) \wedge \\
&\quad m \in \text{dom}(f) \cup \{r\} \wedge \\
&\quad (m, n) \in g \\
&\text{THEN} \\
&\quad f := f \cup \{n \mapsto m\} \parallel \\
&\quad s := s - \{n\} \parallel \\
&\quad d := ko \\
&\text{END}
\end{aligned}$$

$$\begin{aligned}
&\text{awake_2} \hat{=} \\
&\text{ANY } n \text{ WHERE} \\
&\quad n \in \text{dom}(f) \cap s \\
&\text{THEN} \\
&\quad s := s - \{n\} \parallel \\
&\quad d := ko \\
&\text{END}
\end{aligned}$$

The concrete event **make_asleep** is straightforward: the candidate node n is a non-sleeping node of the tree, it is made sleeping. We have a new event called **shrink**. Its rôle is to remove a *sleeping leaf* of the tree. A leaf is a node n that is in the domain of f but not in its range (that is, n has no “sons”). Finally, the guard of the concrete event **detection** just tests that the root r has no sons. As a consequence the tree is certainly empty (except for the root) and, according to the invariant, all nodes are thus sleeping.

$$\begin{aligned}
&\text{make_asleep} \hat{=} \\
&\text{ANY } n \text{ WHERE} \\
&\quad n \in \text{dom}(f) - s \\
&\text{THEN} \\
&\quad s := s \cup \{n\} \\
&\text{END}
\end{aligned}$$

$$\begin{aligned}
&\text{shrink} \hat{=} \\
&\text{ANY } n \text{ WHERE} \\
&\quad n \in \text{dom}(f) - \text{ran}(f) \wedge \\
&\quad n \in s \\
&\text{THEN} \\
&\quad f := f - \{n \mapsto f(n)\} \\
&\text{END}
\end{aligned}$$

$$\begin{aligned}
&\text{detection} \hat{=} \\
&\text{SELECT} \\
&\quad r \notin \text{ran}(f) \\
&\text{THEN} \\
&\quad d := ok \\
&\text{END}
\end{aligned}$$

The formal verification of this refinement requires 18 proofs among which 2 needed an interaction. The total number of proofs is thus 24 with 2 of them interactive.

Example 6: Distributed Construction of a Breadth-first Spanning Tree

This is also a very classical example treated in many places: for instance, in [15], [13], [11]. The complete mechanization of its proof is quite interesting. We are given a set N of nodes, a graph g constructed on these nodes, and a particular node r . Moreover, the graph is supposed to be *connected* from r . Formally:

$$\begin{array}{l}
 g \in N \leftrightarrow N \\
 r \in N \\
 \forall S. (S \subseteq N \wedge r \in S \wedge g[S] \subseteq S \Rightarrow N \subseteq S)
 \end{array}$$

We would like to construct simultaneously: (1) a tree, of root r , *spanning* the graph g , and (2) a function yielding, at the same time and for each node of the tree, the *depth* of that node in the tree and also the *minimal distance* of that node to the node r in the graph g . The tree is represented by the “father” function f defined on each node except at the root. The distance is defined by a total numeric function l . These are typed as partial functions only. Formally:

$$\begin{array}{l}
 f \in N - \{r\} \rightarrow N \\
 l \in N \rightarrow \mathbb{N}
 \end{array}$$

In the abstraction, the algorithm computes both variables in one shot as follows (where the notation $f, l : P(f, l)$ is to be read “ f and l take some values satisfying the predicate $P(f, l)$ ”):

$$\begin{array}{l}
 \text{result} \hat{=} \\
 \text{BEGIN} \\
 \left(\begin{array}{l}
 f \in N - \{r\} \rightarrow N \wedge \\
 f \subseteq g^{-1} \wedge \\
 \forall S. (S \subseteq N \wedge r \in S \wedge f^{-1}[S] \subseteq S \Rightarrow N \subseteq S) \wedge \\
 f, l : l \in N \rightarrow \mathbb{N} \wedge \\
 l(r) = 0 \wedge \\
 \forall y. (y \in N - \{r\} \Rightarrow l(y) = l(f(y)) + 1) \wedge \\
 \forall (x, y). ((x, y) \in g \Rightarrow l(f(y)) \leq l(x))
 \end{array} \right) \\
 \text{END}
 \end{array}$$

The first three predicates express that f is a “father” function included in the converse of the graph, that the tree is rooted at r and connected from r : the tree, indeed, spans the graph. The next predicates define the

total numeric function l , whose value at r is null. This function is incremented along descending branches of the tree: it thus represents the depth of the node in the tree. Moreover, if y is connected to x in the graph (thus the node x is a potential father of y in the tree) then $l(f(y))$ is smaller than or equal to $l(x)$. In other words, y is connected through the tree to a node $f(y)$, which is by definition one of its neighbours in the graph, and whose l -value is *minimal* among the other l -values of all the neighbours x of y in the graph.

We now proceed with a concrete refinement. The concrete f is the father function of a *partial spanning tree*, which is only defined on the domain of l . The value of l is defined at r , it is equal to 0. Moreover the increasing property of l along descending branches in the tree is not as fine as in the final tree: here it is, at each node y of the tree, only strictly greater than that at $f(y)$, not just equal to that at $f(y)$ plus 1.

$$\begin{aligned}
& f \in \text{dom}(l) - \{r\} \mapsto \text{dom}(l) \\
& f \subseteq g^{-1} \\
& \forall S \cdot (S \subseteq \text{dom}(l) \wedge r \in S \wedge f^{-1}[S] \subseteq S \Rightarrow \text{dom}(l) \subseteq S) \\
& r \in \text{dom}(l) \\
& l(r) = 0 \\
& \forall y \cdot (y \in \text{dom}(l) - \{r\} \Rightarrow l(f(y)) < l(y))
\end{aligned}$$

We have two events computing the tree and the function l : these are **progress_1** and **progress_2**. The guard of the concrete event **result** corresponds to the negation of the guards of the others. This means that the result is obtained when the two events **progress_1** and **progress_2** do *deadlock*.

```

progress_1  $\hat{=}$ 
  ANY  $p, q$  WHERE
     $(p, q) \in g \wedge$ 
     $p \notin \text{dom}(l) \wedge$ 
     $q \in \text{dom}(l)$ 
  THEN
     $f := f \cup \{p \mapsto q\} \parallel$ 
     $l := l \cup \{p \mapsto l(q) + 1\}$ 
  END

```

```

progress_2  $\hat{=}$ 
  ANY  $p, q$  WHERE
     $(p, q) \in g \wedge$ 
     $p \in \text{dom}(l) \wedge$ 
     $q \in \text{dom}(l) \wedge$ 
     $l(q) + 1 < l(p)$ 
  THEN
     $f(p) := q \parallel$ 
     $l(p) := l(q) + 1$ 
  END

```

```

result  $\hat{=}$ 
  SELECT
     $\text{dom}(l) = N \wedge$ 
     $\forall (p, q) \cdot (p, q) \in g \Rightarrow l(p) \leq l(q) + 1$ 
  THEN
    skip
  END

```

In event **progress_1**, a *new* node p (“new” because p is not already a member of the tree) is connected to a node q of the tree (this is, of course, because p and q are neighbours in the graph): the value of l at p is thus made equal to $l(q) + 1$. In event **progress_2**, we discover that node p , which is already a member of the tree, can be assigned a “better” father q than its actual father $f(p)$. This is because $l(q)$ is strictly smaller than $l(p) - 1$, the “normal” value of the function l at the father of p . Node p is given this new father q and the value of l at p is adjusted to one more than that, $l(q)$, of the new father of p . Notice that the father of p could already be q . In that case, only the value of l at p is improved.

The formal proof that the event **progress_2** does not destroy the tree structure is a bit elaborate. It is due to the intuitive fact that the modification in the tree does not result in a cycle. Informal hint: the new father

q of p has a smaller l -value than that of p , it cannot thus be a descendant of p , since the l -values of such descendants are strictly greater than that of p . Formally it amounts to proving that the modified induction rule holds, under the hypotheses of the invariant (not written here) and of the guard of the event. This yields:

$$\begin{aligned} & p \in \text{dom}(l) \ \wedge \\ & q \in \text{dom}(l) \ \wedge \\ & l(q) + 1 < l(p) \\ \Rightarrow \\ & S \subseteq \text{dom}(l) \ \wedge \ r \in S \ \wedge \ (f \triangleleft \{p \mapsto q\})^{-1}[S] \subseteq S \ \Rightarrow \ \text{dom}(l) \subseteq S \end{aligned}$$

This is to be proved under the induction rule which is an invariant:

$$\forall S. (S \subseteq \text{dom}(l) \ \wedge \ r \in S \ \wedge \ f^{-1}[S] \subseteq S \ \Rightarrow \ \text{dom}(l) \subseteq S)$$

The idea is to instantiate this rule with S . We already have the antecedents $S \subseteq \text{dom}(l)$ and $r \in S$. But we do not have the antecedent $f^{-1}[S] \subseteq S$, only $(f \triangleleft \{p \mapsto q\})^{-1}[S] \subseteq S$. Now, if we can prove that $q \in S$ holds, then we are done since, in this case, it is easy to prove that $(f \triangleleft \{p \mapsto q\})^{-1}[S]$ is equal to $f^{-1}[S]$. In order to prove that $q \in S$ holds, the “idea” is to instantiate the induction rule with the set:

$$S \cup \{p\} \cup \{x \mid x \in \text{dom}(l) \ \wedge \ l(p) + 1 \leq l(x)\}$$

Provided we can prove *the instantiated antecedent* of the induction rule (which is not done yet), we have its consequent, namely:

$$\text{dom}(l) \subseteq S \cup \{p\} \cup \{x \mid x \in \text{dom}(l) \ \wedge \ l(p) + 1 \leq l(x)\}$$

Clearly q does belong to the right hand side of this inclusion since it belongs to its left hand side $\text{dom}(l)$. But now it is easy to prove that q does *not* belong to the set

$$\{p\} \cup \{x \mid x \in \text{dom}(l) \ \wedge \ l(p) + 1 \leq l(x)\}$$

As a consequence q belongs to S . It just remains then to prove the main instantiated antecedent of the induction rule (the first two are trivial), namely:

$$f^{-1}[S \cup \{p\} \cup \{x \mid x \in \text{dom}(l) \ \wedge \ l(p) + 1 \leq l(x)\}] \subseteq S \cup \{p\} \cup \{x \mid x \in \text{dom}(l) \ \wedge \ l(p) + 1 \leq l(x)\}$$

The left hand side can be decomposed into three parts: $f^{-1}[S]$, $f^{-1}[\{p\}]$, and $f^{-1}[\{x \mid x \in \text{dom}(l) \ \wedge \ l(p) + 1 \leq l(x)\}]$. It is not difficult to prove both inclusions

$$f^{-1}[\{p\}] \subseteq \{x \mid x \in \text{dom}(l) \ \wedge \ l(p) + 1 \leq l(x)\}$$

and

$$f^{-1}[\{x \mid x \in \text{dom}(l) \ \wedge \ l(p) + 1 \leq l(x)\}] \subseteq \{x \mid x \in \text{dom}(l) \ \wedge \ l(p) + 1 \leq l(x)\}$$

according to the invariant properties of f and l . We finally prove easily:

$$f^{-1}[S] \subseteq S \cup \{p\}$$

since we already have:

$$(f \triangleleft \{p \mapsto q\})^{-1}[S] \subseteq S$$

and also clearly

$$f^{-1}[S] = (f \triangleleft \{p \mapsto q\})^{-1}[S] \cup \{p\}$$

The formal verification of this example requires 33 proofs, among which 11 are interactive. The one sketched above is the only one that is difficult: it has entirely been discovered and conducted with Atelier B.

We leave it to the reader to engage in another refinement consisting in taking some ideas from the previous example (termination detection) in order to simplify the guard of the event `result` (Hint: do a gradual and “virtual” shrink of the spanning tree when the two events `progress_1` and `progress_2` are deadlocked).

Example 7: A Distributed Routing Algorithm for Mobile Agents

This example is taken from [12]. A, so-called, *mobile agent* \mathcal{M} is supposed to travel between various sites. Fixed agents situated in the sites in question want to establish some communications with \mathcal{M} . To simplify matters, such communications are supposed to be unidirectional: they take the practical form of messages sent from the fixed agents to \mathcal{M} .

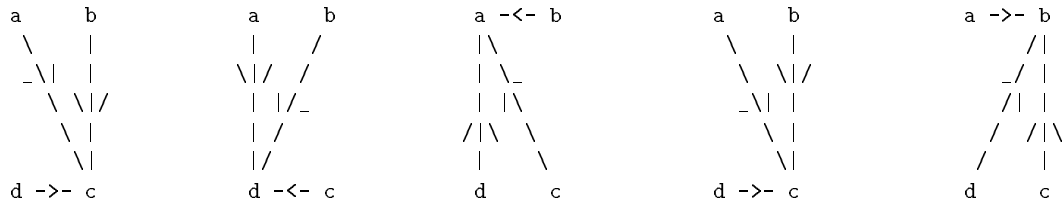


Fig. 12. The mobile agent goes from site c successively to sites d , a , c , and b .

In an ideal (*abstract*) world, the movements of \mathcal{M} from one site to another one are instantaneous, as well as the knowledge by the fixed agents of the exact position of \mathcal{M} . In this case, the fixed agents “follow” \mathcal{M} as indicated by figure 12.

In a more realistic (*concrete*) world, the movements of \mathcal{M} from one site to another one are still instantaneous, but the only site to know where \mathcal{M} is the site that it has just left. The other sites are not aware of the movement, they continue thus to send messages to the site where they still *believe* that \mathcal{M} resides. As it might not be the case anymore, then some messages arrive to a destination which is not currently that of \mathcal{M} . As a consequence, each site, besides sending its own messages, is thus also in charge of possibly *forwarding* the messages received while \mathcal{M} is not present anymore locally. It is quite possible that several such forwardings take place before a message eventually reaches \mathcal{M} . This is illustrated in figure 13. As can be seen, when \mathcal{M} reaches a new site, that site destroys the previous knowledge it has concerning the location of \mathcal{M} . For instance, in the third snapshot, where \mathcal{M} has just moved to site a , the link between a and c that existed in the previous situation is removed. Similarly, when \mathcal{M} leaves a site, that site re-actualizes its knowledge by “storing” the new location of \mathcal{M} (again, supposed to be known instantaneously). In the fourth snapshot where \mathcal{M} has just moved to c , a new link between a and c is established. Intuitively for the moment, we can figure out that the communication channels are dynamically modified while maintaining a tree structure whose root is the actual site of \mathcal{M} as shown on figure 14. Each site is then indirectly *connected* to the site of \mathcal{M} and there exists *no circuit* that might put some forwarded messages in an endless loop.

In a still more realistic world, the movements of \mathcal{M} between sites are not instantaneous anymore. In fact, when \mathcal{M} leaves a site, it does not know necessarily where it is going. Only when \mathcal{M} arrives at its destination,

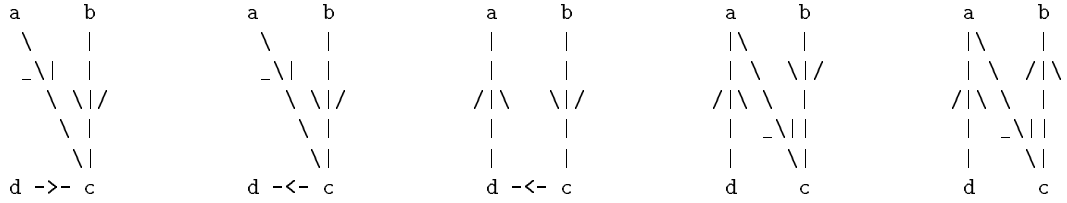


Fig. 13. The left site is the only one to know where the mobile agent is.

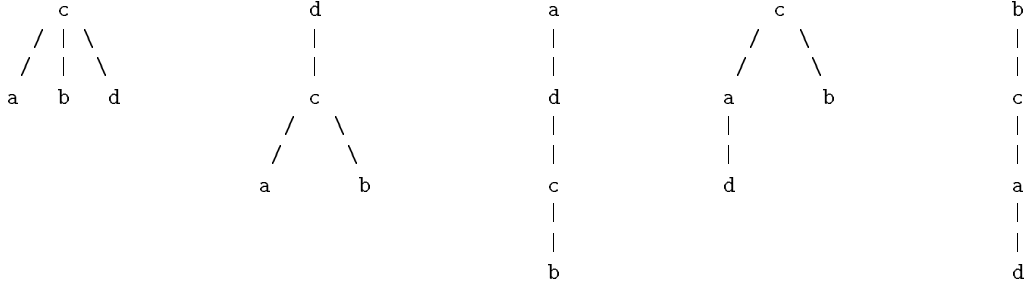


Fig. 14. The communication channels form a tree structure.

is it able to send a “service” message to its previous site in order to inform it of its present location. Of course, the message in question does not itself travel instantaneously. Communication messages are thus still forwarded from sites to sites, but that forwarding might be *suspended* in some sites, which \mathcal{M} has left in the past, until such sites receive a service message informing them of the “actual” location of \mathcal{M} (actual, however, when the message was sent, maybe not anymore when the message is received).

We have no control over the relative speed of the service messages: some of them can reach their destination quite quickly, while some others might take more time (but we suppose that they will eventually arrive at their destination). In figure 15, we have put some dashed lines to indicate that the corresponding service messages have not yet arrived: notice that messages following the dashed lines circulate in a direction that is the opposite one of that followed by the communication messages that will be established upon reception of the service message. In fact, when a service message arrives at its destination, the corresponding dashed line is transformed in a “plain” line going in the opposite direction.

In figure 15, we have shown a series of snapshots where all service messages are pending. Notice that the situation pictured in the last diagram contains a potential problem. This is because site c is expecting two service messages. In fact a site might expect as many service messages as there has been past visits of that site by the mobile agent. In the present case, if the service message between d and c is (very) late, then, upon arrival it may have the disastrous effect of: (1) isolating completely site b , and (2) forming a circuit within which communication messages may circulate for ever. This is shown in figure 16.

This is due to the fact that site c is misled by the late incoming service message. It should have *discarded* it. But how can c know that this message is a late one that is not meaningful anymore? The purpose of the distributed routing algorithm presented here and developed by L. Moreau in [12] is precisely to solve this problem. The idea of L. Moreau is to have \mathcal{M} travelling with a *logical clock* that is incremented each time it moves from one site to the other. Upon arrival in a site the local clock is stored. It thus records the “time”

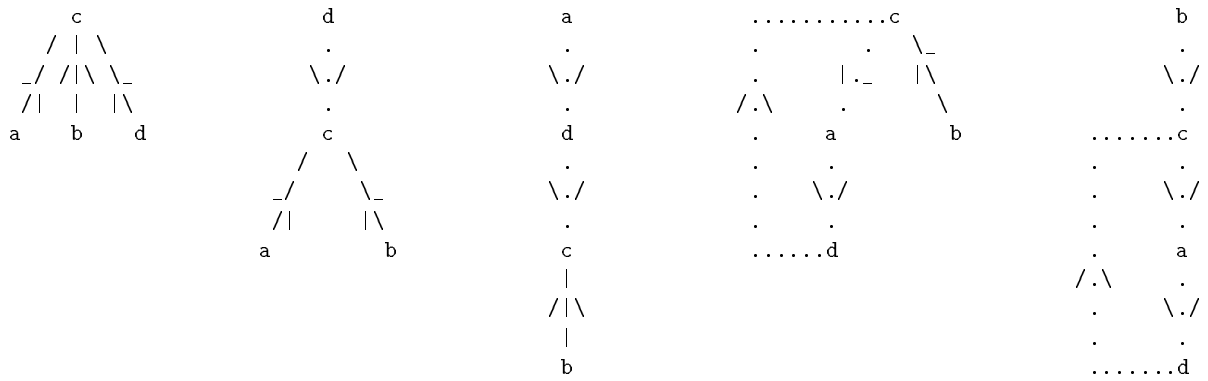


Fig. 15. The plain lines represent the communication channels, and the dashed ones the service channels.

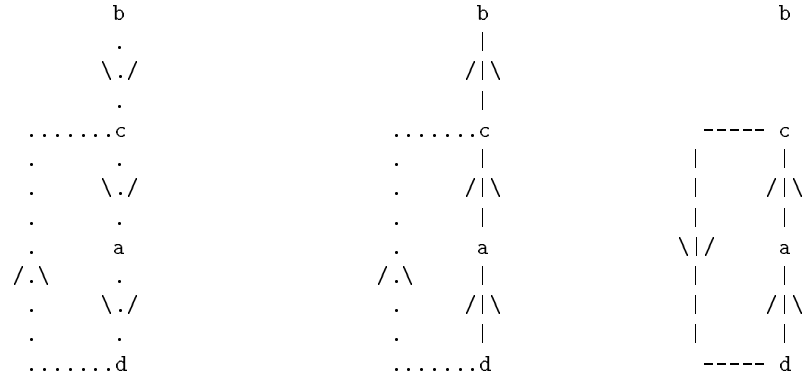


Fig. 16. The late arrival of a service message may cause serious communication problems

of the last visit of \mathcal{M} . When \mathcal{M} sends its service message, it stamps that message with the new time (the one that has just been incremented and recorded in the new site). As a consequence, the service message is stamped with a value that is certainly greater than that recorded in its destination. When the service message arrives, it is filtered: if the stamp is smaller than or equal to the local time then it is discarded. Simultaneously, the local clock of the site is updated with the stamp travelling with the service message so that this message could not be used a second time (in case of misbehaviour of the network). Figure 17 shows the previous series of situations painted with the clocks and the stamps.

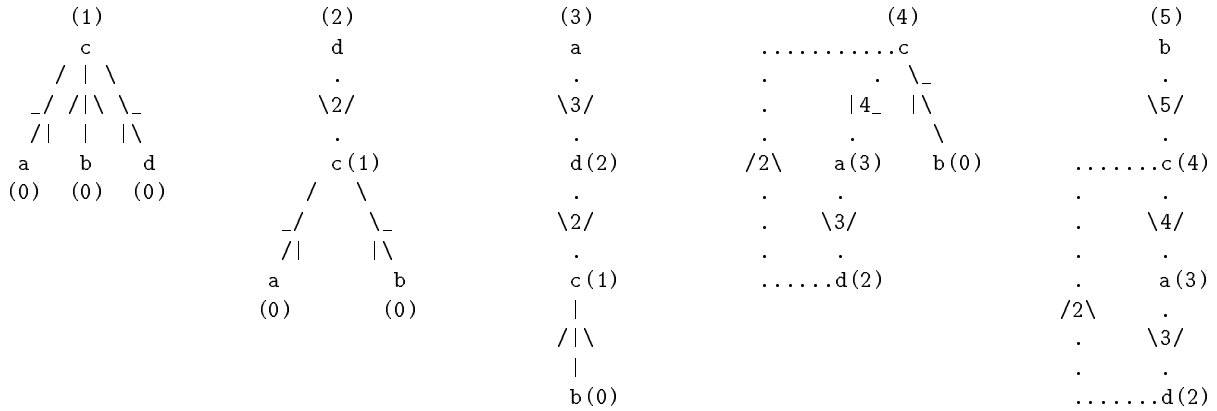


Fig. 17. Each site records the “time” of the last visit. Service messages are stamped.

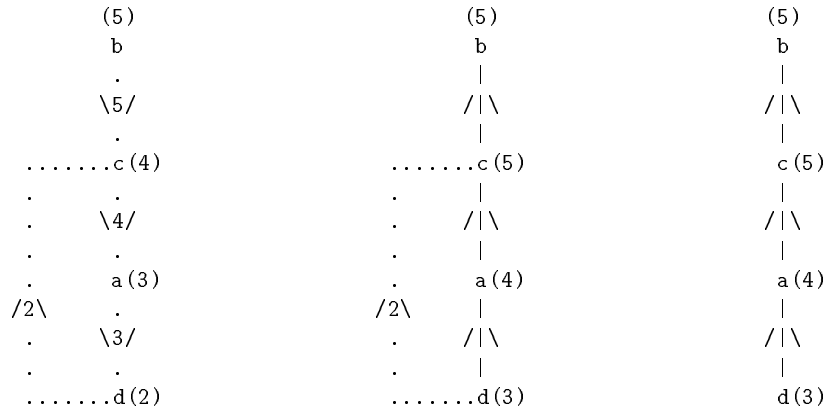


Fig. 18. The late service message is discarded

We now have enough information to start the formal construction of this routing protocol. The first question that one must ask oneself with an example such as this one (and many others) is that concerning the *level of description* we have to start from. It is out of the question to start from the final solution, because then nothing really can be proved. We must start from an abstract enough level, which must be pretty obvious (and where

no technical difficulties exist yet, in particular those dealing with time and distances), so that the proposed solution can be proved to indeed solve the problem that has been informally described.

In the present case, we are going to start from the second “abstract” informal level, where the exact position of the mobile is only known by its previous site. This level is quite simple: the communication channels, as we have said informally, form a tree structure (an invariant that owes to be proved, of course), the moves of the mobile are timeless, and finally the knowledge concerning the new location of the mobile is instantaneously communicated to its previous site (no “service” message thus).

We have two abstract sets S and M : the set S denotes the set of sites and M the set of communication messages. To have an abstract set representing the set of communication messages is really a useful abstraction since we are not interested in the contents of these messages: we can then suppose that they are all distinct. We then have three variables in our model: l , c , and p . The variable l denotes the actual location of the mobile. The variable c denotes the dynamically changing communication channels between sites: it is a total function from sites to sites. Notice that this function is obviously not meaningful at l . Finally the variable p denotes the pool of messages that are waiting to be forwarded on each site: clearly a given message is at most in one site at a time, so that p is a partial function from messages to sites. Initially the mobile is at some pre-defined site and all channels are pointing to that site. Formally:

$$\begin{array}{l} l \in S \\ c \in S \rightarrow S \\ p \in M \rightarrow S \end{array}$$

It remains now for us to formalize the tree structure of the communication channels. The root of the tree is l and the father function is the function c where the pair $l \mapsto c(l)$ has been removed. We shall use exactly the same induction rule as the one introduced in **Example 3** above, namely:

$$\forall T \cdot \left(\begin{array}{l} T \subseteq S \wedge \\ l \in T \wedge \\ \forall s \cdot (s \in T - \{l\} \wedge c(s) \in T \Rightarrow s \in T) \\ \Rightarrow \\ S \subseteq T \end{array} \right)$$

We have four events: **rcv_agt** corresponds to the mobile moving instantaneously from the site l to another one (different from l), **snd_msg** corresponds to a new message sent from one site to the mobile, **fwd_msg** corresponds to a message being forwarded from one site to another one by means of the corresponding channel (note that this transfer is also instantaneous for the moment), and finally **dlv_msg** corresponds to a message being delivered to the mobile. Here are these events:

$\begin{array}{l} \text{rcv_agt} \hat{=} \\ \text{ANY } s \text{ WHERE} \\ \quad s \in S - \{l\} \\ \text{THEN} \\ \quad l := s \parallel \\ \quad c(l) := s \\ \text{END} \end{array}$	$\begin{array}{l} \text{snd_msg} \hat{=} \\ \text{ANY } s, m \text{ WHERE} \\ \quad s \in S \wedge \\ \quad m \in M - \text{dom}(p) \\ \text{THEN} \\ \quad p(m) := s \\ \text{END} \end{array}$	$\begin{array}{l} \text{fwd_msg} \hat{=} \\ \text{ANY } m \text{ WHERE} \\ \quad m \in \text{dom}(p) \wedge \\ \quad p(m) \neq l \\ \text{THEN} \\ \quad p(m) := c(p(m)) \\ \text{END} \end{array}$	$\begin{array}{l} \text{dlv_msg} \hat{=} \\ \text{ANY } m \text{ WHERE} \\ \quad m \in \text{dom}(p) \wedge \\ \quad p(m) = l \\ \text{THEN} \\ \quad p := p - \{m \mapsto p(m)\} \\ \text{END} \end{array}$
--	---	--	---

It should be noted that the channel structure is, for the moment, modified instantaneously when the mobile moves to another site. The formal verification of this model required 10 proofs among which 2 needed an easy interaction.

In our first refinement, we are more concrete: when the agent move (still instantaneously) to a new site, it sends an acknowledgement message to its previous site, in order for that site to update its forward pointer. So, during the transmission delay, the former site cannot transmitt any forward message because it does not know where the agent is. It knows however that it is expecting an acknowledgement message.

Clearly the channel structure in this new refined model is not in phase with that of the previous model where it was modified at the very time of the move of the agent. We have thus a new variable, d , denoting this new channel structure. The abstract function c is more up to date than d : this is due to the delay within which the acknowledgement messages travel. We have a variable, a , denoting the acknowledgement channel containing the “service” messages mentioned in the informal description. It is a partial function from sites to sites: more precisely, from the site where the mobile was before the move to the site where it is currently (this function contains the *future* of the communication channel). We shall make precise in what follows the reason why a is such a *function*, which is far from obvious a priori: we shall see that, in this abstraction, this channel has a rather “magic” behaviour. Formally:

$$\begin{array}{l} d \in S \rightarrow S \\ a \in S \rightarrow S \\ c = d \triangleleft a \end{array}$$

The last invariant is the so-called *gluing invariant* between the abstract communication channel c and its concrete counterpart d . It says that the abstract channel c corresponds to the concrete one, d , *overridden* by the acknowledgement channel. The various concrete events are very close to their abstraction. The events `snd_msg` and `dlv_msg` do not change. Here are the two others:

$$\begin{array}{l} \text{rcv_agt} \hat{=} \\ \text{ANY } s \text{ WHERE} \\ \quad s \in S - \{l\} \\ \text{THEN} \\ \quad l := s \parallel \\ \quad a(l) := s \\ \text{END} \end{array}$$

$$\begin{array}{l} \text{fwd_msg} \hat{=} \\ \text{ANY } m \text{ WHERE} \\ \quad m \in \text{dom}(p) \wedge \\ \quad p(m) \neq l \wedge \\ \quad p(m) \notin \text{dom}(a) \\ \text{THEN} \\ \quad p(m) := d(p(m)) \\ \text{END} \end{array}$$

The event `fwd_msg` has a stronger guard than its abstraction: a message whose site is in the domain of the acknowledgement channel cannot be sent since its destination is not yet known (notice that the event can have an “appropriate” look within the acknowledgement channel, a privilege that will be removed in further refinements). Finally, the event `rcv_agt` has a very interesting behaviour. We note that when putting a new message $\{l \mapsto s\}$ in the acknowledgement channel a , we “magically” *remove the other pair (if any) whose first component was l* . In other words, we “naturally” clean the channel by removing old messages that might have not been yet delivered. In this way there is at most one acknowledgement message pointing to a given site, and there is thus no risk of having an old message arriving late (that is, after a more recent one) and having the kind of bad effect we have described above. Of course, this is quite magic for the moment. What we only wanted to express at this level is the intended behaviour of the channel. It will remain, of course, to be implemented,

which is another matter: this will be the business of the next refinement.

Finally, we have a new event corresponding to the reception by a site of an acknowledgement message informing it of the new location of the mobile (notice that after we have just said, it is indeed the *actual location* of the mobile otherwise the message would have been discarded). The communication channel is updated and the acknowledgement message removed from its channel. Here is the event:

$$\begin{array}{l}
 \text{rcv_ack} \hat{=} \\
 \text{ANY } s \text{ WHERE} \\
 \quad s \in \text{dom}(a) \\
 \text{THEN} \\
 \quad d(s) := a(s) \parallel \\
 \quad a := a - \{s \mapsto a(s)\} \\
 \text{END}
 \end{array}$$

The formal verification of this refinement requires 18 proofs, among which 3 needed an interaction.

In the next refinement, we shall implement the magic acknowledgement channel of previous abstraction. This is the heart of the development. We now have a clock k travelling with the agent: it is also called the “ticket”. We also have in each site a , so-called, *mobility counter*, r , recording the “time” of the last visit of the mobile in the corresponding site. Formally:

$$\begin{array}{l}
 k \in \mathbb{N} \\
 r \in S \rightarrow \mathbb{N}
 \end{array}$$

Finally the new acknowledgement channel b has now a structure far richer than its abstraction a . It may contain several stamped messages to the same site s . These messages are of the form, say:

$$s \mapsto \{\{3 \mapsto s1\}, \{5 \mapsto s2\}, \{9 \mapsto s3\}, \dots\}$$

This means that there has been a message $s \mapsto s1$ emitted at “time” 3, a message $s \mapsto s2$ emitted at “time” 5, a message $s \mapsto s3$ emitted at “time” 9, etc. the channel b is thus typed as follows:

$$b \in S \rightarrow (\mathbb{N} \rightarrow S)$$

Next comes the invariant “gluing” the abstract acknowledgement channel a and the concrete one b : the acknowledgement message to s in the abstract channel corresponds, among all acknowledgement messages to s in the concrete channel, to the one with the greatest time (the most recent one):

$$\forall s \cdot (s \in \text{dom}(a) \Rightarrow a(s) = b(s)(\max(\text{dom}(b(s))))$$

When the mobility counter $r(s)$ of the recipient s of an acknowledgement message is strictly smaller than the maximum time of the pending acknowledgement messages for that recipient then the recipient in question is indeed expecting a real message as in the abstraction. But the problem, of course, is that, *a priori*, the recipient does not know that it is indeed receiving the maximum in question (this difficulty will be circumvented below in the last invariant law). The next law allows us to prove the guard strenghtening of event `rcv_ack` (with the help of the last law below)

$$\forall s \cdot (s \in S \wedge r(s) < \max(\text{dom}(b(s))) \Rightarrow s \in \text{dom}(a))$$

We now have three more laws concerned with the mobility counters and the ticket. (1) The “times” in the pending acknowledgement messages are never bigger than the ticket. (2) The mobility counter is equal to the ticket at the site of the agent. (3) In other sites, the mobility counter is at most equal to the ticket

$$\begin{aligned} \forall s \cdot (s \in S \Rightarrow \max(\text{dom}(b(s))) \leq k) \\ r(c) = k \\ \forall s \cdot (s \in S - \{l\} \Rightarrow r(s) \leq k) \end{aligned}$$

Now comes at last the *key invariant law*. When the recipient of an acknowledgement message receives a message with a time strictly greater than its own mobility counter then it can be *absolutely certain* that it is indeed receiving the message with the greatest time, therefore the same message as in the abstraction according to the main gluing invariant.

$$\forall (s, n) \cdot (s \in S \wedge n \in \text{dom}(b(s)) \wedge r(s) < n \Rightarrow n = \max(\text{dom}(b(s))))$$

This is not completely intuitive. The informal explanation is as follows. If several acknowledgement messages are expected at a site s , this means that the mobile has visited s several times. And on each such visit it has updated the mobility counter of s with the most recent value of the clock. Upon leaving the site it has sent (when arriving at its new location) an acknowledgement message to s with a stamp value which is one more than that of the mobility counter of s at the time. So, during its *last* visit, which has certainly taken place *after* the sending of the previous acknowledgement messages, the last value of the mobility counter of s was then certainly greater than that of the stamp of any pending acknowledgement messages. As a consequence, when the mobile leaves s again for the last time it sends (upon arrival at its new location) yet another acknowledgement message, which is then *the only one* with a stamp greater than the value of the mobility counter of s . *All this, clearly, needs confirmation from a formal proof.* Thanks to this invariant, we can safely implement the “magic” abstract channel a with the concrete channel b . The only events that differs from their abstraction are events rcv_agt where acknowledgement message are sent and event rcv_ack where they are received:

```
rcv_agt ≐
  ANY s WHERE
    s ∈ S - {l}
  THEN
    l := s ||
    r(s) := k + 1 ||
    k := k + 1 ||
    b(l)(k + 1) := s
  END
```

```
rcv_ack ≐
  ANY s, n WHERE
    s ∈ S ∧
    n ∈ dom(b(s)) ∧
    r(s) < n
  THEN
    d(s) := b(s)(n) ||
    r(s) := n
  END
```

Note that, to simplify matters, we do not clean the channel b in event rcv_ack . As a matter of fact, *it is not necessary*. Since the abstract channel was cleaned (the refinement is correct), this means that the message will not be accepted another time. This is because of the updating of the mobility counter, this gives us a “cleaning effect”. The formal verification of this refinement required 31 proofs, among which 15 needed an interaction.

There are two more refinements. In the next one, we allow for a possible delay in the migration of the mobile. There is thus a time where the mobile is not any more at site l and has not yet arrived at another site s . In fact, during that time, the agent is “situated” between the two and there is a “migration message” in progress containing the new destination site s of the mobile. The mobile is supposed to travel with its former site l and with the ticket k . In the last refinement, we implement the effective migration of the forwarded communication messages. We leave it as an exercise to the reader to develop these refinements.

The verification of these last refinements required respectively 19 proofs (among which 7 interactive) and 18 (among which 10 interactive). The overall formal verification effort required 96 proofs among which 37 are interactive (38%).

Acknowledgement: I would like to thank D. Cansell and D. Méry for many helpful discussions on these matters.

References

1. J.R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press (1996).
2. J.R. Abrial. *Extending B Without Changing it (for Developing Distributed Systems)*. In H. Habrias, editor, *1st Conference on the B-Method*. November 1996.
3. J.R. Abrial and L. Mussat. *Introducing Dynamic Constraints in B*. In D. Bert editor, *B'98: Recent Advances in the Development and Uses of the B-Method*. LNCS vol 1393. 1998.
4. J.R. Abrial. *Event Driven Sequential Program Construction*. (to appear) 2000.
5. J.R. Abrial. *Event Driven Electronic Circuit Construction*. (to appear) 2001.
6. J.R. Abrial. *Event Driven System Construction*. (to appear) 1999.
7. J.R. Abrial and L. Mussat. *Specification and Design of a Transmission Protocol by successive Refinements Using B*. In M. Broy and B. Schieder editors, *Mathematical Methods in Program Development* Springer NATO ASI Series Vol. 158. 1996
8. N.V. Stenning. *A Data Transfer Protocol*. Computer Networks, September 1976.
9. K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. *A note on reliable full-duplex transmission over half-duplex links*. Communications of the ACM. May 1969.
10. G. Le Lann. *Distributed systems - towards a formal approach*. In B Gilchrist, editor *Information Processing 77* North-Holland 1977.
11. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers 1996
12. L. Moreau. *Distributed Directory Service and Message Routing for Mobile Agent*. Science of Computer Programming 1999.
13. W.H.J. Feijen and A.J.M. van Gasteren. *On a Method of Multi-programming* Springer 1999.
14. E.W. Dijkstra and C. S. Scholten. *Termination detection for diffusing computations*. Information Processing Letters. August 1980.
15. E.W. Dijkstra. *A Note on Two Problems in Connexion with Graphs*. Numerische Mathematik. 1959.