

Event Driven Sequential Program Construction

by J.-R. Abrial

January 2001

Event Driven Sequential Program Construction

Introduction

Sequential programs (i.e. loops), when formally constructed, are usually developed gradually by means of a series of more and more refined “sketches” starting with the formal specification and ending up in the final program. Each such sketch is already (although often in a highly non-deterministic form) a unique piece concentrating the final intended program on a single formula. This is precisely that initial “formula”, which is gradually transformed into the final program.

It is argued here that this might *not be the right approach*. After all, in order to prove a large formula, a logician usually breaks it down into various pieces, on which he performs some little work before putting them again together in a final proof. We would like to experiment with such a paradigm and thus possibly decide whether it is applicable to construct programs as well.

A sequential program is essentially made up of a number of individual assignments that are glued together by means of various constructs: conditional statements, sequential compositions, loops. The rôle of such constructs is to explicitly *schedule* these assignments in a proper order so that the execution of the program can achieve its intended goal.

The idea we want to explore here is to completely separate, during the design, these individual assignments from their scheduling. This approach is thus essentially one by which we favor an initial implicit *distribution of computation* over a centralized explicit one. At a certain stage, the “program” is just made of a number of “naked” guarded commands (which we call here “events”), performing some actions under the control of certain guarding conditions. And at this point the synchronization of these events is not our concern. Thinking operationally, it is done *implicitly* by a hidden scheduler, which *may fire* an event once its guard holds.

In the course of the development process, such events might be (data-)refined (guards might thus be strengthened) and other events might be added. As shall be seen below, this has to be done in a certain disciplined manner. What is interesting about this approach is that it gives us full freedom to refine small pieces of the future program, and also to create new ones, *without being disturbed by others* : the program is developed by means of little *independant* parts that so remain until they are eventually put together systematically at the end of the process.

When all the individual pieces are “on the table”, and only then, we start to be interested in their *explicit* scheduling. For this, we apply certain systematic rules whose rôle is to gradually organize these pieces into a single entity forming our final program. The application of these rules has the effect of making the explicit guards pointless. As a matter of fact, at the end of the development, they will have completely disappeared.

The paper is essentially made up of ten illustrating examples. Before engaging in the examples however, we formally define the properties of our event systems and we then propose a number of transformation rules that we are going to apply systematically in the sequel. In

the examples, we shall only make brief mentions of the formal proofs since they have been performed mechanically with **Atelier B**.

The Shape of an Event System

An event system is first made of a state, which is defined by means of *constants* and *variables*. In practical terms, these constants and variables mainly show simple mathematical objects: sets, binary relations, functions, numbers etc. Moreover, they are constrained by some conditions expressing the *invariant properties* of the model.

Besides its state, an event system contains a number of *events* which show the way it may evolve. Each event is composed of a *guard* and an *action*. The guard is the necessary condition under which the event may occur. In other words, once its guard hold, the occurrence of the event may be observed at any time (but it may also never be observed). As soon as the guard does not hold however, the event cannot be observed. Events are *atomic* and when the guards of several events hold simultaneously then *at most one of them* may be observed. The choice of the possibly elected event is non-deterministic.

The action, as its name indicates, determines the way in which the state variables are going to evolve when the event does occur. Practically speaking, an event, named xxx, is presented in the following form:

<pre>xxx ≐ SELECT P(v, w, ...) THEN S(v, w, ...) END</pre>
--

where $P(v, w, \dots)$ is a predicate denoting the guard, and $S(v, w, \dots)$ is the action. The list v, w, \dots denotes some constants and variables of the state. Sometimes, the guard is simply missing (the event may thus take place at any time), it is then written as follows:

<pre>xxx ≐ BEGIN S(v, w, ...) END</pre>

The action presents itself in the form of the simultaneous assignment of certain state variables a, b, \dots to certain expressions E, F, \dots depending upon the state (it is to be noted that those variables which are not mentioned in the list a, b, \dots do not change):

$$a, b, \dots := E, F, \dots$$

When the list is too big, it can be splitted into several assignments as indicated

$$\begin{array}{l} a := E \quad || \\ b := F \quad || \\ \dots \end{array}$$

Sometimes the action may take the form of a, so called, *generalized assignment* of the following form:

$$a, b : R(a, b, a_0, b_0)$$

Here the variables a, b are assigned *any values* satisfying the condition $R(a, b, a_0, b_0)$ where the indexed identifiers denote, by convention, the value of the corresponding variables just before the assignment. Notice that the simple assignment $a, b := E(a, b), F(a, b)$ is equivalent to the following generalized assignment:

$$a, b : (a, b = E(a_0, b_0), F(a_0, b_0))$$

Another more general shape of event is the following:

<pre> xxx ≐ ANY x, y, ... WHERE P(x, y, ..., v, w, ...) THEN S(x, y, ..., v, w, ...) END </pre>

where the identifiers x, y, \dots denotes some constants that are local to the event. In this case, the guard of the event corresponds to the following existential predicate:

$$\exists (x, y, \dots) \cdot P(x, y, \dots, v, w, \dots)$$

In other words, the necessary (but insufficient) condition for the event to take place with the current value of the state variables or constants v, w, \dots of the model, is that it be possible to assign to the local constants x, y, \dots , of the event, some values making the predicate $P(x, y, \dots, v, w, \dots)$ true.

Consistency of an Event System

Once a system is built, one must prove that it is *consistent*. This is made by proving that each event of the system preserves the invariant. More precisely, it must be proved that the action associated to each event modifies the state variables in such a way that the corresponding new invariant predicate holds under the hypothesis of the former invariant predicate and of the guard of the event. For a system with state variable v and invariant $I(v)$, and an event of the form:

<pre> ANY x WHERE P(x, v) THEN v : R(x, v, v_0) END </pre>
--

the statement to be proved is thus

$I(v) \wedge P(x, v) \wedge R(x, v', v) \Rightarrow I(v')$
--

In the case of a simpler assignment of the form $v := E(x, v)$, then $R(x, v', v)$ has to be replaced by $v' = E(x, v)$, yielding

$$I(v) \wedge P(x, v) \Rightarrow I(E(x, v))$$

Refining an Event System

Refining a system consists in refining its state and its events. A concrete system (with regards to a more abstract one) has got a state that should be related to that of the abstraction through a, so-called, *gluing invariant*. Sometimes the concrete state is a simple extension of the abstract one so that the abstraction relation is then just a projection. But in general the abstraction relation can be any relation that should however be defined on all the concrete states.

Each event of the abstract model is refined into a corresponding event of the concrete one. Informally speaking, a concrete event is said to refine its abstraction (1) when the guard of the former is stronger than that of the latter (guard strenghtening), (2) and when the gluing invariant is preserved by the conjoined action of both events.

Suppose we have an abstract with state v and invariant $I(v)$, and also a corresponding concrete model with state w and gluing invariant $J(v, w)$. If an abstract and corresponding concrete events are as follows:

```

ANY  $x$  WHERE
   $P(x, v)$ 
THEN
   $v := E(x, v)$ 
END

```

```

ANY  $y$  WHERE
   $Q(y, w)$ 
THEN
   $w := F(y, w)$ 
END

```

then the statement to prove is

$$I(v) \wedge J(v, w) \wedge Q(y, w) \Rightarrow \exists x \cdot (P(x, v) \wedge J(E(x, v), F(y, w)))$$

This states that for each possible choice of the local variables of the concrete event there is a choice of the local variables of the abstraction that makes the gluing invariant (as modified by *both* events) true: indeed, the concrete event refines its abstraction. As can be seen the concrete guard is stronger than its abstract counterpart. In case of events of a simpler form

```

SELECT
   $P(v)$ 
THEN
   $v := E(v)$ 
END

```

```

SELECT
   $Q(w)$ 
THEN
   $w := F(w)$ 
END

```

then the statement to prove is

$$I(v) \wedge J(v, w) \wedge Q(w) \Rightarrow P(v) \wedge J(E(v), F(w))$$

Properties of the Event System

In this section, we shall express the specific properties that an event system used for program development should satisfy. We shall also fix the *style* we shall adopt in our future program development.

(1) *At the beginning of a development*, our event system is first characterized by some *parameters*, which denotes some constant “input” of the futur program. In other words they will not evolve while the futur program will “run”. The constant *parameters* are declared as follows:

$$parameters \in S_p \wedge Pre_condition$$

where S_p denotes the type of the parameters and *Pre_condition* denotes a predicate defining a certain condition, which the *parameters* should satisfy (besides typing, of course). The initial system also has some variables, called here *results*. These variables are typed with S_r as follow

$$results \in S_r$$

The initial event system contains only one event that can be fired any time: its guard is simply missing (hence it always holds). It involves the *results* and describe the characteristic properties of the outcome of the futur program. Here is the general shape of this event:

$$\begin{array}{l} \text{aprog} \hat{=} \\ \text{BEGIN} \\ \quad results : (results \in S_r \wedge Post_condition) \\ \text{END} \end{array}$$

where S_r denotes the type of the *results* and *Post_condition* denotes the final condition, which the program should satisfy. This condition involves the *parameters* as well as the *results*. The pre- and post- conditions together represents the *specification* of our program.

Notice that the initial system might contain another implicit event called *init*, which initialise *results* to freely “float” within its type as follows¹:

$$\begin{array}{l} \text{init} \hat{=} \\ \text{BEGIN} \\ \quad results : \in S_r \\ \text{END} \end{array}$$

¹ The construct $x : \in s$ is a shorthand for $x : (x \in s)$.

(2) *During the development*, we are performing various refinements of the initial event system. As a consequence, at each stage of the development, the current event system may contain more variables and more events. These events are together constrained by the following laws:

- LAW 1: The disjunction of the guards of the events must always be true (under the current invariant). This is so because we want that our system remains “alive”: in no way should the system be able to deadlock (remember in the initial abstraction the system obviously was alive).
- LAW 2: The new events that are introduced at some point in the development are not allowed to “take control” for ever: this means that these events must always decrease a certain natural number quantity or a certain finite set.

(3) *At the end of the development*, one should obtain again a single event of the following shape:

```
cprog ≐
  BEGIN
    Initialisation ;
    Program
  END
```

where *Initialisation* corresponds to the last version of *init* and *Program* is the last version of *aprogram*.

Some Event System Transformation (Refinement) Rules

Let S be a choice of events (guarded actions). Let \Longrightarrow be the guarding operator². Let \sqcap be the choice operator. We consider the following transformation rules³:

```

 $S \rightsquigarrow S' \sqcap U$ 

Side Conditions:

 $S \sqsubseteq S'$ 
 $\text{skip} \sqsubseteq U$ 
 $I \Rightarrow \text{grd}(S') \vee \text{grd}(U)$ 
 $I \Rightarrow V \in \mathbb{N}$ 
 $I \Rightarrow (V = n \Rightarrow [U](V < n))$ 
```

RULE 1

where \sqsubseteq denotes the data refinement operator. As can be seen, the new event U refines skip ⁴ and decreases some natural number quantity (under the current invariant I). Note that the last three side conditions essentially verify that the above LAW 1 and LAW 2 are

² The construct $P \Longrightarrow S$ is just a shorthand for the more verbose `SELECT P THEN S END`

³ A construct such as $S \sqsubseteq S'$ denotes that S is refined by S'

⁴ `skip`, as its name indicates, does nothing.

satisfied.

The effect of the next law is to put together two events to form an IF statement in an obvious manner:

$$\begin{array}{l}
 (P \wedge Q \Rightarrow S) \parallel (P \wedge \neg Q \Rightarrow T) \parallel U \\
 \sim \\
 (P \Rightarrow \text{IF } Q \text{ THEN } S \text{ ELSE } T \text{ END}) \parallel U
 \end{array}
 \quad \text{RULE 2}$$

Our third law transforms a single event into a WHILE statement. The idea is to force the firing of that event as much as possible by excluding that of other enabled events.

$$\begin{array}{l}
 (P \wedge Q \Rightarrow S) \parallel (P \wedge \neg Q \Rightarrow T) \parallel U \\
 \sim \\
 (P \Rightarrow \text{WHILE } Q \text{ DO } S \text{ END ; } T) \parallel U \\
 \\
 \text{Side Condition:} \\
 \\
 S \text{ and } T \text{ are not guarded} \\
 I \wedge P \wedge Q \Rightarrow [S]P
 \end{array}
 \quad \text{RULE 3}$$

Notice that the loop body S should maintain P invariant (under the guard Q). Two special cases of RULE 3 are quite often encountered. These are those where the predicate P is simply missing (in which case the second side condition holds trivially), or where T reduces to `skip`.

RULE 1 is essentially used in the first half of the development (putting the events “on the table”), while RULE 2 and RULE 3 are used in the second half (putting the events together in order to build a sequential program).

Clearly, other rules could be invented, but for simplicity, and because we shall not use other ones in this paper, we have limited ourselves to these three rules only.

The Technique of Anticipation

In previous section, we have seen how it can be possible to introduce new events in a refinement by means of RULE 1. Such events must refine `skip` and also decrease a certain natural number or finite set expression. The decreasing quantity is defined in terms of some of the new variables that are introduced at the corresponding refinement step.

In order to easily implement this rule, we shall proceed as follows. Suppose that a variable y is normally introduced, together with its typing S_y and possibly some gluing invariant, at refinement step \mathcal{R}_{i+1} . Also suppose that a new event `xxx` is also introduced at this stage. Suppose, finally, that the corresponding variant, which `xxx` is supposed to decrease, is denoted by the natural number expression $V(y)$. The technique of *anticipation* consists in introducing the variable y and also the event `xxx` at the *previous step*, that is at \mathcal{R}_i ,

together with the minimal requirement of maintaining y within its type and decreasing the quantity $V(y)$ (the gluing invariant is *not* introduced at \mathcal{R}_i , still at \mathcal{R}_{i+1}). This can be done as follows:

```

xxx  $\hat{=}$ 
  BEGIN
     $y : (y \in S_y \wedge V(y) < V(y_0))$ 
  END

```

Since y was new at step \mathcal{R}_{i+1} it is, a fortiori, new at step \mathcal{R}_i . The above anticipating event `xxx` trivially refines `skip` and decreases the quantity $V(y)$, so that **RULE 1** is satisfied at step \mathcal{R}_i . The usual refinement rule and proof are thus applied at step \mathcal{R}_{i+1} , guaranteeing that indeed the “implementation” of `xxx` is correct. It is then quite possible that some new variables and events are introduced at step \mathcal{R}_{i+1} in anticipation of step \mathcal{R}_{i+2} , and so on.

In the case where another variable, say x , which is not involved in the variant expression, would normally be introduced at stage \mathcal{R}_{i+1} , it can also be declared in the anticipating step \mathcal{R}_i and mentioned in the anticipating event, yielding:

```

xxx  $\hat{=}$ 
  BEGIN
     $x, y : (x \in S_x \wedge y \in S_y \wedge V(y) < V(y_0))$ 
  END

```

This will often be the case at the first step \mathcal{R}_1 , considered to refine an implicit step \mathcal{R}_0 with no variables and all events defined as `skip`.

Example 1: Searching in an Array

Our intention with this first example is to build a simple search program able to find the index of a sequence whose corresponding value is given (this value being known however to belong to the sequence). Let S be a certain set, and let n , f , and x be three constants, and i be a variable such that the following olds

```

 $n \in \mathbb{N}_1$ 

 $f \in 1..n \rightarrow S$ 

 $x \in \text{ran}(f)$ 

```

```

 $i \in 1..n$ 

```

Let, finally, `aprog` be the following event, which sets the variable i to any index of the domain of f , in such a way that $f(i)$ is exactly x .

```

aprog  $\hat{=}$ 
  BEGIN
     $i : (i \in 1..n \wedge f(i) = x)$ 
  END

```

We also use an anticipating event **progress** which is as follows:

```

progress  $\hat{=}$ 
  BEGIN
     $i : (i \in 1..n \wedge n - i < n - i_0)$ 
  END
  
```

Notice that here there is no anticipating variable. Our goal is to eventually “implement” this event by means of a search loop. In our first and unique refinement, we introduce the following invariant:

```

 $\forall j \cdot (j \in 1..i - 1 \Rightarrow f(j) \neq x)$ 
  
```

This invariant is quite intuitive: it simply expresses that the first $i - 1$ elements of the sequence f have already been explored unsuccessfully. This can be represented pictorially as follow:

1	failure	$i - 1$	i	?	n
---	----------------	---------	-----	---	-----

The success will thus be certainly obtained eventually in the remaining part of the sequence (but where ?) since we know that x belongs to the range of f . Here are the refinement of the events:

<pre> init $\hat{=}$ BEGIN $i := 1$ END </pre>	<pre> aprog $\hat{=}$ SELECT $f(i) = x$ THEN skip END </pre>	<pre> progress $\hat{=}$ SELECT $f(i) \neq x$ THEN $i := i + 1$ END </pre>
---	---	--

The correct data-refinements of **aprog**, **init** and **progress** relative to their abstractions are not difficult to prove. Notice that **aprog** does not compute any more: as a matter of fact, it is just a “witness”, which observes that the computation is complete. By applying now **RULE 3**, we group together events **aprog** and **progress**. Adding **init**, we obtain the following program:

```

cprog  $\hat{=}$ 
  BEGIN
     $i := 1$ ; init
    WHILE  $f(i) \neq x$  DO
       $i := i + 1$  progress
    END
  END
  
```

Example 2: Looking for the Maximum of an Array of Numbers

Our next elementary example consists in looking for the maximum of a non empty sequence of natural numbers. Let n and f be two constants, and m a variable and k an anticipating variable:

$n \in \mathbb{N}_1$ $f \in 1..n \rightarrow \mathbb{N}$	$m \in \text{ran}(f)$ $k \in 1..n$
--	------------------------------------

Let **aprog** be the following event, which sets m to the maximum of the range of f .

```

aprog  $\hat{=}$ 
  BEGIN
     $m : (m \in \text{ran}(f) \wedge \forall x \cdot (x \in 1..n \Rightarrow f(x) \leq m))$ 
  END
    
```

We also have two anticipating events **test_1** and **test_2**:

<pre> test_1 $\hat{=}$ BEGIN $m, k : \left(\begin{array}{l} m \in \text{ran}(f) \quad \wedge \\ k \in 1..n \quad \wedge \\ n - k < n - k_0 \end{array} \right)$ END </pre>	<pre> test_2 $\hat{=}$ BEGIN $m, k : \left(\begin{array}{l} m \in \text{ran}(f) \quad \wedge \\ k \in 1..n \quad \wedge \\ n - k < n - k_0 \end{array} \right)$ END </pre>
---	---

Both anticipating events decreases the variant $n - k$. Also notice the presence of variable m , which freely floats within its type. Our goal is to implement **aprog** by means of an exhaustive analysis of the sequence f . For this, we introduce the following invariant:

$$\forall x \cdot (x \in 1..k \Rightarrow f(x) \leq m)$$

This invariant expresses that m is indeed the maximum of the first k elements of the sequence f . The refinement leads to the following events:

<pre> init $\hat{=}$ BEGIN $k := 1 \quad$ $m := f(1)$ END </pre>	<pre> aprog $\hat{=}$ SELECT $k = n$ THEN skip END </pre>	<pre> test_1 $\hat{=}$ SELECT $k \neq n \quad \wedge$ $f(k+1) \leq m$ THEN $k := k + 1$ END </pre>	<pre> test_2 $\hat{=}$ SELECT $k \neq n \quad \wedge$ $f(k+1) > m$ THEN $k := k + 1 \quad$ $m := f(k+1)$ END </pre>
---	--	---	---

The refinements are not difficult to prove. Notice again that the event **aprog** does not calculate any more. Our next step is to put together events **test_1** and **test_2** by applying **RULE 2**. Applying then **RULE 3** with event **aprog** and adding **init**, we obtain:

```

cprog ≐
BEGIN
  [k, m := 1, f(1)];           init
  WHILE k ≠ n DO
    IF f(k + 1) ≤ m THEN
      [k := k + 1]             test_1
    ELSE
      [k, m := k + 1, f(k + 1)] test_2
    END
  END
END
END

```

Example 3: Searching in a Matrix

This example is very close to **Example 1**. Rather than searching in a sequence, our intention is now to search in a matrix. We would like to see whether this will significantly modify the structure of our final program. Given a set S , let m, n, f , and x be four constants, and i and j be two variables such that the following holds:

$m \in \mathbb{N}_1$ $n \in \mathbb{N}_1$ $f \in (1..m) \times (1..n) \rightarrow S$ $x \in \text{ran}(f)$	$i \in 1..m$ $j \in 1..n$
--	---------------------------

Let, finally, **aprog** be the following event, which sets the variables i and j to any indices of the domain of f , in such a way that $f(i, j)$ is exactly x .

```

aprog ≐
BEGIN
  i, j : (i ∈ 1..m ∧ j ∈ 1..n ∧ f(i, j) = x)
END

```

We also use two anticipating events **progress_i** and **progress_j**, which are as follows:

```

progress_i ≐
BEGIN
  i, j : (
    i ∈ 1..m ∧
    j ∈ 1..n ∧
    n × (m - i) - j < n × (m - i_0) - j_0
  )
END

```

```

progress_j ≐
BEGIN
  i, j : ( i ∈ 1..m  ∧
           j ∈ 1..n  ∧
           n × (m - i) - j < n × (m - i_0) - j_0 )
END

```

We now introduce the following invariant:

```

∀(u, v) · (u ∈ 1..i - 1 ∧ v ∈ 1..n ⇒ x ≠ (f(u, v)))
∀v · (v ∈ 1..j - 1 ⇒ x ≠ (f(i, v)))

```

This invariant is also quite intuitive: it simply expresses that the first $i - 1$ rows of the matrix f have already been explored unsuccessfully as well as the first $j - 1$ columns of row i . The events are refined as follows:

<pre> init ≐ BEGIN i, j := 1, 1 END </pre>	<pre> aprog ≐ SELECT f(i, j) = x THEN skip END </pre>	<pre> progress_i ≐ SELECT j = n ∧ f(i, j) ≠ x THEN i, j := i + 1, 1 END </pre>	<pre> progress_j ≐ SELECT j ≠ n ∧ f(i, j) ≠ x THEN j := j + 1 END </pre>
--	---	---	---

The data refinements of events `aprog`, `init`, `progress_i` and `progress_j` are easily proved to be correct. By applying **RULE 2**, we group together events `progress_i` and `progress_j`, and we then apply **RULE 3** with `aprog`. By adding `init`, we obtain the following:

```

cprog ≐
BEGIN
  [i, j := 1, 1];           init
  WHILE f(i, j) ≠ x DO
    IF j = n THEN
      [i, j := i + 1, 1]   progress_i
    ELSE
      [j := j + 1]         progress_j
    END
  END
END

```

Notice that, *quite unexpectedly*, we end up with a single loop. In fact, the structure of the final program is very close to that obtained at the end of **Example 1**. In both cases the guard of the loop denotes the condition under which the result is obtained, and the body of the loop “increments” the “index”.

Example 4: Testing for a Row Filled in with 0's in a Number Matrix

As in previous example, we shall now consider a natural number matrix. Our intention is to develop a program searching for the possibility for that matrix to have a row entirely filled in with zeros. In previous examples we were certain to end up “before” the end of the exhaustive search. That is, we did not test in any way that the indices i and j were reaching their natural end. This was so because it was known in advance that the search was certainly successful.

Here the situation is different in that we may terminate either before the natural end (in case of success, when there exists a row entirely filled in with zeros) or, alternatively, at the natural end (in case of failure, when there does not exist such a row). We would like to see whether this makes the final program look differently. We have thus three constants m , n and f , and a variable r , together with two anticipating variables typed as follows:

$m \in \mathbb{N}_1$	$r \in \{success, failure\}$
$n \in \mathbb{N}_1$	$k \in 1..m$
$f \in (1..m) \times (1..n) \rightarrow \mathbb{N}$	$l \in 1..n$

Let `aprog_0` and `aprog_1` be the following events, which calculates r . Notice that we have *two* `aprog`s here instead of one as advocated above. This is because there is a natural breakdown into two parts due to the fact that there are two result values (*success* and *failure*) discriminated according to a certain predicate (we could have used a conditional statement instead but this would have been heavier at this stage). As can be seen, the variable r is equal to *success* if and only if there exists a row i in the matrix f that is entirely filled in with zeros:

```
aprog_0 ≐
SELECT
  ∃ i · ( i ∈ 1..m ∧ ∀ j · ( j ∈ 1..n ⇒ f(i,j) = 0 ) )
THEN
  r := success
END
```

```
aprog_1 ≐
SELECT
  ¬ ∃ i · ( i ∈ 1..m ∧ ∀ j · ( j ∈ 1..n ⇒ f(i,j) = 0 ) )
THEN
  r := failure
END
```

We also have two anticipating events `progress_k` and `progress_l`:

```

progress_k ≐
BEGIN
  k, l : ( k ∈ 1..m  ∧
           l ∈ 1..n  ∧
           n × (m - k) - l < n × (m - k_0) - l_0 )
END

```

```

progress_l ≐
BEGIN
  k, l : ( k ∈ 1..m  ∧
           l ∈ 1..n  ∧
           n × (m - k) - l < n × (m - k_0) - l_0 )
END

```

We now introduce a new variable d , together with the following invariant:

```

d ∈ {working, finished}

∀ i · ( i ∈ 1..k - 1 ⇒ ¬∀ j · ( j ∈ 1..n ⇒ f(i, j) = 0 ) )

∀ j · ( j ∈ 1..l - 1 ⇒ f(k, j) = 0 )

```

This invariant says (1) that each of the $k - 1$ first rows of f is not entirely filled in with zeros (in other words, we have a failure on all these rows: this is indeed why row k is considered), and, (2) that, on the contrary, the first $l - 1$ elements of row k are all equal to zero (that is, success so far on this row: this is why we shall continue exploring it). Here are the data refinement of the events:

```

init ≐
BEGIN
  k, l, d := 1, 1, working
END

```

```

aprog_0 ≐
SELECT
  d = working  ∧
  f(k, l) = 0  ∧
  l = n
THEN
  r, d := success, finished
END

```

```

aprog_1 ≐
SELECT
  d = working ∧
  f(k, l) ≠ 0 ∧
  k = m
THEN
  r, d := failure, finished
END

```

```

progress_l ≐
SELECT
  d = working ∧
  f(k, l) = 0 ∧
  l ≠ n
THEN
  l := l + 1
END

```

```

progress_k ≐
SELECT
  d = working ∧
  f(k, l) ≠ 0 ∧
  k ≠ m
THEN
  k, l := k + 1, 1
END

```

We now apply RULE 2 three times in a straightforward way: first on events `aprog_1` and `progress_k`, then on events `aprog_0` and `progress_l`, and, finally, on the results of these two applications. By applying now RULE 3⁵, and adding `init`, we obtain our final program `cprog`. Notice the symmetry of the loop body.

```

cprog ≐
BEGIN
  [k, l, d := 1, 1, working];           init
  WHILE d = working DO
    IF f(k, l) = 0 THEN
      IF l = n THEN
        [r, d := success, finished]     aprog_0
      ELSE
        [l := l + 1]                     progress_l
      END
    ELSE
      IF k = m THEN
        [r, d := failure, finished]     aprog_1
      ELSE
        [k, l := k + 1, 1]              progress_k
      END
    END
  END
END
END
END

```

Example 5: Finding a Common Number of two Sets

The example we shall consider now is a bit different from the previous ones. Here, data refinement will play a more important rôle as we shall start from more abstract data structures (sets) in comparison with those encountered above (sequence, matrices). Our aim is to construct an algorithm for finding any element belonging to two finite sets of natural numbers, whose intersection is guaranteed to be not empty. We have two constants a and b , and a variable x , together with two anticipating variable a' and b' typed as follows⁶:

$a \in \mathbb{F}(\mathbb{N})$	$x \in \mathbb{N}$
$b \in \mathbb{F}(\mathbb{N})$	$a' \in \mathbb{F}(\mathbb{N})$
$a \cap b \neq \emptyset$	$b' \in \mathbb{F}(\mathbb{N})$

We now define the abstract event `aprog`

```

aprog ≐
BEGIN
  x := a ∩ b
END

```

⁵ The reader might discover that a little event is missing here. Guess which.

⁶ The construct $\mathbb{F}(E)$ denotes the set of finite subsets of a set E .

Next are the two anticipating events:

$\text{rmv_a} \hat{=} \\ \text{BEGIN} \\ a', b' : \left(\begin{array}{l} a' \in \mathbb{F}(\mathbb{N}) \quad \wedge \\ b' \in \mathbb{F}(\mathbb{N}) \quad \wedge \\ a' \cup b' \subset a'_0 \cup b'_0 \end{array} \right) \\ \text{END}$

$\text{rmv_b} \hat{=} \\ \text{BEGIN} \\ a', b' : \left(\begin{array}{l} a' \in \mathbb{F}(\mathbb{N}) \quad \wedge \\ b' \in \mathbb{F}(\mathbb{N}) \quad \wedge \\ a' \cup b' \subset a'_0 \cup b'_0 \end{array} \right) \\ \text{END}$

Notice that the variant quantity $a' \cup b'$ is not a natural number, but a finite set. We require that it is strictly decreasing. The idea of the next step is to maintain the value of the intersection of a' and b' to be equal to that of the intersection of a and b , and to gradually remove some elements of the sets a' and b' while maintaining the following invariant:

$a' \cap b' = a \cap b$

Here are the data refinements of the events :

$\text{init} \hat{=} \\ \text{BEGIN} \\ a', b' := a, b \\ \text{END}$	$\text{aprog} \hat{=} \\ \text{BEGIN} \\ x := a' \cap b' \\ \text{END}$	$\text{rmv_a} \hat{=} \\ \text{ANY } y \text{ WHERE} \\ y \in a' - b' \\ \text{THEN} \\ a' := a' - \{y\} \\ \text{END}$	$\text{rmv_b} \hat{=} \\ \text{ANY } y \text{ WHERE} \\ y \in b' - a' \\ \text{THEN} \\ b' := b' - \{y\} \\ \text{END}$
---	---	--	--

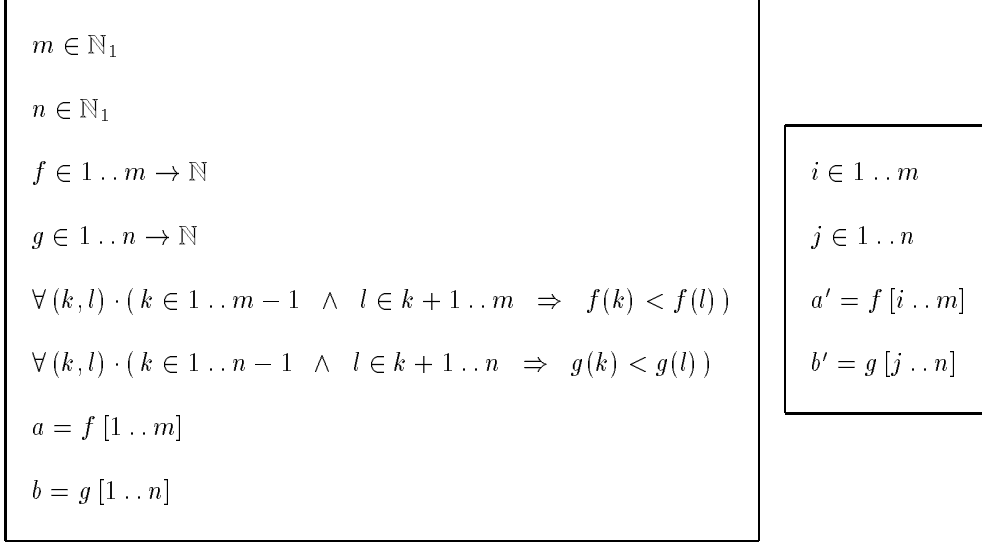
Notice the non-deterministic choice of an element y in these events. We now make the two previous events more deterministic by choosing y to be the minimum of the corresponding set. Event init is left unchanged, whereas event aprog , rmv_a , and rmv_b are made completely deterministic.

$\text{aprog} \hat{=} \\ \text{SELECT} \\ \min(a') \in b' \\ \text{THEN} \\ x := \min(a') \\ \text{END}$	$\text{rmv_a} \hat{=} \\ \text{SELECT} \\ \min(a') \notin b' \\ \text{THEN} \\ a' := a' - \{\min(a')\} \\ \text{END}$	$\text{rmv_b} \hat{=} \\ \text{SELECT} \\ \min(b') \notin a' \\ \text{THEN} \\ b' := b' - \{\min(b')\} \\ \text{END}$
--	--	--

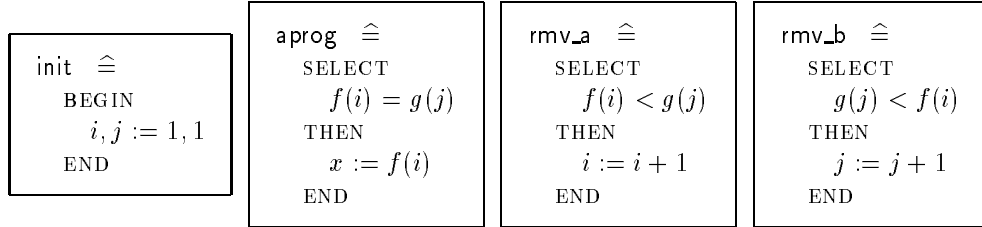
The membership tests of each minimum in the guards of these events can be realized by means of a comparison with the other minimum, yielding

$\text{aprog} \hat{=} \\ \text{SELECT} \\ \min(a') = \min(b') \\ \text{THEN} \\ x := \min(a') \\ \text{END}$	$\text{rmv_a} \hat{=} \\ \text{SELECT} \\ \min(a') < \min(b') \\ \text{THEN} \\ a' := a' - \{\min(a')\} \\ \text{END}$	$\text{rmv_b} \hat{=} \\ \text{SELECT} \\ \min(b') < \min(a') \\ \text{THEN} \\ b' := b' - \{\min(b')\} \\ \text{END}$
--	---	---

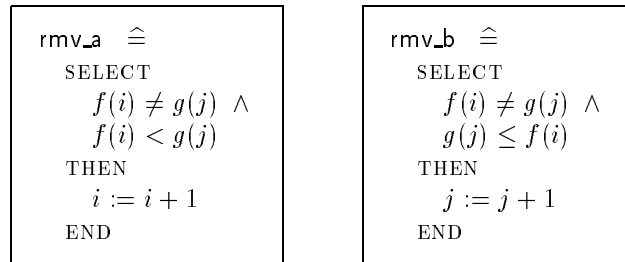
Clearly the guards are strengthened. We now decide to implement the constant sets a and b by means of two constant *ascending and injective* sequences f and g . More precisely, the sets a and b are equal to the range of these sequences. The size of these sequences are m and n respectively. We also introduce two variables i and j in such a way that a' and b' are respectively $f[i..m]$ and $g[j..n]$ (this is the gluing invariant). This implies immediately that $\min(a')$ is equal to $f(i)$ and $\min(b')$ is equal to $g(j)$. It also implies that removing $\min(a')$ from a' or $\min(b')$ from b' just corresponds to incrementing i or j .



We obtain the following refinements



We can rearrange equivalently the guards of the last two events as follows:



Putting the last two together by means of RULE 2, applying then RULE 3 and adding `init` lead to our final program

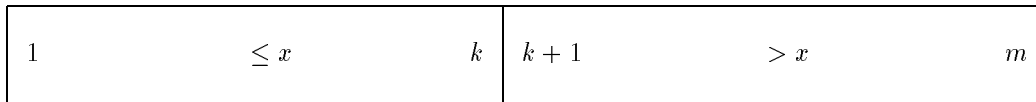
```

cprog  $\hat{=}$ 
BEGIN
   $i, j := 1, 1$ ;           init
  WHILE  $f(i) \neq g(j)$  DO
    IF  $f(i) < g(j)$  THEN
       $i := i + 1$          rmv_a
    ELSE
       $j := j + 1$          rmv_b
    END
  END
END ;
 $x := f(i)$              aprog
END

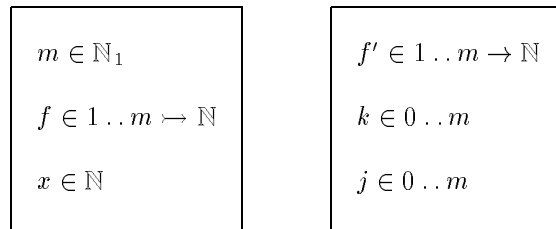
```

Example 6: Array Partitioning

The problem we study now is a variant of the well known partitioning problem used in Quicksort. Let f be a sequence of m natural numbers (supposed to be distinct for simplification). Let x be a natural number. We would like to transform f in another sequence f' with exactly the same elements as the initial f , in such a way that there exists a number k ranging from 0 to m such that all elements in $f'[1..k]$ are smaller than or equal to x while all elements in $f'[k+1..m]$ are strictly greater than x . This final result is thus shown in the following diagram:



Note that in case all elements of f are all greater than x , then k should be equal to 0. And in case all elements are smaller than or equal to x , then k should be equal to m . We have three constants m , f and x , and two variables f' and k , together with an anticipating variable j with the following invariant:



We have the event **aprog** defined as follows

```

aprog  $\hat{=}$ 
BEGIN
   $k, f' : \left( \begin{array}{l} k \in 0..m \\ f' \in 1..m \mapsto \mathbb{N} \\ \text{ran}(f') = \text{ran}(f) \\ \forall l \cdot (l \in 1..k \Rightarrow f'(l) \leq x) \\ \forall l \cdot (l \in k+1..m \Rightarrow f'(l) > x) \end{array} \right)$ 
END

```

We have the following anticipating events:

```

progress_1 ≐
BEGIN
  k, f', j :  $\left( \begin{array}{l} k \in 0..m \\ f' \in 1..m \rightarrow \mathbb{N} \\ j \in 0..m \\ m - j < m - j_0 \end{array} \right)$ 
END

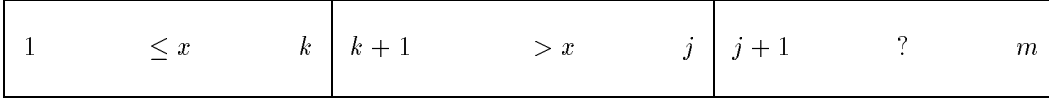
```

```

progress_2 ≐
BEGIN
  k, f', j :  $\left( \begin{array}{l} k \in 0..m \\ f' \in 1..m \rightarrow \mathbb{N} \\ j \in 0..m \\ m - j < m - j_0 \end{array} \right)$ 
END

```

Our next step is to introduce an invariant in such a way that k and j partition the sequence f' as indicated by the following diagram:



The idea is then to possibly increment j alone or both k and j while maintaining the corresponding invariant. The process is completed when j is equal to m . More formally, this yields:

```

k ≤ j
∀l · (l ∈ 1..k ⇒ f'(l) ≤ x)
∀l · (l ∈ k + 1..j ⇒ f'(l) > x)

```

Here are the refinements of the events.

```

init ≐
BEGIN
  j, k := 0, 0
END

```

```

aprog ≐
SELECT
  j = m
THEN
  skip
END

```

```

progress_1 ≐
SELECT
  j ≠ m ∧
  f'(j + 1) > x
THEN
  j := j + 1
END

```

```

progress_2 ≐
SELECT
  j ≠ m ∧
  f'(j + 1) ≤ x
THEN
  k, j := k + 1, j + 1 ||
  IF k ≠ j THEN
    f' := f' ◁ {k + 1 ↦ f'(j + 1)}
    ◁ {j + 1 ↦ f'(k + 1)}
  END
END

```

By putting together events `progress_1` and `progress_2` (using RULE 2), applying then RULE 3 with `aprog` and adding `init` lead to the following program

```

cprog ≐
BEGIN
   $k, j := 0, 0$ ;                               init
  WHILE  $j \neq m$  DO
    IF  $f'(j+1) > x$  THEN
       $j := j + 1$                                   progress_1
    ELSE
       $k, j := k + 1, j + 1$  ||
      IF  $k \neq j$  THEN
         $f' := f' \triangleleft \{k + 1 \mapsto f'(j + 1)\}$  progress_2
           $\triangleleft \{j + 1 \mapsto f'(k + 1)\}$ 
      END
    END
  END
END
END

```

Example 7: In Place Reversing of an Array

Our next example is the classical “in place” reversing of an array. Given a set S , we have two constants m and f , and a variable f' , together with an anticipating variable j

$m \in \mathbb{N}_1$ $f \in 1..m \rightarrow S$	$f' \in 1..m \rightarrow S$ $j \in 1..m$
--	---

Our event `aprog` is as follows:

```

aprog ≐
BEGIN
   $f' : \left( f' \in 1..m \rightarrow S \wedge \right.$ 
     $\left. \forall k \cdot (k \in 1..m \Rightarrow f'(k) = f(m - k + 1)) \right)$ 
END

```

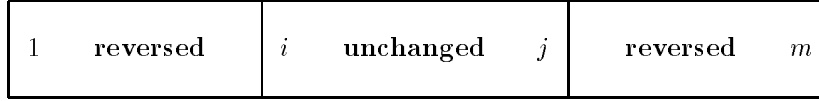
The anticipating event is as follows:

```

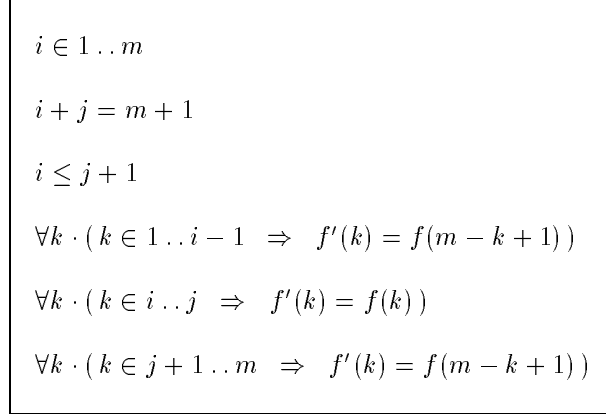
swap ≐
BEGIN
   $f', j : \left( f' \in 1..m \rightarrow S \wedge \right.$ 
     $\left. j \in 1..m \wedge j < j_0 \right)$ 
END

```

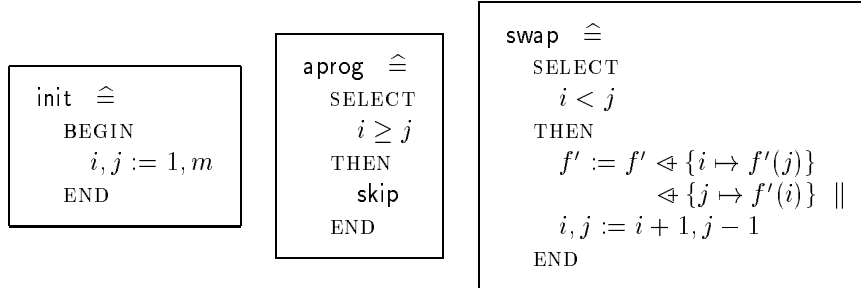
The next transformation consists in introducing another index i starting at 1 (j starts at m). The indices i and j move towards each other. The sequence f' is gradually reversed by swapping elements $f'(i)$ and $f'(j)$ while, of course, i is strictly smaller than j . This is done in the event **swap**. In this way, the sub-sequences of f' ranging from 1 to $i - 1$ and from $j + 1$ to m respectively have all their elements reversed with regard to the original sequence f . And the middle part is still unchanged with regards to f . This can be illustrated in the following diagram:



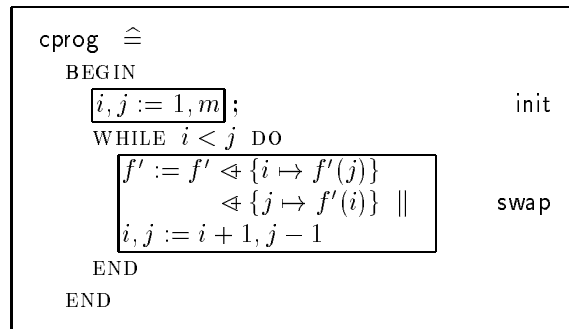
Notice that the quantity $i + j$ is always equal to $m + 1$. At the end of the process either i is equal to j when m is odd, or i is equal to $j + 1$ when m is even (but, in both case, we have $i \geq j$). Here is the new invariant



Here are the refined events:



Applying the usual rules to **aprogram** and **swap**, we obtain the following final program **cprog**:



Example 8: In Place Reversing of a Linear Chain

So far all our examples were dealing with arrays and corresponding indices. As a consequence some of the proofs heavily relied on (elementary) arithmetic properties. In this example, we experiment our approach on a data structure that deals with “pointers”.

The problem we shall tackle is very classical and simple: we just want to reverse a linear chain. Notice that to simplify matters the chain we consider is made of pointers only (it has no information “field”).

Each element in the chain “points” to its immediate successor. The chain, c , starts with an element called f (for “first”) and ends up with an element called l (for “last”). Notice that f and l might be equal, so that the chain has (at least) one element. By convention, l points to a “special element” called nil . All this can be represented in the following diagram:



Before engaging in our problem (the in place reversing of this linear chain), we first have to formalize what we have just introduced. The elements participating in the chain are supposed to form a certain set S . The constants f , l and nil are members of S , with f and l distinct from nil . The chain itself is defined by means of a function c , which is a bijection from $S - nil$ to $S - f$. Notice that $c(l)$ is nil . But, clearly, this is not sufficient: we have to express somehow that the chain *has no gap*, in other words that we can continuously “travel” from f to nil by following the pointers. This is the rôle of the quantification:

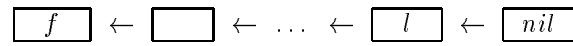
$$\begin{array}{l} f \in S \wedge l \in S \wedge nil \in S \wedge f \neq nil \wedge l \neq nil \\ c \in S - \{nil\} \mapsto S - \{f\} \\ c(l) = nil \\ \forall A \cdot (A \subseteq S \wedge nil \in A \wedge c^{-1}[A] \subseteq A \Rightarrow A = S) \end{array}$$

The intuition behind the above quantification is this. Think of A being the set of members of S from which it is possible to reach nil by following the chain c . This set must be equal to S and here are the characteristic properties of this set: clearly nil itself is in A and if some point p is in A then so is the point $c^{-1}(p)$ when it exists, thus $c^{-1}[A] \subseteq A$. The quantification says that provided a set A has got these properties than it *must* be equal to S . This quantified predicate is an induction rule that will be used for proving properties of the elements of the set S .

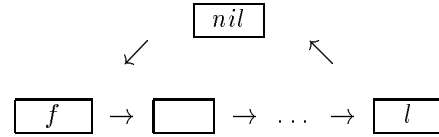
We would like to reverse the chain in such a way that l becomes the first element of the result and f its last element (pointing, of course, to nil). Thus the transformed chain should look like this:



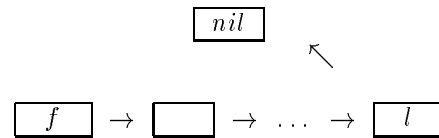
Clearly it would have been simpler to transform our initial list into the following one:



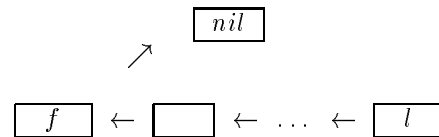
because then the specification of our problem would have been straightforward : the final result would have been the converse of c . Unfortunately, the problem is different, we have thus to find a better abstraction. The idea then is to consider that nil might point to f as indicated in the following diagram, representing a “circular chain” (a cycle) :



This is, of course, different from our initial chain, but this is just an abstraction. In fact, our initial chain is as follows (it is obtainable from the cycle by removing the link $nil \mapsto f$)



and our transformed chain is as follows (likewise it is obtainable from the converse of the cycle by removing the link $nil \mapsto l$):



We now formalize the constant cycle h , obtained from c , as follows:

$$\boxed{\begin{array}{l} h \in S \mapsto S \\ h = c \cup \{nil \mapsto f\} \end{array}}$$

Notice that the bijection property of h could easily be deduced from the properties of c and the definition of h . But what is less obvious to prove is the fact that indeed h denotes a cycle. So our next problem is to define the characteristic property of such cycles. If a subset A of S forms a cycle under h then, clearly, the image of A under h^{-1} is exactly A (a property also shared by the empty set). As a consequence the only possibility for h to have a single *outermost* cycle is that the subsets A of S having that property (that is, $h^{-1}[A] = A$) are just S itself and, of course, the empty set. Our circular property is thus formalized as follows:

$$\boxed{\forall A \cdot (A \subseteq S \wedge h^{-1}[A] = A \Rightarrow A = S \vee A = \emptyset)} \quad (1)$$

This property can be formally proven by using the induction rule defined on c , and the definition of h in terms of c (hint: do a proof by case concerning the membership of nil to A). Our result variable r is just typed as a relation from S to S . We also have an anticipating variable T , which is a subset of S

$$\begin{array}{l} r \in S \leftrightarrow S \\ T \subseteq S \end{array}$$

We have then the usual event **aprog**. Notice that the result r in **aprog** is obtained, as announced above, by taking the converse of the cycle h with the link $nil \mapsto l$ removed.

$$\begin{array}{l} \text{aprog} \hat{=} \\ \text{BEGIN} \\ r := h^{-1} \triangleright \{l\} \\ \text{END} \end{array}$$

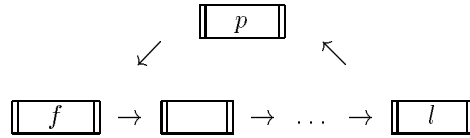
The anticipating event **progress** decreases the set T

$$\begin{array}{l} \text{progress} \hat{=} \\ \text{BEGIN} \\ r, T : (r \in S \leftrightarrow S \wedge T \subseteq S \wedge T \subset T_0) \\ \text{END} \end{array}$$

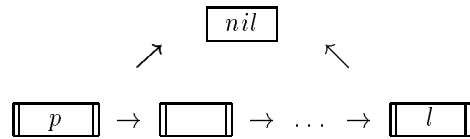
The next step proceeds with one more variables : p is a pointer moving through the cycle (it starts at nil). Notice that T is the subset of S representing the various elements whose pointer still has to be reversed. Here is the declaration of p :

$$p \in S$$

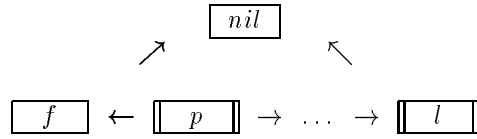
The dynamic situation can be depicted on the following diagrams where a double box means membership to T . At the beginning, T covers the entire set S , p is equal to nil , and r is equal to c .



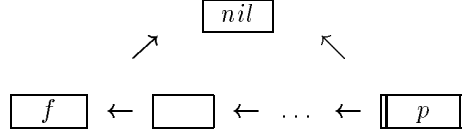
At each step, p is removed from T and moved to $h(p)$. Moreover, the link from p to $h(p)$ is changed to a link from $h(p)$ to p . So, after the first step we have the following situation:



After the second step, we have thus the following:



The process does stop when p is equal to l . We can figure out on the diagram that, in this case, T is also exactly equal to $\{l\}$.



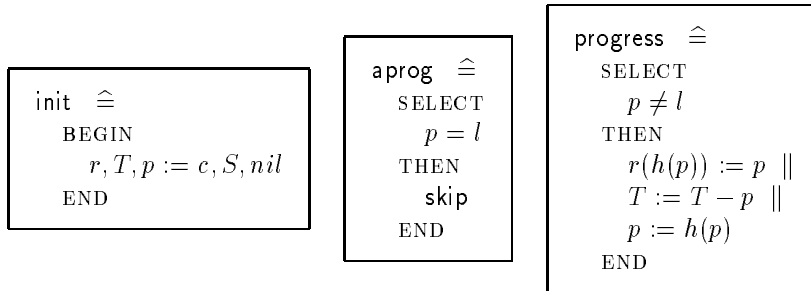
From this diagrammatic description, we can “see” a number of invariant properties. First, p and l are always members of T . Second, the image of the set $T - \{l\}$ under h is exactly $T - \{p\}$ (a “fixpoint” is thus obtained when p is equal to l). In other words, p is the “smallest” element of T with respect to the relation h . Third, the variable r is exactly c overridden by h^{-1} with links ending in T removed. Formally

$$\boxed{
 \begin{array}{l}
 p \in T \\
 l \in T \\
 h[T - \{l\}] = T - \{p\} \quad \mathbf{(2)} \\
 r = c \Leftarrow (h^{-1} \triangleright T)
 \end{array}
 }$$

The property of p (that is, $p \in T$) implies that when T is equal to $\{l\}$, then p clearly is equal to l . Conversely, when p is equal to l then we have $h[T - \{l\}] = T - \{l\}$ according to **(2)**, thus $T - \{l\}$ is empty according to **(1)** (since $T - \{l\}$ is clearly not equal to S). As a consequence, the set T is also equal to $\{l\}$ in that case since l always belongs to T ($l \in T$ being an invariant). We have thus informally proved the following equivalence:

$$\boxed{T = \{l\} \Leftrightarrow p = l}$$

We now refine `init` and `aprog` and we introduce the new event `progress`.



Notice that the function h is still used in the event **progress**, it will be eliminated in the next refinement. The proofs are not difficult although a bit tedious. Notice that **progress** decreases the set T since p always belongs to T . Our next step proceeds by throwing the set T , which just appears thus to be an *abstract artifact* needed to perform the various proofs. We also introduce an extra pointer q , which is equal to $h(p)$.

$$q = h(p)$$

The test $p = l$ is then clearly equivalent to $q = nil$ since nil is equal, by definition, to $h(l)$ and since h is injective (that is, we have indeed $p = l \Leftrightarrow h(p) = h(l)$). Formally

$$p = l \Leftrightarrow q = nil$$

Next are the three refinements of our events:

```

init ≐
BEGIN
  r, p, q := c, nil, f
END

```

```

aprog ≐
SELECT
  q = nil
THEN
  skip
END

```

```

progress ≐
SELECT
  q ≠ nil
THEN
  r(q) := p ||
  p := q ||
  q := r(q)
END

```

Notice that in event **progress**, the variable q is assigned to $r(q)$, not $h(q)$ as one would expect according to the definition of q . For the refinement to be correct it must then be shown that $r(q)$ and $h(q)$ are equal (intuitively, this is because q is one step behind p , hence that part of r dealing with q has not yet been “reversed”, it is thus still as in h). Applying **RULE 3** to **aprog** and **progress**, and adding **init** leads to the final program:

```

cprog ≐
BEGIN
  r, p, q := c, nil, f;      init
  WHILE q ≠ nil DO
    r(q) := p ||
    p := q ||
    q := r(q)              progress
  END
END

```

Example 9: Sorting

We are not going to develop a very smart sorting algorithm in this example. Our intention is only to use sorting as an opportunity to develop a little program containing an *embedded*

loop. We want to figure out whether this embedding would come *naturally*.

We have two constants: m , which is a positive natural number, and f , which is a total injective function from $1..m$ to the natural numbers. We have a result variable g and an anticipating variable k .

$m \in \mathbb{N}_1$ $f \in 1..m \mapsto \mathbb{N}$	$g \in 1..m \rightarrow \mathbb{N}$ $k \in 1..m$
---	---

The event **aprog** is as follows:

```

aprog  $\hat{=}$ 
  BEGIN
     $g : \left( \begin{array}{l} g \in 1..m \rightarrow \mathbb{N} \wedge \\ \forall (i, j) \cdot \left( \begin{array}{l} i \in 1..m-1 \wedge \\ j \in i+1..m \\ \Rightarrow \\ g(i) < g(j) \end{array} \right) \wedge \\ \text{ran}(g) = \text{ran}(f) \end{array} \right)$ 
  END

```

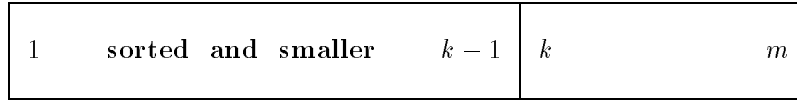
The non-deterministic condition in event **aprog** stipulates that the resulting g is sorted in an ascending way and that it exactly has the same elements as the original f . The anticipating event **progr** decreases the quantity $m - k$ and keeps g floating:

```

progr  $\hat{=}$ 
  BEGIN
     $g, k : \left( \begin{array}{l} g \in 1..m \rightarrow \mathbb{N} \wedge \\ k \in 1..m \wedge \\ m - k < m - k_0 \end{array} \right)$ 
  END

```

In our first refinement, we suppose that the elements of the sub-part of g ranging from 1 to $k - 1$ are all sorted and also smaller than the elements lying in the other sub-part, namely those ranging from k to m . This can be illustrated in the following diagram



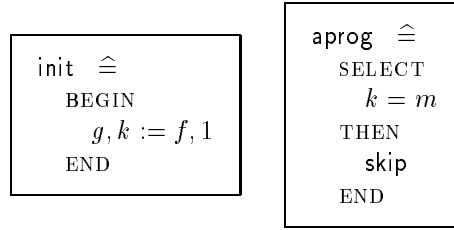
The formal invariant is as follows. We also have an anticipating variable j for next step.

```

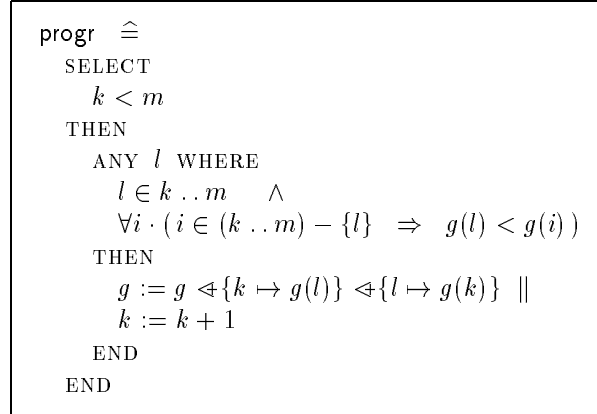
 $\forall (i, j) \cdot (i \in 1..k-1 \wedge j \in i+1..m \Rightarrow g(i) < g(j))$ 
 $j \in 1..m$ 

```

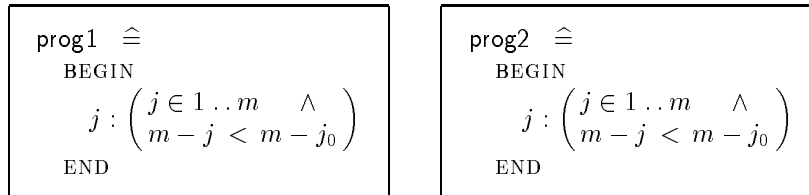
Next are the refinement of `init` and `aprog`.



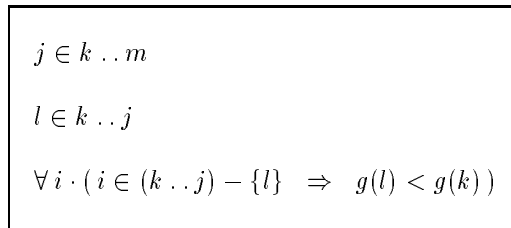
In the refinement of event `progr`, an index l is chosen “arbitrarily” in the range $k..m$, with a corresponding value $g(l)$ that happens to be the minimum of g in that sub-part. This index is then exchanged with k . Finally, k is incremented.



We finally have two anticipating events decreasing the quantity $m - j$



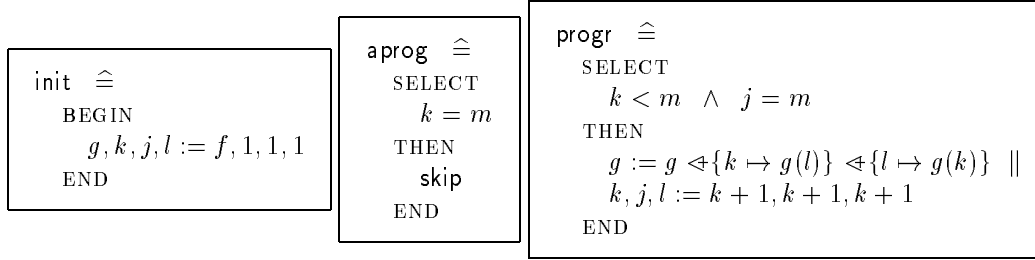
Our next step consists in determining the minimum chosen “arbitrarily” above. For this, we introduce an extra index l . The index j ranges from k to m , whereas l ranges from k to j . The value of g at index l is supposed to be the minimum of g on the sub-part ranging from k to j , Formally



This can be illustrated on the next diagram extending the previous one:

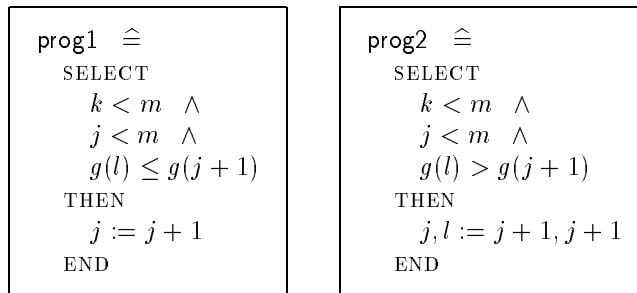
1	sorted and smaller	$k - 1$	k	$g(l)$ is the minimum	j	m
---	--------------------	---------	-----	-----------------------	-----	-----

Next are the refinements of the abstract events with a few modifications only for the first three:



In the concrete event **progr**, the strengthening of the guard (with condition $j = m$) implies that the value of the variable l corresponds exactly to the minimum chosen arbitrarily in the abstraction.

Here are the refinement of events **progr1** and **progr2**. Notice that in event **progr1**, the condition $g(l) \leq g(j + 1)$ is equivalent to $g(l) < g(j + 1)$ since $g(l)$ and $g(j + 1)$ cannot be equal. This is due to the facts that: (1) the variable l (ranging from k to j) and the expression $j + 1$ have distinct values, and (2) f is injective.



Applying RULE 2, we now put together these events. Applying then RULE 3 with **progr**, we obtain the outermost loop body. By applying again RULE 3, this time with event **aprogr**, and adding **init**, we obtain eventually the following program:

```

cprog  $\hat{=}$ 
BEGIN
   $g, k, j, l := f, 1, 1, 1$                                init
  WHILE  $k < m$  DO
    WHILE  $j < m$  DO
      IF  $g(l) \leq g(j + 1)$  THEN
         $j := j + 1$                                        prog1
      ELSE
         $j, l := j + 1, j + 1$                                prog2
      END
    END
  END ;
  BEGIN
     $g := g \Leftarrow \{k \mapsto g(l)\} \Leftarrow \{l \mapsto g(k)\} \parallel$ 
     $k, j, l := k + 1, k + 1, k + 1$                          progr
  END
END
END

```

Note that the initialisation of the inner loop variables, namely j and l , is made in two different places: either in the proper initialisation at the beginning of the program, or in the trailing statement after the inner loop itself.

Example 10: Almost Linear Sorting

Contrarily to the previous example, we now study a very efficient (linear) sorting algorithm, which can be used successfully in certain circumstances. Here are the informal explanations. Suppose that the n distinct values that have to be sorted are all members of a certain interval $1..m$. When m is equal to n , then the sorting is trivial since the sorted sequence is then clearly the identity function built on the interval $1..n$. Sorting is just done in a linear time.

Suppose now that this number n of distinct values is only *slightly smaller* than the maximum m of the interval $1..m$ within which these values are supposed to be. In other words, not all members of $1..m$ have to be sorted, but *almost all of them*. In that case, it seems quite reasonable to conjecture that there exists a way of sorting these numbers, which should not be very far from being linear: there is no reason indeed for that small difference between n and m to suddenly induce a large difference in the sorting time with respect to the linear time that we had when n and m were identical. The purpose of this exercise is to construct an algorithm with this intuition in mind.

Formally, we have two positive constant numbers n and m , and also a constant function f , the sequence to be sorted, which is total and injective from the interval $1..n$ to the interval $1..m$. We finally have a result variable g and an anticipating variable k .

$n \in \mathbb{N}_1$ $m \in \mathbb{N}_1$ $f \in 1..n \mapsto 1..m$	$g \in 1..n \leftrightarrow 1..m$ $k \in 0..m$
---	--

The event `aprog` is almost the same as in the previous example (the destination of the function g was \mathbb{N} , it is now replaced by the interval $1..m$):

$\text{aprog} \hat{=} \text{BEGIN}$ $g : \left(\begin{array}{l} g \in 1..n \rightarrow 1..m \wedge \\ \forall (i, j) \cdot \left(\begin{array}{l} i \in 1..n-1 \wedge \\ j \in i+1..n \\ \Rightarrow \\ g(i) < g(j) \end{array} \right) \wedge \\ \text{ran}(g) = \text{ran}(f) \end{array} \right)$ END

The two anticipating events are as follows

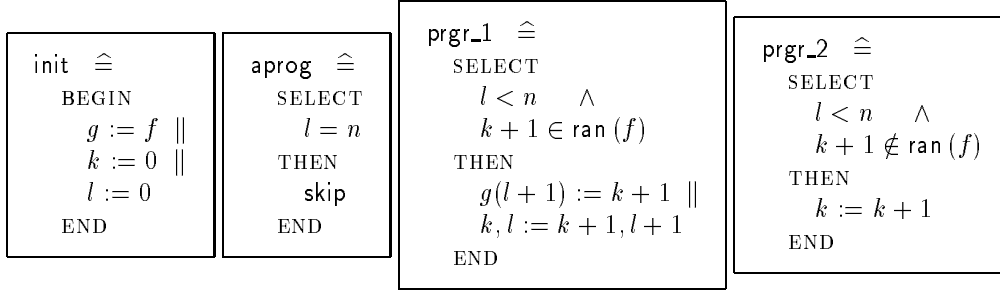
$\text{prgr_1} \hat{=} \text{BEGIN}$ $g, k : \left(\begin{array}{l} g \in 1..n \leftrightarrow 1..m \wedge \\ k \in 0..m \wedge \\ m - k < m - k_0 \end{array} \right)$ END
--

$\text{prgr_2} \hat{=} \text{BEGIN}$ $g, k : \left(\begin{array}{l} g \in 1..n \leftrightarrow 1..m \wedge \\ k \in 0..m \wedge \\ m - k < m - k_0 \end{array} \right)$ END
--

We now refine this specification by introducing another index, l , belonging to $1..n$. We shall suppose, as an invariant, that the sequence g is indeed sorted in its sub-part ranging from 1 to l . Moreover, the sorted values are within the sub-range $1..k$. Finally, we state that the cardinal of the domain of f , when range-restricted to the interval $1..k$, is exactly l . We also have an anticipating variable j . Formally :

$g \in 1..n \rightarrow 1..m$ $l \in 0..n$ $\forall (i, j) \cdot (i \in 1..l-1 \wedge j \in i+1..l \Rightarrow g(i) < g(j))$ $g[1..l] = \text{ran}(f) \cap (1..k)$ $\text{card}(\text{dom}(f \triangleright 1..k)) = l$ $j \in 0..n$
--

Next are the refinement of the events.

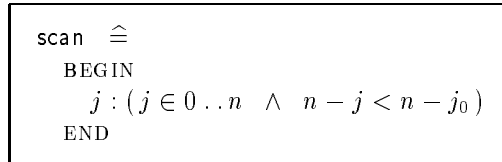


As can be seen, we gradually fill in the sequence g (in the event `prgr1`) with the value $k+1$, since it is the *next* value to be sorted (remember that the values in $1..k$ have already been sorted according to the invariant). The membership of $k+1$ in the range of the sequence f is checked by the guards of events `prgr1` and `prgr2`. The process is completed (as observed by event `aprog`) when l is equal to n .

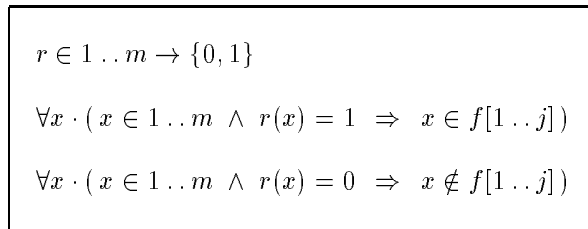
It is not completely obvious that `aprog` refines its abstraction: when l is equal to n (according to the guard of `aprog`) then clearly g is sorted, but how can we be certain that the *entire* sequence f has been sorted? Here is an informal argument. According to the last invariant, the domain of f , range-restricted to $1..k$, is n , therefore the domain of f , range-restricted to $k+1..m$, is 0 and the corresponding set is empty. This means that the set $\text{ran}(f) \cap (1..k)$ covers exactly the sorted numbers. According to the last but one invariant this is exactly (since l is n) the range of g .

Likewise, it is not completely obvious that the event `prgr_2` maintains the invariant $k \in 0..m$. But, in fact, we can prove that the guard $l < n$ implies $k < m$. We shall give here some informal argument in favor of the contraposition. When k is m then, according to the last invariant, the cardinal of the domain of f is l , which is then equal to n since the domain in question is $1..n$.

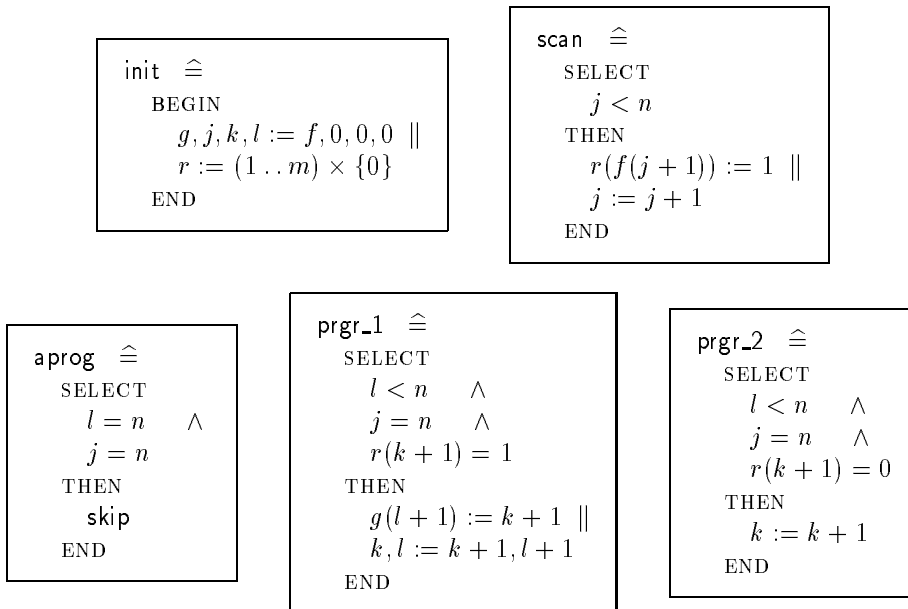
The anticipating event `scan` is as follows



Our next refinement consists in “storing” in advance the test corresponding to the predicate $k+1 \in \text{ran}(g)$. For this we introduce a function r from $1..n$ to $\{0, 1\}$. The function r is filled in by scanning f (this will be done by event `scan` working with index j). The test $k+1 \in \text{ran}(g)$ is then replaced by $r(k+1) = 1$. Formally, we have:



Here are the refinement of the events (notice the strengthening of the guards):



By applying RULE 2 to events prgr1, prgr2 and then RULE 3 twice, we obtain:

