IV. A Simple File Transfer Protocol (October 2008)

The example introduced in this chapter is quite different from the previous one, where the program was supposed to control an external situation (cars on a bridge or mechanical press). Here we present a, so-called, protocol to be used on a computer network by two agents. This is the very classical two-phase handshake protocol. This example has been presented in many places. A very nice presentation is the one given in the book by L. Lamport [1].

This example will allow us to extend our usage of the mathematical language with such constructs as partial and total functions, domain and range of functions, and function restrictions. We shall also extend our logical language by introducing universally quantified formulas and corresponding inference rules.

1 Requirements

The purpose of the protocol is to transfer a sequential file from one agent, the sender, to another one, the receiver. The transmitted file should be equal to the original file.

The protocol ensures the copy of a file from one site to another one	FUN-1

The sequential file, as its name indicates, is made of a number of items disposed in a an ordered fashion.

The file is supposed to be made of a sequence of items	FUN-2

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 kinds of messages on the network.

The file is send piece by piece between the two sites	FUN-3

Such programs are working on different machines: the overall protocol is indeed a *distributed program*.

2 Refinement Strategy

We are not going to model right away the final protocol, this would be too complicated and error prone. The refinement strategy we are going to adopt is explained now.

In the initial model (section 3), the idea is to present the final result of the protocol which one can observe when the protocol is finished. At this initial stage, the two participants in the protocol - the sender and the receiver - are not supposed to reside on different sites. This is a technique which we shall always use when modelling protocols. This initial model is important because it tells us exactly *what the protocol is supposed to achieve without telling us how*.

In the first refinement (section 4), we shall separate the sender and the receiver. Moreover, the file will be transmitted piece by piece between them, not in one shot as in the initial model. However, this separation of the sender and the receiver will not be complete: we suppose that the receiver can "see" what remains to be transmitted in the sender's site and is able to take "directly" the next item from the sender and add it to its own file. At this stage, we explain the essence of the algorithm but we do not see yet the details of the distributed behavior as performed on each site. This kind of refinement is very important in the modelling of a protocol: we simplify our task by allowing separate participants to "cheat" by looking directly into other participants private memories.

In the next refinement (section 5), the receiver is not cheating any more: it is not able to access directly the sender's site. In fact, the sender will *send messages* that the receiver will read. The receiver then responds to these messages by returning some *acknowledgment messages* to the sender. The fine details of the distributed algorithm are revealed in full. What is important here is that the messages between the participants can be seen as a means of implementing the previous abstraction where the receiver could have a direct access to the contents of the sender's memory. Again, this is a technique that we shall frequently use in protocol modelling.

In the last refinement (section 6), we shall optimize what is sent between the two participants. The protocol is not modified any more, it is made just more efficient.

3 Protocol Initial Model

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 finite sequence f. 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 at the receiver's site on a file named g supposed to be empty initially. This is illustrated on Figure 1 below.

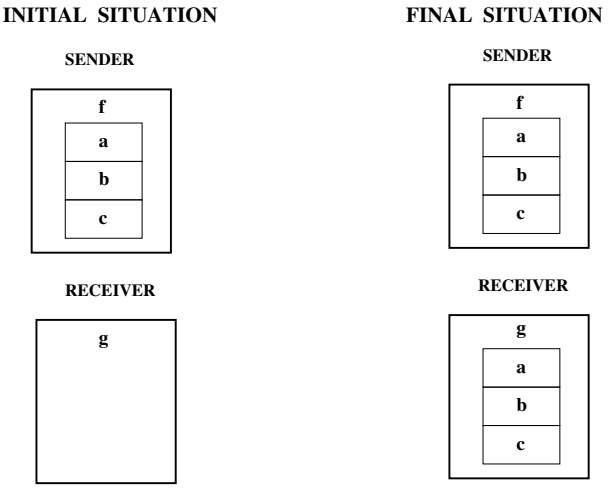


Fig. 1. Initial and Final Situations

3.1 The State

The context is made of a set D, which is called a *carrier set*. This set represents the data that are stored in the file. The only implicit property that we assume concerning carrier sets is that they are not empty. The presence of this set makes our development *generic*. It means that the set D could be instantiated later to a particular set. Furthermore, we have two constants. First the constant n, which is a positive natural number, and second the constant f, which is a *total function* from the interval $1 \,..\, n$ to the set D. This is the way we formalize finite sequences. These properties are written below as **axm0_1** and **axm0_2**.

sets: D

constants: n, f

axm0_1: $0 < n$

axm0_2: $f \in 1 \,..\, n \rightarrow D$

We have a variable, g, which is a *partial function* from the interval $1 \,..\, n$ to the set D. It is written below in invariant **inv0_1**. We have also a boolean variable b stating when the protocol is finished ($b = \text{TRUE}$). As we shall see below in section 3.3, variable g is empty when the protocol is not finished whereas equal to f when the protocol is finished. This is formalised in invariants **inv0_2** and **inv0_3**

variables: g, b

inv0_1: $g \in 1 \,..\, n \nrightarrow D$

inv0_2: $b = \text{FALSE} \Rightarrow g = \varnothing$

inv0_3: $b = \text{TRUE} \Rightarrow g = f$

3.2 Reminder of Mathematical Notations

In the previous sections we have used some mathematical concepts such as intervals, partial functions, and total functions. We recall here a few notations and definitions concerning such concepts and similar ones.

Given two natural numbers a and b, the interval between a and b is the set of natural numbers x where $a \leq x$ and $x \leq b$. It is denoted by the construct $a \,..\, b$. Note that when b is smaller than a, the interval $a \,..\, b$ is empty.

$x \in S$	set membership operator
$\mathbb{N}$	set of natural numbers: $\{0, 1, 2, 3, \ldots\}$
$a \,..\, b$	interval from a to b: $\{a, a+1, \ldots, b\}$ (empty when $b < a$)

Given two sets S and T, and two elements a and b belonging to S and T respectively, the *ordered pair* made of a and b in that order is denoted by the construct $a \mapsto b$. The set of all such ordered pairs made out of S and T is called the *Cartesian product* of S and T. It is denoted by the construct $S \times T$.

Given a set T, the fact that a set S is a *subset* of T, is denoted by the predicate $S \subseteq T$. The set of all subsets of a set S is called the *power set* of S. It is denoted by the construct $\mathbb{P}(S)$.

$a \mapsto b$	pair constructing operator
$S \times T$	Cartesian product operator: the set of all pairs from S to T
$S \subseteq T$	set inclusion operator
$\mathbb{P}(S)$	power set operator: set of all subsets of a given set S

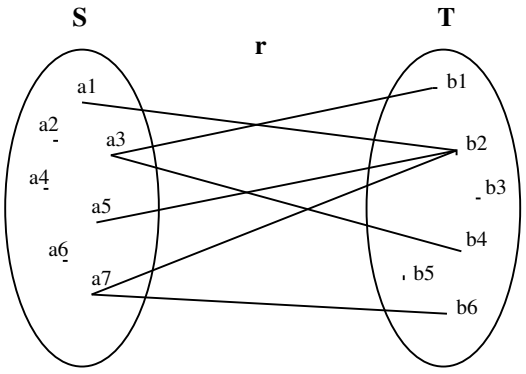
Given two sets S and T, the power set of their Cartesian product is called the set of *binary relations* built on S and T. It is denoted by $\mathbb{P}(S \times T)$, usually abbreviated by the construct $S \leftrightarrow T$. A binary relation is thus a set of pairs. It can be empty. In this case it is denoted by the empty set $\varnothing$.

Given a binary relation r built on two sets S and T (thus r belongs to the set $S \leftrightarrow T$), the *domain* of r is the subset of S whose elements x are such that there exists an element y belonging to T such that the pair $x \mapsto y$ belongs to r. It is denoted by the construct $\mathrm{dom}(r)$.

Symmetrically, the *range* of r is the subset of T whose elements y are such that there exists an element x of S such that the pair $x \mapsto y$ belongs to r. It is denoted by the construct $\mathrm{ran}(r)$.

$S \leftrightarrow T$	set of binary relations from S to T
$\mathrm{dom}(r)$	domain of a relation r
$\mathrm{ran}(r)$	range of a relation r

Next is an illustration of a binary relation r between sets S and T:

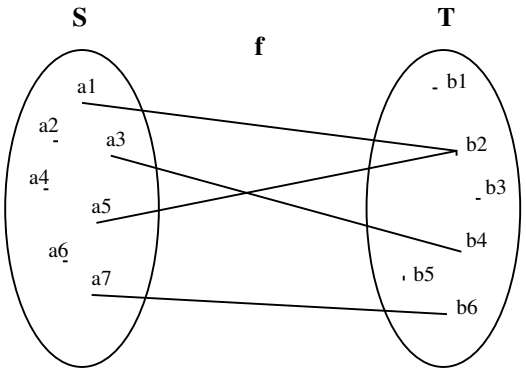


Given two sets S and T, a *partial function* f from S to T is a binary relation from S to T where any two pairs $x \mapsto y$ and $x \mapsto z$ belonging to f are such that y is equal to z. The set of all partial functions from S to T is denoted by the construct $S \nrightarrow T$.

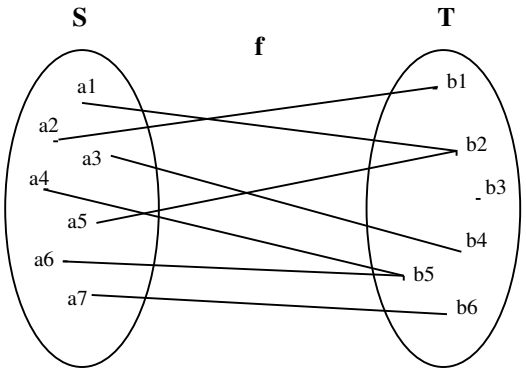
Given two sets S and T, a *total function* f from S to T is a partial function from S to T whose domain is exactly S. The set of all total functions from S to T is denoted by the construct $S \rightarrow T$.

$S \nrightarrow T$	set of partial functions from S to T
$S \rightarrow T$	set of total functions from S to T

Next is an illustration of a partial function f from set S to set T, where the domain of f is the set $\{a1, a3, a5, a7\}$ and its range is the the set $\{b2, b4, b6\}$:



Next is an illustration of a total function f from set S to set T. As can be seen, the domain of f is now exactly S:



3.3 The Events

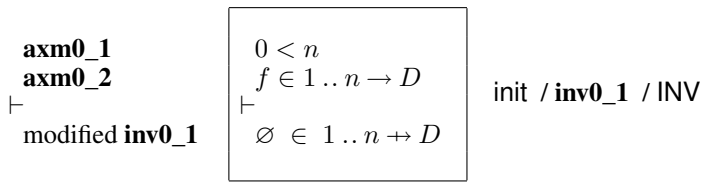
Coming back to our example, let us now define the events of our first model. Initially, g is empty. This is indicated in the special event init below. The very global transfer action of the protocol can be abstracted by means of a *single* event called final:

Event final does not exist by itself. In other words, it is not part of the protocol: it is just a *temporal 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 into the file g.

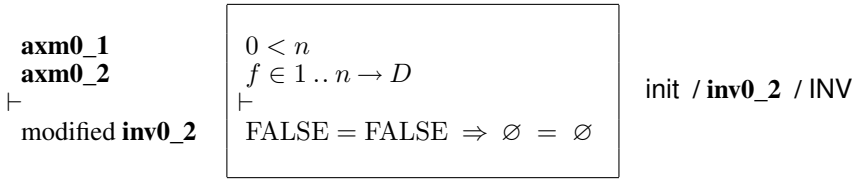
At this point, it should be noted that we are not committed to any particular protocol: this model is thus, in a sense, the most general one corresponding to a given class of protocols, namely that of file transfers. Some more sophisticated specifications could have been proposed, in which the file might have only been partially transfered (this case will be studied in chapter 6), but such an extension is not studied in the present example.

3.4 Proofs

Let us now turn our attention to the proofs. At this stage, the only proofs which are to be considered are invariant proofs and the deadlock freeness proof. Here are the proof obligations concerning the establishment of invariants **inv0_1** and **inv0_2** by the initialization event init. Here is the first proof obligation concerning the establishment of invariant **inv0_1** by event init:



The corresponding proof will be done by using informal arguments only: clearly the empty function is a partial function from $1 \mathinner{..} n$ to D. For proofs involving set theoretic constructs, we shall not provide specific inference rules as we have done in chapter 2 for propositional logic and equality, we shall instead use a "generic" inference rule named SET, which we shall justify informally each time. Here are now the proof obligations concerning the establishment of invariant **inv0_2** and **inv0_3** by event init:



axm0_1
axm0_2
⊢
modified **inv0_3**

$0 < n$
$f \in 1 \mathinner{..} n \to D$
⊢
$\mathrm{FALSE} = \mathrm{TRUE} \Rightarrow \varnothing = f$

init / **inv0_3** / INV

The corresponding proofs can be done easily. Here is the proof obligation concerning the preservation of invariant **inv0_1** by event final:

axm0_1
axm0_2
inv0_1
inv0_2
inv0_3
guard
⊢
modified **inv0_1**

$0 < n$
$f \in 1 \mathinner{..} n \to D$
$g \in 1 \mathinner{..} n \pfun D$
$b = \mathrm{FALSE} \Rightarrow g = \varnothing$
$b = \mathrm{TRUE} \Rightarrow g = f$
$b = \mathrm{FALSE}$
⊢
$f \in 1 \mathinner{..} n \pfun D$

final / **inv0_1** / INV

After applying MON, the proof goes as indicated below. A total function from one set to another is indeed a partial function built on the same sets.

$f \in 1 \mathinner{..} n \to D$
⊢
$f \in 1 \mathinner{..} n \pfun D$

SET

Here are the proof obligations concerning the preservation of invariant **inv0_2** and **inv0_3** by event final:

axm0_2
axm0_3
inv0_1
inv0_2
inv0_3
guard
⊢
modified **inv0_2**

$0 < n$
$f \in 1 \mathinner{..} n \to D$
$g \in 1 \mathinner{..} n \pfun D$
$b = \mathrm{FALSE} \Rightarrow g = \varnothing$
$b = \mathrm{TRUE} \Rightarrow g = f$
$b = \mathrm{FALSE}$
⊢
$\mathrm{TRUE} = \mathrm{FALSE} \Rightarrow g = \varnothing$

final / **inv0_2** / INV

axm0_2
axm0_3
inv0_1
inv0_2
inv0_3
guard
⊢
modified **inv0_3**

$0 < n$
$f \in 1 \mathinner{..} n \to D$
$g \in 1 \mathinner{..} n \pfun D$
$b = \mathrm{FALSE} \Rightarrow g = \varnothing$
$b = \mathrm{TRUE} \Rightarrow g = f$
$b = \mathrm{FALSE}$
⊢
$\mathrm{TRUE} = \mathrm{TRUE} \Rightarrow f = f$

final / **inv0_3** / INV

The corresponding proofs can be done easily using inference rules introduced in chapter 2.

4 Protocol First Refinement

4.1 Informal Presentation

We are now going to *refine* the file transfer done in one shot by the previous *abstract* event final acting "magically" on the receiver's side. For this, we have an additional *concrete* event named receive corresponding to an intermediate phase of the protocol. It aims at transferring the file *piece by piece*. Of course, the abstract event final should not disappear: it will have a concrete counterpart in which the same observation as in the abstraction can be done.

On Figure 2 below, you can see on top what could have been observed in the abstraction, namely the init event followed by the final event. On the bottom, you can see what we can observe during this refinement. We can say that the observer is now opening eyes more often than in the abstraction. It is possible to observe a number of occurrences of the event receive in between that of event init and that of event final.

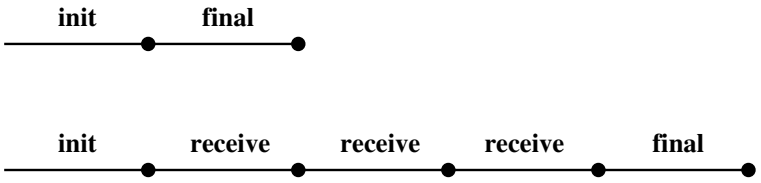


Fig. 2. Initial Abstraction and First Refinement Observations

We change the variable g to another one, h, which is modified by event receive. In fact, this event will gradually copy the file f from the sender's side to the receiver's side. For this it will use an index r which is progressing as indicated in Figure 3 below.

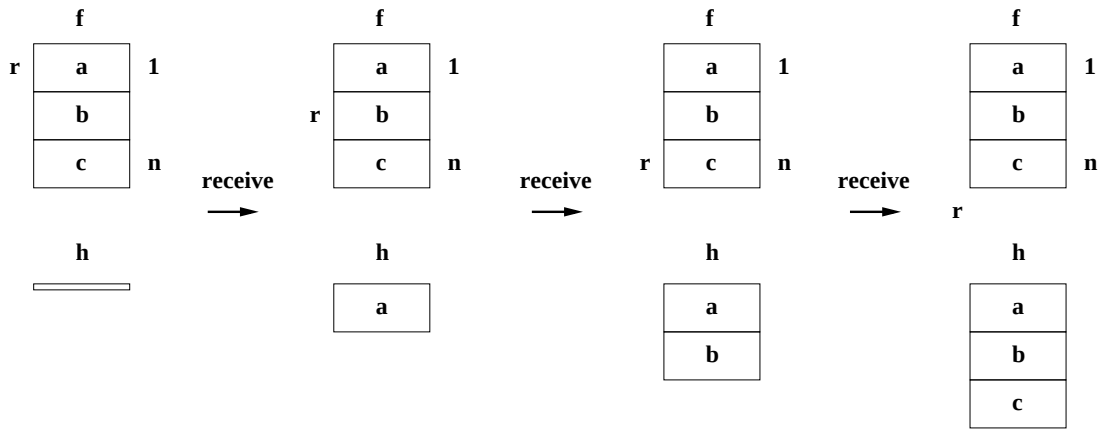


Fig. 3. A Trace of the First Refinement Behavior

As can be seen, event receive is adding an element to file h by copying the rth element of file f to file h. It will be interesting to see what event final does now (wait until section 4.4).

4.2 The State

We enlarge our state by adding to it a variable r which is a natural number. This variable is initialized to 1. It will serve as an index on the file f: it is thus within the interval $1 \,..\, n+1$ as indicated in invariant **inv1_1** below. We also replace the variable g by another one named h. Variable h is exactly equal to the constant f with domain *restricted* to the interval $1 \,..\, r-1$ (see next section). This is written by means of the following construct: $(1 \,..\, r-1) \lhd f$. In other words, in $(1 \,..\, r-1) \lhd f$ we are considering only those pairs $x \mapsto y$ of f where x is in the set $1 \,..\, r-1$. This is recorded in invariant **inv1_2**. Finally, we have to establish the connection between the concrete variable h and the abstract variable g: at the end of the protocol (when b is TRUE) then r must be equal to $n+1$. This is stated in invariant **inv1_3**. It is then easy to prove theorem **thm1_1** stating that g is equal to h when b is equal to TRUE.

variables: b, h, r

inv1_1: $r \in 1 \,..\, n+1$

inv1_2: $h = (1 \,..\, r-1) \lhd f$

inv1_3: $b = \text{TRUE} \Rightarrow r = n+1$

thm1_1: $b = \text{TRUE} \Rightarrow g = h$

4.3 More Mathematical Symbols

In the previous section we have introduced operator $\lhd$ for restricting the domain of a relation. In this section we introduce more restriction operators.

Given a relation r form S to T and a subset s of S, expression $s \lhd r$ denotes the relation r with only those pairs whose first element is in s. It is called a domain restriction.

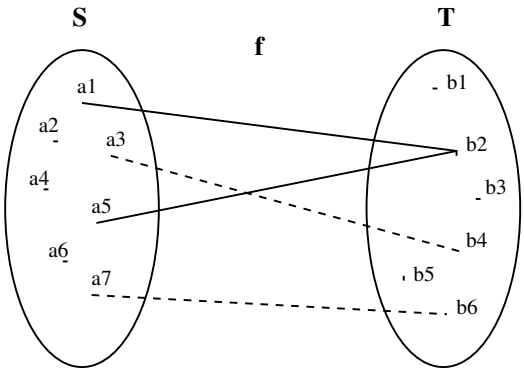
Given a relation r form S to T and a subset s of S, expression $s \lhdd r$ denotes the relation r with only those pairs whose first element is not in s. It is called domain subtraction.

Given a relation r form S to T and a subset t of T, expression $r \rhd t$ denotes the relation r with only those pairs whose second element is in t. It is called a range restriction.

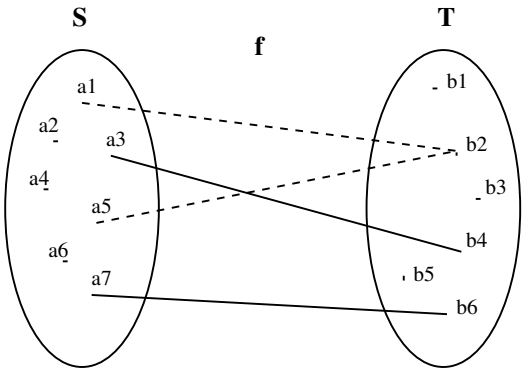
Given a relation r form S to T and a subset s of S, expression $r \rhdd t$ denotes the relation r with only those pairs whose second element is not in t. It is called range subtraction.

$s \lhd r$	domain restriction operator
$s \lhdd r$	domain subtraction operator
$r \rhd t$	range restriction operator
$r \rhdd t$	range subtraction operator

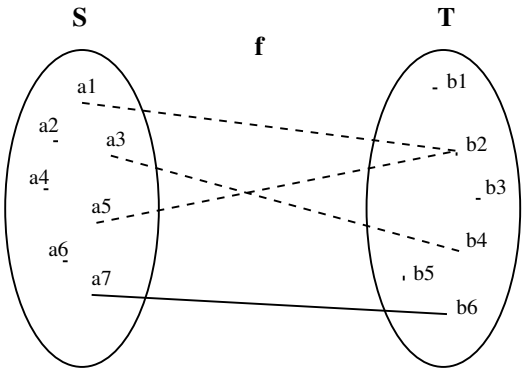
Next is an illustration where the dotted lines correspond to $\{a3,\ a7\} \lhd f$.



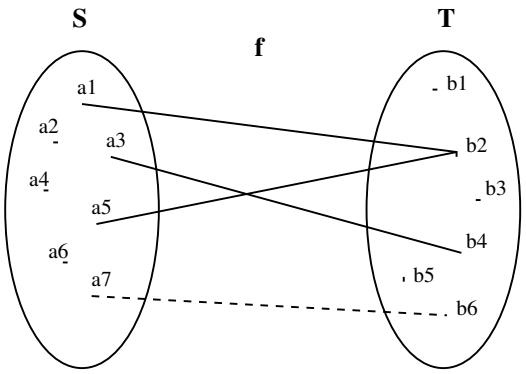
Next is an illustration where the dotted lines correspond to $\{a3,\ a7\} \lhd\!\!\cdot\ f$.



Next is an illustration where the dotted lines correspond to $f \rhd \{b2, b4\}$.



Next is an illustration where the dotted lines correspond to $f \rhd\!\!\cdot\ \{b2, b4\}$.



4.4 The Events

Coming back to our example, let us now define the events of this refinement. The initializing event init set b to FALSE as in the abstraction, h to the empty set and r to 1. Event receive is adding an element to file h by copying the rth element of file f into h, it also increment r, and event final does nothing (just setting b to TRUE as in the abstraction)! This seems strange, but we shall prove that it refines its abstraction. In fact, it just acts now as a witness: when its guard is true, that is when condition $r = n + 1$ holds and b is FALSE, then file g must be equal to file f as stipulated in its abstraction.

init
$b := \text{FALSE}$
$h := \varnothing$
$r := 1$

receive
status
convergent
when
$r \leq n$
then
$h := h \cup \{r \mapsto f(r)\}$
$r := r + 1$
end

final
when
$r = n + 1$
$b = \text{FALSE}$
then
$b := \text{TRUE}$
end

Notice the **status** of event receive: it is convergent meaning that we have to prove that it cannot "keep control" for ever. For proving this, we have to exhibit a numerical **variant** and prove that event receive decreases it. This will be done in section 4.6.

4.5 Refinement Proofs

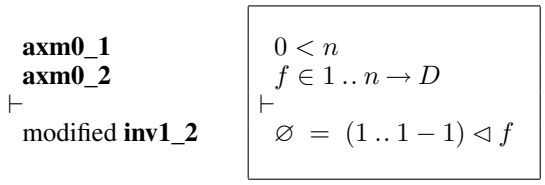
The proof for the initializing event init is simple. Here is the proof obligation for the establishment of invariant **inv1_1**:

axm0_1
axm0_2
$\vdash$
modified **inv1_1**

$0 < n$
$f \in 1 \mathinner{..} n \rightarrow D$
$\vdash$
$1 \in 1 \mathinner{..} n + 1$

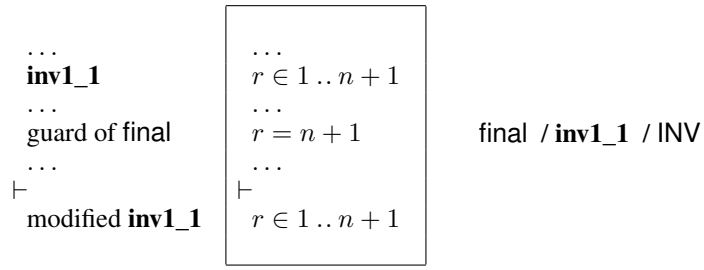
init / **inv1_1** / INV

The proof is done easily by transforming the goal $1 \in 1 \mathinner{..} n+1$ into $1 \leq 1 \land 1 \leq n+1$. Then we apply inference rule AND_R followed by simple arithmetic calculations. Here is now the proof obligation for the establishment of invariant **inv1_2**:

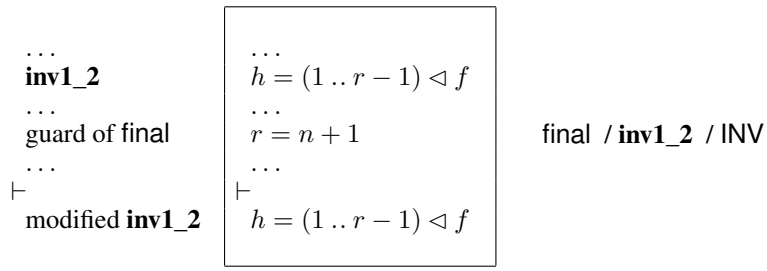


For proving this, we first transform the interval $1 \mathinner{..} 1-1$ into $1 \mathinner{..} 0$, which is empty. Then we notice that the expression $\varnothing \lhd f$ denotes the empty set. We finally apply inference rule EQL. Finally, proving **inv1_3** is trivial.

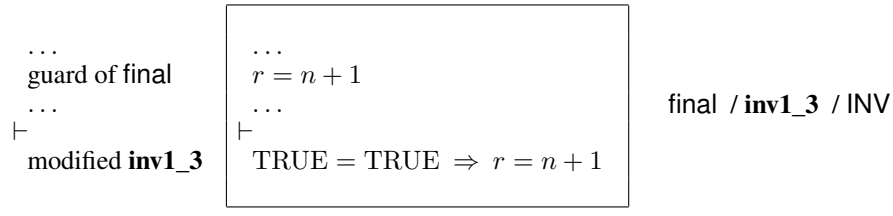
More interesting are the refinement proofs for event final. First, we have to apply the proof obligation rule GRD, which is obvious since the guard of the concrete version, namely $r = n+1$, is clearly stronger than that of the abstraction which is missing (thus always true). Applying now rule INV to invariant **inv1_1** leads to the proof of the following sequent, whose proof is obvious according to inference rules MON and then HYP (since the goal $r \in 1 \mathinner{..} n+1$ is also an hypothesis):



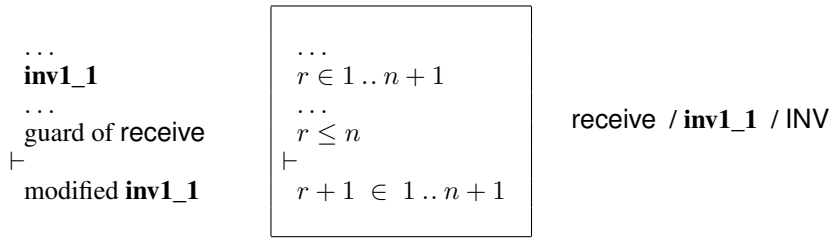
Likewise, invariant **inv1_2** is trivially proved using inference rules MON and HYP:



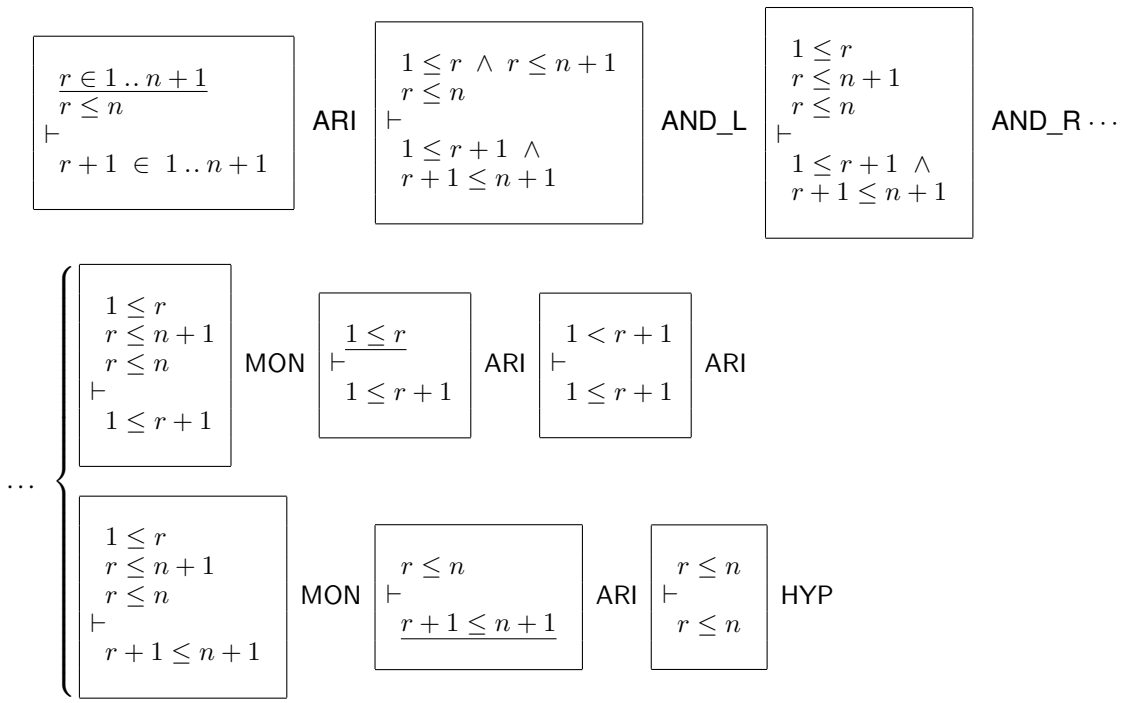
The preservation of invariant **inv1_3** by event final requires proving the following which is obvious:



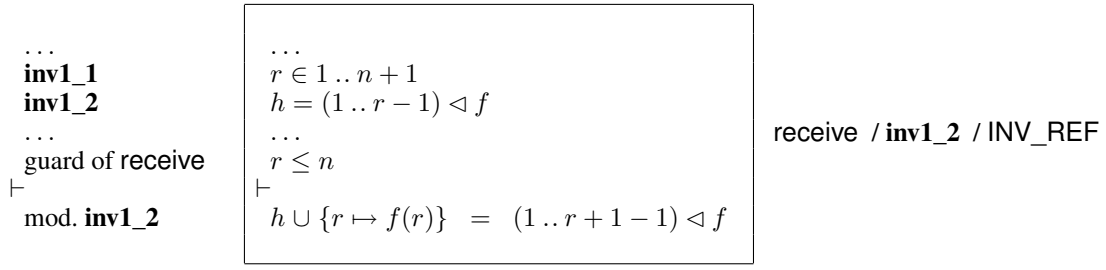
The preservation of invariant **inv1_1** by event receive requires proving:



Here is the proof after applying MON:



The preservation of invariant **inv1_2** by event receive requires proving:



Here is the proof after applying MON:

$$\frac{f \in 1\,..\,n \to D \quad r \in 1\,..\,n+1 \quad h = (1\,..\,r-1) \lhd f \quad r \le n}{h \cup \{r \mapsto f(r)\} \;=\; (1\,..\,r+1-1) \lhd f}\ \text{ARI}\qquad \frac{f \in 1\,..\,n \to D \quad 1 \le r \quad h = (1\,..\,r-1) \lhd f \quad r \le n}{h \cup \{r \mapsto f(r)\} = (1\,..\,r) \lhd f}\ \text{EQ\_LR}\ldots$$

$$\ldots\ \frac{f \in 1\,..\,n \to D \quad 1 \le r \quad r \le n}{(1\,..\,r-1) \lhd f \;\cup\; \{r \mapsto f(r)\} \;=\; (1\,..\,r) \lhd f}\ \text{SET}$$

The last sequent is discharged by noticing that adding the mini-function $\{r \mapsto f(r)\}$ (where r is in the domain of f) to the function f restricted to the interval $1\,..\,r-1$ yields exactly f restricted to the interval $1\,..\,r$. The preservation of invariant **inv1_3** by event receive requires proving:

$$\frac{\ldots \quad \textbf{inv1\_3} \quad \text{guard of receive}}{\text{modified } \textbf{inv1\_3}}\qquad \frac{\ldots \quad b = \text{TRUE} \Rightarrow r = n+1 \quad r \le n}{b = \text{TRUE} \Rightarrow r+1 = n+1}\ \text{receive / inv1\_3 / INV\_REF}$$

The proof goes as follows after applying MON:

$$\frac{b = \text{TRUE} \Rightarrow r = n+1 \quad r \le n}{b = \text{TRUE} \Rightarrow r+1 = n+1}\ \text{IMP\_R}\qquad \frac{b = \text{TRUE} \Rightarrow r = n+1 \quad r \le n \quad b = \text{TRUE}}{r+1 = n+1}\ \text{IMP\_L}\ \ldots$$

$$\ldots\ \frac{r = n+1 \quad r \le n \quad b = \text{TRUE}}{r+1 = n+1}\ \text{EQL\_LR}\qquad \frac{\dfrac{n+1 \le n}{b = \text{TRUE}}}{r+1 = n+1}\ \text{ARI}\qquad \frac{\bot \quad b = \text{TRUE}}{r+1 = n+1}\ \text{FALSE\_L}$$

4.6 Convergence Proof of Event receive

We have to prove that the new event receive converges. For this, we have to exhibit a variant, that is a non-negative expression which is decreased by event receive. The most obvious variant is the following:

variant1: $n+1-r$

Proving that this variant is decreased is easy. We have to apply proof obligation rules NAT (the variant denotes a natural number) and VAR (the variant is decreased by event receive). The proof of the decreasing of this variant is extremely important because it shows that the concrete "execution" of event final might be *eventually reachable*. In other words, it shows that our initial goal stated in the abstract version of final might be reachable in the concrete version despite the new event receive. We have written "might be" on purpose, because what we have proved is that event receive cannot be executed for ever. But it might stop in a position where event final cannot be enabled because its guard would not be true. It is precisely the purpose of the next section to prove that this cannot happen.

4.7 Proving Relative Deadlock Freeness

We are now going to prove that this system never deadlocks (as was the case for the abstraction). Applying rule DLF, it is easy to prove that the disjunction of the guards of events receive and final is always true. Applying the rule leads to the following after some simplifications:

$$\begin{array}{l} r \in 1 \mathinner{..} n+1 \\ \vdash \\ r \leq n \ \lor \ r = n+1 \end{array}$$

As can be seen, the "execution" is the following: init, followed by one or more "executions" of event received, followed by a single "execution" of event final.

5 Protocol Second Refinement

The previous refinement is not satisfactory as the event receive, supposedly "executed" by the receiver, has a direct access to the file f which is supposed to be situated at the sender site. We want to have a more distributed execution of this protocol. Our observer is now opening eyes more frequently and he can see that another event, send, occurs before each occurrence of event receive. On Figure 4 you can see first what the observer could see at previous stages and, in the bottom, what he can see now.

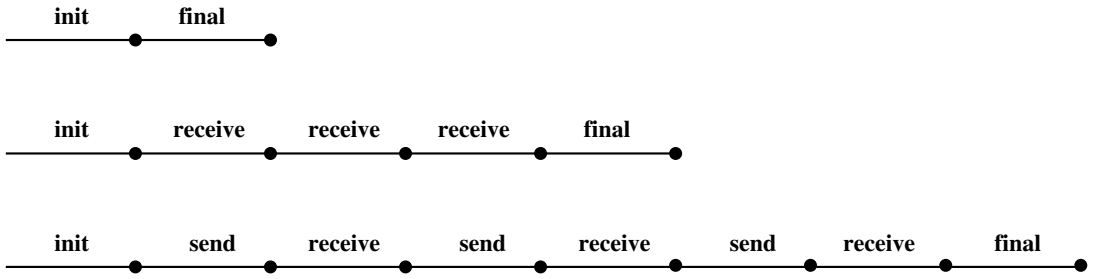


Fig. 4. A Trace of the Second Refinement Behavior

5.1 The State and the Events

The sender has a local counter, s, which records the index of the next item to be sent to the receiver (initially, s is set to 1). When a transmission does occur, the data item d, which is equal to $f(s)$, is sent to the receiver, the counter s is incremented, and the new value of s is also sent together with d to the receiver (event send). Notice that the sender does not immediately send the next item. It waits until it receives an *acknowledgement* from the receiver. This acknowledgement, as we shall see, is the counter r.

When the receiver receives a pair "index-item", it compares the received counter with r and accepts the item if the counter it receives is different from r (event receive). In this case, r is incremented and then sent as an acknowledgment. When the sender receives a number r which is equal to its own counter s, it considers this to be an acknowledgement and proceeds with the next item, and so on.

The sender and the receiver are thus connected by means of two channels as indicated on Figure 5: the data channel and the acknowledgement channel.

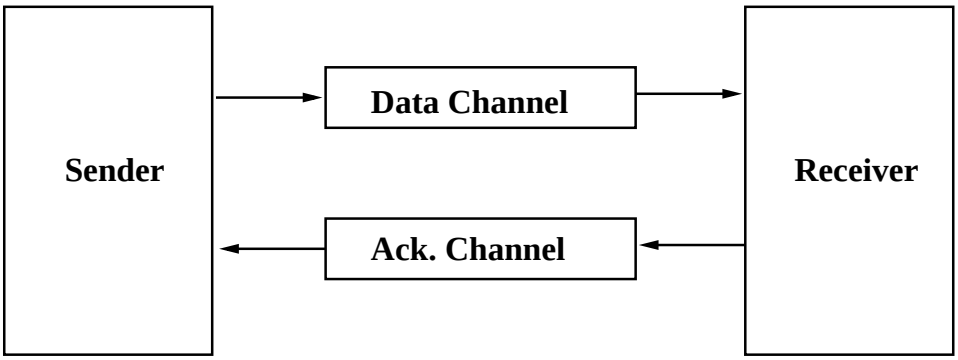


Fig. 5. The Channels

Invariant **inv2_1** and **inv2_2** below correspond to the main properties of s. It states that the value of the counter s is at most one more than that of the counter r. It remains now for us to formalize the channels. For the moment (in this refinement) the data channel contains the counter s of the sender and also the data item d. As the counter s has already been formalized, we only have to define the invariants corresponding to d. This is done in invariants **inv2_3** which states that the transmitted data d is exactly the rth element of the input file f when s is different from r (that is when s is equal to $r+1$ according to invariant **inv2_2**). The Acknowledgment channel just contains the counter r of the receiver.

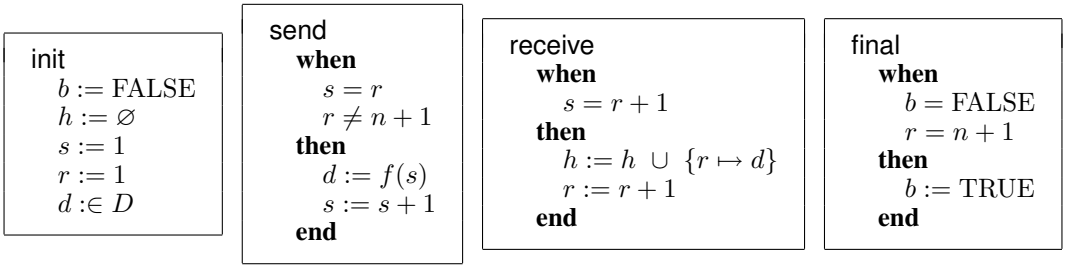
variables:	b
	h
	s
	r
	d

inv2_1: $s \leq n+1$

inv2_2: $s \in r \mathinner{..} r+1$

inv2_3: $s = r+1 \;\Rightarrow\; d = f(r)$

Next are the various events. They encode the informal behavior of the protocol as described above:



Notice our usage of the non-deterministic assignment $d :\in D$ in event init. Non-deterministic assignment will be explained in greater details in section 1.8 of chapter 5. One has just to understand for now that d is assigned *any value* pertaining to the set D.

5.2 Proofs

All proofs are left as exercises to the reader. We encourage you to only take weaker invariants than those proposed in the previous section. More precisely, first drop invariant **inv2_2** and replace invariant **inv2_1** by a weaker one such as $s \in \mathbb{N}$, so that you will be able to see exactly where you need them. Remember that you will have to prove in turn that:

- event init establishes the invariants,
- event receive and final correctly refine their more abstract versions,
- event send refines the implicit event skip,
- event send converges: for this, you will have to exhibit a variant expression,
- taken together, events never deadlock.

Also do not forget that for variables that are the same as those in the abstraction, here b, h, and r, you will have to prove that the actions done on them by old events receive and final are identical.

6 Protocol Third Refinement

In this refinement, we shall give the final implementation of the two-phase handshake protocol. The idea is to observe that it is not necessary to transmit the entire counters s and r on the data and acknowledgment channels. This is so for three reasons: (1) the only tests made on both sites are equality tests ($s = r$ or $s \neq r$, as can be seen in the events defined at the end of section 5.1), (2) the only modifications of the counters are simple incrementations (again, this can be seen in the events defined in the section 5.1), and (3) the difference between s and r is at most 1 (look at invariant **inv2_2**). As a consequence, these equality tests can be performed on the *parities* of these pointers only. These are thus the quantities we are going to transfer between the sites.

6.1 The State

Here are a few obvious definitions concerning the parities of natural numbers. The parity of 0 is 0 and the parity of $x + 1$ is $1 - parity(x)$:

constants: $\ldots$
$parity$

axm3_1: $parity \in \mathbb{N} \rightarrow \{0,1\}$

axm3_2: $parity(0) = 0$

axm3_3: $\forall x \cdot x \in \mathbb{N} \Rightarrow parity(x+1) = 1 - parity(x)$

Notice that in **axm3_3**, we see for the first time a predicate logic formula, which is introduced by the quantifier $\forall$ (to be read "forall").

It is then easy to prove the following result (in section **??**), which we are going to exploit. It says that the comparison of two natural numbers is identical to the comparison of their parities when the difference between these two numbers is at most one:

thm3_1: $\forall x, y \cdot x \in \mathbb{N}$
$y \in \mathbb{N}$
$x \in y \mathinner{..} y+1$
$parity(x) = parity(y)$
$\Rightarrow$
$x = y$

This is a theorem, i.e. a *consequence* to be proved of what has been said elsewhere, namely properties of constants and invariants. We now refine the state and introduce two new variables p and q defined to be the parities of s and r respectively:

variables: $\ldots$
p
q

inv3_1: $p = parity(s)$

inv3_2: $q = parity(r)$

6.2 The Events

The refined events are as follows:

init
$b := \text{FALSE}$
$h := \varnothing$
$s := 1$
$r := 1$
$p := 1$
$q := 1$
$d :\in D$

send
when
$p = q$
$s \neq n + 1$
then
$d := f(s)$
$s := s + 1$
$p := 1 - p$
end

receive
when
$p \neq q$
then
$h := h \cup \{r \mapsto d\}$
$r := r + 1$
$q := 1 - q$
end

final
when
$b = \text{FALSE}$
$r = n + 1$
then
$b := \text{TRUE}$
end

It can be seen that each counter s and r is now modified on one site only. So the only data transmitted from one site to the other are d and p from the sender to the receiver and q from the receiver to the sender. Again, all proofs are left as exercises to the reader.

6.3 Inference Rules for Universally Quantified Predicates

Before proving theorem **thm3_1**, we need clearly some inference rules dealing with universally quantified formulas. As for elementary logic, we need two rules: one for universally quantified assumptions (left rule) and one for a universally quantified goal (right rule). Here are these rules:

$$\frac{\mathbf{H},\ \forall \mathbf{x} \cdot \mathbf{P(x)},\ \mathbf{P(E)} \ \vdash\ \mathbf{Q}}{\mathbf{H},\ \forall \mathbf{x} \cdot \mathbf{P(x)} \ \vdash\ \mathbf{Q}}\ \mathsf{ALL\_L}$$

$$\frac{\mathbf{H} \ \vdash\ \mathbf{P(x)}}{\mathbf{H} \ \vdash\ \forall \mathbf{x} \cdot \mathbf{P(x)}}\ \mathsf{ALL\_R}\ (\mathbf{x} \text{ not free in } \mathbf{H})$$

The first rule (ALL_L) allows us to add another assumption when we have a universally quantified one. This new assumption is obtained by instantiating the quantified variable **x** by any expression **E** in the predicate **P(x)**. The second rule (ALL_R) allows us to remove the "∀" quantifier appearing in the goal. This can be done however only if the quantified variable (here **x**) *does not appear free* in the the set of assumptions **H**: this requirement is called a side condition.

Equipped with the rule introduced in this section, we can now prove theorem **thm3_1**. The proof obligation for this theorem consists in building a sequent with **thm3_1** as a goal and all relevant axioms as assumptions, yielding the following:

$$\begin{array}{l} \ldots \\ parity \ \in\ \mathbb{N} \to \{0,1\} \\ parity(0) = 0 \\ \forall\, x \cdot x \in \mathbb{N} \ \Rightarrow\ parity(x+1) = 1 - parity(x) \\ \vdash \\ \forall\, x, y \cdot \ \begin{array}{l} x \in \mathbb{N} \\ y \in \mathbb{N} \\ x \in y \mathinner{..} y+1 \\ parity(x) = parity(y) \\ \Rightarrow \\ x = y \end{array} \end{array}$$

This proof is left to the reader.

7 Development Revisited

7.1 Motivation and the Introduction of Anticipated Events

In the development we have done so far, we were changing the file variable g of the initial model to another file variable h in the first refinement. Moreover, in order to establish the relationship between both variables (gluing invariants **inv0_2** and **inv0_3**) we had to introduce the boolean variable b which is not really a variable of the protocol. All this seems a bit artificial.

In fact, the reason why we had to change from variable g in the initial model to variable h in the first refinement is purely technical. This is because the new event receive introduced in the first refinement must refine skip (as each new event does). But this new event modifies h: it adds to h an item taken in f. As a consequence, it cannot do that on g: h must be distinct from g.

In order to circumvent this difficulty, we introduce the concept of an *anticipated* event[1]. In the initial model, we introduce the event receive as "anticipated". Its only action is to possibly modify the variable g

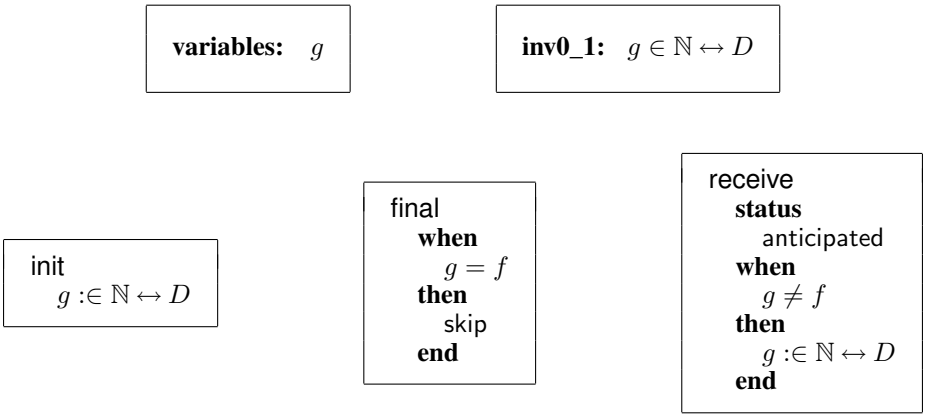
[1] This concept was developed together with D. Cansell and D. Méry.

in a non-deterministic way. More generally, if a new anticipated event is introduced in a refinement (which is not the case here), it needs not decrease a variant, it will do that only when it becomes convergent in a further refinement. However, an anticipated event must not increment the current variant (if any).

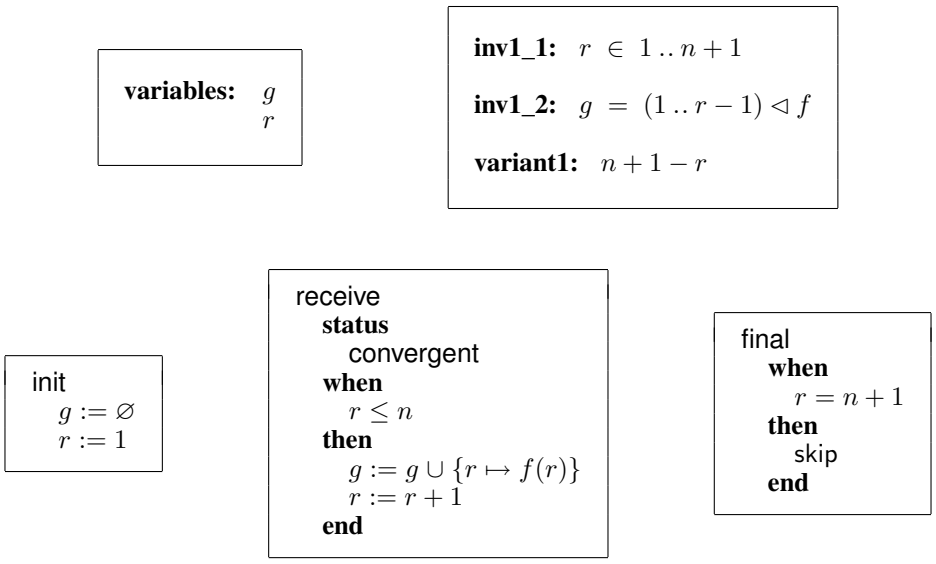
In this new development, event receive becomes convergent in the first refinement. It is exactly as event receive in the previous development except that it works now with variable g. By this, we avoid introducing the artificial file variable h and the boolean variable b.

In the following section, we quickly present this technique applied to our current development. As you will see, it is simpler than the previous one thanks to the introduction of an anticipated event in the initial model.

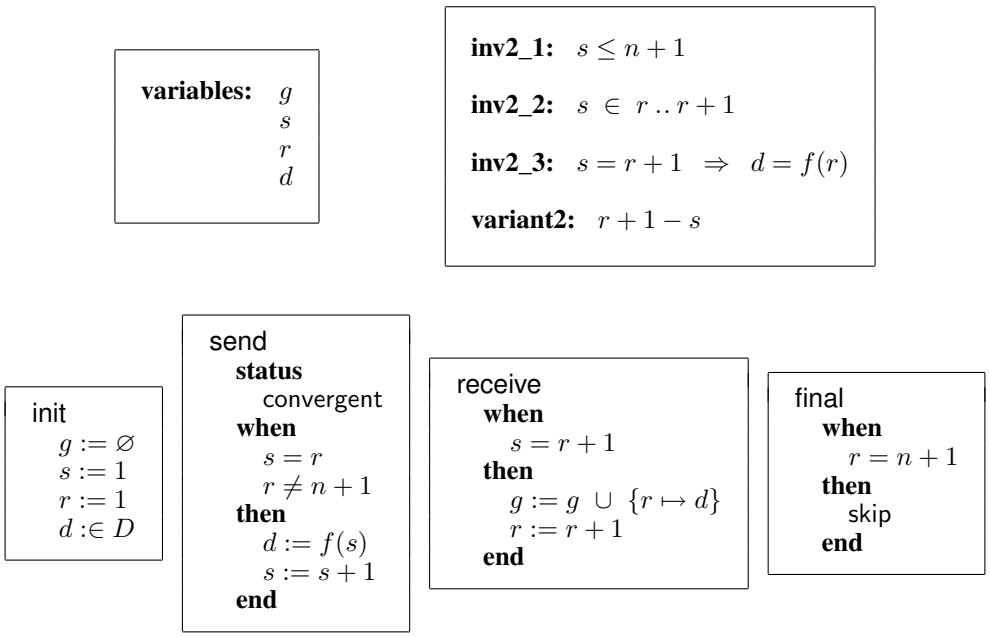
7.2 Initial Model



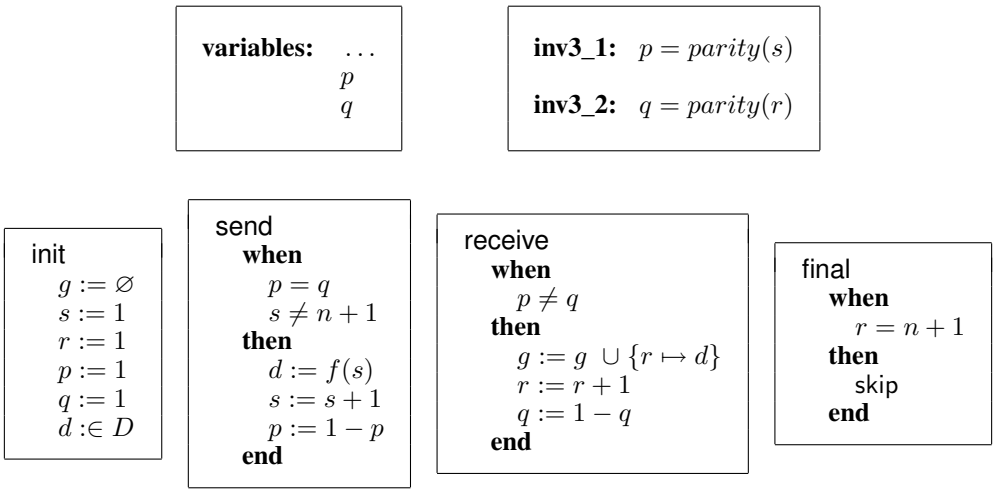
7.3 First Refinement



7.4 Second Refinement



7.5 Third Refinement



References

1. L. Lamport *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers* Addison-Wesley 1999.