Development of Concurrent Systems by Incremental Transformation[1]

E. Pascal Gribomont
Philips Research Laboratory
avenue van Becelaere, 2
B - 1170 Brussels (Belgium)

Abstract. A formal development method for concurrent programs is proposed. It generalizes several variants of the stepwise refinement method often used in concurrency, in that not only atomicity refinements, but also arbitrary transformations, are taken into account. The method is illustrated by simple examples.

1 Introduction

It is now widely accepted that systems of concurrent processes are complex objects, which should be specified, designed and verified in a formal way. Much work has been devoted to this problem, and useful results have been obtained. Some of them are now briefly recalled, upon which our work is based.

Logics and related formalisms are adequate tools for the expression of properties of concurrent systems. They are also appropriate for proving these properties [38,28,34,35]. The design and the verification of concurrent systems are easier when appropriate abstract languages are used, for instance CSP [23,24], Action Systems [5] or UNITY [10]. The semantics of these languages and of the programs written in them can be formally stated. The notion of invariant, initially introduced for sequential programming, has proved very useful in parallel programming as well [1,38,43].

These formal tools (languages, logics and deduction systems) should be the foundation of a formal methodology of parallel programming [24,32,10]. The usual methodology for the design of concurrent systems relies on the notion of "stepwise refinement". Roughly speaking, this method consists in generating a sequence of systems, each of them being slightly more complicated and, hopefully, "better" than its predecessor. Many examples of concurrent systems developed by the stepwise refinement method (in fact, several variants of it) have appeared in the literature; some of them are [14,8,27,3,9]. It should be emphasized that this method seems to be quite general; it is not restricted to some languages or programming constructs.

In practice, the designer has first to discover a possible refinement, and then to establish that this refinement is valid. The first step is usually a creative one: the designer wants to transform the program in order to satisfy a stronger specification, or the same specification in a more efficient way. The designer has to identify some parts of the program which maybe should be replaced by new parts. The second step is to check whether an attempted refinement is acceptable or not. This can be done by producing an appropriate invariant of the refined version. It has been observed that an adequate "small" change in the system usually induces a "small" change in the invariant; the nature of this change is not identified easily and its expression depends on the language used to write the assertions.

Much work about the problem of stepwise refinement in concurrency has been and still is performed. The obtained results can be roughly classified in two categories. The papers of the first category contain examples of development. The authors introduce a sequence of versions of their programs, each of them with a proof, that is, an invariant. This kind of presentation allows the readers and the authors themselves to understand the subtleties of the problem and its proposed solution in an incremental way. An early example in this category is [14]; many examples presented in [10] are also based on the stepwise refinement method.

The second category contains more theoretical results about the notion of refinement itself. These results are usually about a particular but important kind of refinement, called the "atomicity refinement",

[1]Supported in part by the ESPRIT project ATES.

or the "sequential refinement". The notion of atomicity occurs in concurrent programming when the interleaving semantics is used, that is, when concurrency is modelled by non-determinism : a step of the computation consists in executing a statement of some arbitrarily chosen process of the system. In a "coarse-grained" system, a statement can be a rather big segment of program, involving the access to several variables; in a "fine-grained" system, a statement is a simple assignment or a simple test. A sequential refinement consists in "breaking" a statement into a sequence of more elementary statements; for instance, the double assignment $(x, y) := (y, x)$ is refined into the sequence $t := x$; $x := y$; $y := t$, where t is a new variable. An early work about refinements is [31]; more recent papers are [27,30,7].

From the theoretical point of view, it is rather easy to deal with sequential refinements. Suppose we have a program P whose invariance properties are established by some invariant I. If a process π of P contains some statement S, sequentially equivalent to $S_1; S_2$, one can attempt to split the statement S into the sequence formed by the more elementary statements S_1 and S_2. Due to possible interaction with other processes, the program P' obtained by such a transformation is not always correct with respect to the specifications of P. A simple method to evaluate the impact of the refinement on the semantics of the program is as follows. Let R be the predicate which is false when the control of the execution of π is between the statements S_1 and S_2 (that means that the next statement executed by π will be S_2) and true otherwise. If a formula J exists such that the formula $I' =_{def} [(R \Rightarrow I) \wedge (\neg R \Rightarrow J)]$ is an invariant of the refined version P', then the specified invariance properties of P, summarized in the invariant I, are preserved in the refined program P' (except maybe in "transient" states, when R is false). On the contrary, when no adequate J can be found, the semantics of P' deeply differs from the semantics of P, and the refinement is likely to be incorrect. This method and the strategy for finding J, when it exists, have been presented and illustrated in detail in [18,19].

In practice, sequential refinement is not sufficient, since new variables are introduced in a very restricted way. Especially, this kind of refinement disallows the strengthening and the weakening of the guards of the statements, and modifying guards is often used in practice during the design of concurrent systems. More general transformations, involving for instance the unrestricted introduction of new variables, may induce substantial changes in the computation, if the guards of some statements of the program are made dependent on the value of the new variables.

The purpose of this paper is to generalize the method recalled above to the general case of arbitrary transformations. The main problem is the choice of a "refinement formula" R; this choice is evident for sequential refinements, but not for more general transformations. On the contrary, when the formula R has been chosen, the determination of an adequate J (when there is one) can be done in the same way for a sequential refinement or for an arbitrary transformation.

The paper goes on as follows. The formal tools needed to explain and apply the method are presented in Section 2; the method for finding R and J is presented in Section 3 and illustrated in Sections 4 and 5. Section 6 is a conclusion.

2 Formal tools

In this section, we first introduce briefly our programming notation, called FCS (for Formal Concurrent System), with a simple example. This example will also show that sequential refinements are not sufficient in practice : more general transformations are needed, even in the development of elementary concurrent systems. Afterwards, the Hoare logic for FCS is introduced, together with the notions of weakest liberal precondition and strongest liberal postcondition.

2.1 A programming notation

The programming notation FCS is elementary and best explained by an example. Here is first a graphical presentation of a toy algorithm.

This is a simple scheme for mutual exclusion. P and Q are cyclic processes sharing the variable T (for "Turn"). A process, say P, can perform internal computation (not involving the shared variable T) either in its non-critical state p_0 or in its critical state p_c. As, at every time, at most one process

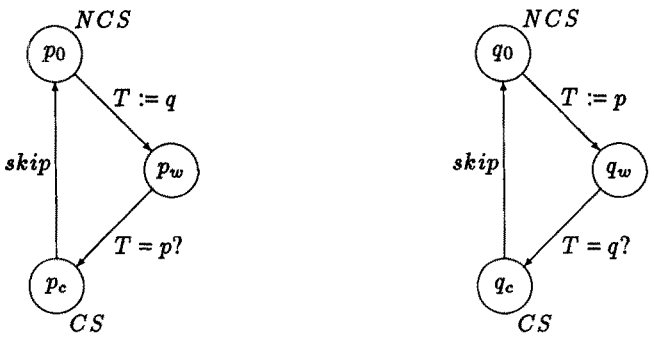

Figure 1: A toy algorithm

can execute its critical section (CS), the shared variable is used to implement the mutual exclusion; when, for instance, process Q is in its critical section, then process P must be delayed, in its waiting state p_w. Classical *place predicates* will be used; for instance, the formula $(at\ p_w \wedge at\ q_c)$ is true when process P is in its waiting state and process Q is in its critical state. The mutual exclusion is modelled by the formula

$$\neg (at\ p_c \wedge at\ q_c).$$

The equivalent FCS, called $\mathcal{S}_{00}$, is simply a lexical version of the graphical representation:

$$\begin{aligned}
&\mathcal{P} = \{P, Q\}, \text{ where } P = \{p_0, p_w, p_c\} \text{ and } Q = \{q_0, q_w, q_c\};\\
&\mathcal{M} = \{T : \{p, q\}\};\\
&\mathcal{T} = \{(p_0,\ T := q,\ p_w), \qquad (q_0,\ T := p,\ q_w),\\
&\quad (p_w,\ T = p \longrightarrow skip,\ p_c), \quad (q_w,\ T = q \longrightarrow skip,\ q_c),\\
&\quad (p_c,\ skip,\ p_0), \qquad (q_c,\ skip,\ q_0)\}.
\end{aligned}$$

An FCS has three components. First, the *set of (formal) processes* $\mathcal{P}$, second, the *memory* $\mathcal{M}$ and, third, the *set of transitions* $\mathcal{T}$. Processes are disjoint non-empty sets of *labels*; the memory is a finite set of typed program variables. A *transition* is an expression like $\tau = (O,\ G \longrightarrow A,\ E)$, where O is the *origin* (or *entry point*), E is the *extremity* (or *exit point*) and $G \longrightarrow A$ is the *guarded (multiple) assignment* of the transition. An axiomatic semantics for FCS is given in the next paragraph. Let us simply mention here that a step of computation consists in executing an arbitrary executable transition (if there is one). The transition τ is *executable* when the formula $cond(\tau) =_{def} (at\ O \wedge G)$ is true; after the execution of τ, $at\ E$ is true. A formula I is an *invariant* of an FCS if each transition τ respects I: if τ is executed from a state satisfying I (and $cond(\tau)$), then the resulting state also satisfies I. An *initial condition* A may be specified for an FCS; it means that only computations whose first state satisfies A are considered. If an FCS is introduced with an invariant, this invariant is also the initial condition, unless stated otherwise.

Comments. Every transition of the system $\mathcal{S}_{00}$ involves a single process (either P or Q). Some systems can contain transitions involving several processes; in this case, the origin and the extremity contains a label of each involved process.

We are not interested in the internal computation performed by the processes; as a consequence, it is not modelled in the formal concurrent system $\mathcal{S}_{00}$.

With self-explaining notation, a useful invariant of the system is

$$\begin{aligned}
I_{00} =_{def}\ & (at\ p_w \Rightarrow (T = q \vee at\ q_w)) \wedge (at\ q_w \Rightarrow (T = p \vee at\ p_w))\\
& \wedge (at\ p_c \Rightarrow T = p) \wedge (at\ q_c \Rightarrow T = q).
\end{aligned}$$

(The invariant I_{00} is also the initial condition.)
Let us emphasize two implicit parts of any invariant. First, the *process rule* asserts that each process is at exactly one place at every time. For process P, this rule is formalized into the assertion

$$at\ p_0 + at\ p_w + at\ p_c = 1$$

(with the usual convention: *true* is identified with 1 and *false* is identified with 0). The second implicit part is the *variable rule*: each variable has exactly one value at a time, and the set of the possible values is the *type* of the variable. For the variable T, one can formalize the rule into the assertion

$$T = p \ \vee \ T = q\,.$$

As the formula $\neg(at\ p_c \wedge at\ q_c)$ is a logical consequence of the invariant I_{00}, the system S_{00} guarantees mutual exclusion.

Some useful pieces of notation are introduced now.
If $L = \ell_1 \cdots \ell_n$ is a sequence (or a set) of labels belonging to distinct processes $P_1, \ldots, P_n$ of some system S, then $at\ L$ stands for $(at\ \ell_1 \wedge \cdots \wedge at\ \ell_n)$. If B is a formula, then $B[at\ L]$ is obtained by "making $at\ L$ true" in formula B; more formally, $B[at\ L]$ is obtained by replacing each place predicate $at\ k$ occurring in B by *true* when $k \in \{\ell_1, \ldots, \ell_n\}$, by *false* when $k \in (P_1 \cup \cdots \cup P_n) \setminus \{\ell_1, \ldots, \ell_n\}$, by $at\ k$ (no change) when $k \notin (P_1 \cup \cdots \cup P_n)$.
Notice that the formulas $(B \wedge at\ L)$ and $(B[at\ L] \wedge at\ L)$ are always equivalent.

Here is an example, about the system S_{00} introduced in the previous section. If B is I_{00}, then $B[at\ p_w]$ is obtained by replacing in B the formulas $at\ p_0$, $at\ p_w$ and $at\ p_c$ by *false*, *true* and *false*, respectively. The result is $[(T = q \vee at\ q_w) \wedge (at\ q_c \Rightarrow T = q)]$, reducing into $(T = q \vee at\ q_w)$. Similarly, $B[at\ p_w q_c]$ is obtained by replacing, in $B[at\ p_w]$, the place predicates $at\ q_0$ and $at\ q_w$ by *false*, and $at\ q_c$ by *true*; the result is $T = q$.

2.2 Hoare logic for FCS

In the previous paragraph, the invariant I_{00} of the system S_{00} has been introduced without proof; let us now introduce the Hoare deduction system [22,2] for proving that some formula is an invariant of some FCS. The deduction system also provides an axiomatic semantics for the language FCS.

The following elements are fixed throughout this paragraph:

- An FCS $S = (\mathcal{P}, \mathcal{M}, \mathcal{T})$.
- A family of processes $\{P_1, \ldots, P_n\} \subset \mathcal{P}$.
- A family of transitions $\mathcal{T}_0 \subset \mathcal{T}$.
- Labels $\ell_i, m_i \in P_i$, for all i ($\ell_i = m_i$ is allowed).
- A transition $\tau = (L,\ C \longrightarrow A\,,\ M) \in \mathcal{T}$, where C and A are respectively a guard and an assignment; L and M stand for $\ell_1 \cdots \ell_n$ and $m_1 \cdots m_n$, respectively. (Only the case $n = 1$ has appeared in the example S_{00}.)
- P and Q are assertions, that is, logical formulas interpreted on the states of S.

Hoare's logic is adapted to FCS by the following rules.

$$\{P\}\ C \longrightarrow A\ \{Q\} \quad =_{def} \quad \{P \wedge C\}\ A\ \{Q\}\,, \tag{1}$$

$$\{P\}\ (L,\ C \longrightarrow A\,,\ M)\ \{Q\} \quad =_{def} \quad \{P[at\ L]\}\ C \longrightarrow A\ \{Q[at\ M]\}\,, \tag{2}$$

$$\{P\}\ \mathcal{T}_0\ \{Q\} \quad =_{def} \quad \bigwedge_{\tau \in \mathcal{T}_0} [\ \{P\}\ \tau\ \{Q\}\ ]\,. \tag{3}$$

These rules model the execution mechanism of FCS.
The first rule expresses the semantics of the guarded assignment $C \longrightarrow A$; it is equivalent to A when C is true; otherwise, it cannot be executed (as a consequence, the triple $\{\neg C\}\, C \longrightarrow A\, \{Q\}$ is vacuously true).
The second rule expresses the semantics of the transition; it can be executed only when $(at\ L \wedge C)$ is true and leads to a state where $at\ M$ is true.
The third rule is not mandatory; it is introduced as an abbreviation.

As an example, we will prove that the invariant I_{00} is respected by the transition $\tau = (p_0,\ T := q\ ,\ p_w)$. Due to rule (2), the triple $\{I_{00}\}\, \tau\, \{I_{00}\}$ reduces to the triple $\{I_{00}[at\ p_0]\}\, T := q\, \{I_{00}[at\ p_w]\}$. The precondition is evaluated in

$$(at\ q_w \Rightarrow T = p) \wedge (at\ q_c \Rightarrow T = q),$$

whereas the postcondition is evaluated in

$$(T = q \vee at\ q_w) \wedge (at\ q_c \Rightarrow T = q).$$

The classical Hoare axiom for the assignment is

$$\{P\}\, x := e\, \{Q\} \iff (P \Rightarrow Q[x/e]);$$

it allows to further reduce the triple to the implication

$$[(at\ q_w \Rightarrow T = p) \wedge (at\ q_c \Rightarrow T = q)] \Rightarrow [(q = q \vee at\ q_w) \wedge (at\ q_c \Rightarrow q = q)],$$

which is a tautology.

2.3 Programming calculus

The liberal version of Dijkstra's programming calculus is adapted to the language FCS as follows.

$$wlp[(C \longrightarrow A);\ Q] =_{def} (C \Rightarrow wlp[A;\ Q]), \tag{4}$$

$$slp[P;\ (C \longrightarrow A)] =_{def} slp[(P \wedge C);\ A], \tag{5}$$

$$wlp[(L,\ C \longrightarrow A,\ M);\ Q] =_{def} at\ L \Rightarrow wlp[(C \longrightarrow A);\ Q[at\ M]], \tag{6}$$

$$slp[P;\ (L,\ C \longrightarrow A,\ M)] =_{def} slp[P[at\ L];\ (C \longrightarrow A)] \wedge at\ M, \tag{7}$$

$$wlp[\mathcal{T};\ Q] =_{def} \bigwedge_{\tau \in \mathcal{T}} wlp[\tau;\ Q], \tag{8}$$

$$slp[P;\ \mathcal{T}] =_{def} \bigvee_{\tau \in \mathcal{T}} slp[P;\ \tau]. \tag{9}$$

The predicate transformers *wlp* and *slp* are strongly related to Hoare's logic; their extensions have been defined in such a way that the three formulas

$$\{P\}\, X\, \{Q\},\quad P \Rightarrow wlp[X;\ Q],\quad slp[P;\ X] \Rightarrow Q$$

are equivalent not only when X is an assignment, but also when it is a guarded assignment, a transition or a set of transitions. The first formula is used when P and Q are both known or both unknown. The second formula is used when only Q is known and the third one is used when only P is known.
Comment. Let us emphasize the difference between *wp* [13], and *wlp* [11]:

$$wlp[(C \longrightarrow A);\ Q] \equiv C \Rightarrow wlp[A;Q],$$
$$wp[\mathbf{if}\ C \longrightarrow A\ \mathbf{fi};\ Q] \equiv C \wedge wp[A;Q].$$

3 A methodology of incremental transformation

In this section, we introduce a notion of refinement that is general enough to allow any kind of program transformation. Afterwards, we show how the effect of such a transformation on the semantics of the refined system can be formalized. The method generalizes the results presented in [18,19].

3.1 Examples and definitions

Various kinds of transformations usually referred to as "refinements" are now illustrated with the elementary system $\mathcal{S}_{00}$. This example provides a useful guideline towards an appropriate definition of the concept of refinement. It formalizes the original development of Peterson's algorithm given in [39].

The system $\mathcal{S}_{00}$ has an obvious drawback: it enforces the processes to access their critical section strictly in turn. This is too restrictive: we would like to allow one process to proceed when the other is in its non-critical section. Otherwise stated, the guards $T = p$ and $T = q$ should be weakened into the guards $(at\ q_0 \vee T = p)$ and $(at\ p_0 \vee T = q)$, respectively.

As place predicates are not considered as program variables in FCS, this transformation is (syntactically) disallowed. However, nothing prevents the introduction of new variables, for recording the truthvalues of $at\ p_0$ and $at\ q_0$. Such variables are called *secondary variables* because their values are fully determined by the values of already existing, *primary* objects (variables and place predicates); on the contrary, the values of the primary variables do not depend on the values of the secondary variables. As a consequence, the impact of the introduction of secondary variables on the invariant I of the system is trivial: if a secondary variable x is introduced, then the invariant becomes $I' =_{def} (I \wedge P(x))$, where $P(x)$ describes the value of x in terms of primary objects.

Let us introduce boolean variables inP and inQ to record the values of the place predicates $at\ p_0$ and $at\ q_0$. More precisely, we define $inP =_{def} \neg at\ p_0$ and $inQ =_{def} \neg at\ q_0$. This leads to the system $\mathcal{S}_{11}$ given below.

$$
\begin{aligned}
&\mathcal{P} = \{P, Q\}, \text{ where } P = \{p_0, p_w, p_c\} \text{ and } Q = \{q_0, q_w, q_c\};\\
&\mathcal{M} = \{T : \{p, q\},\ inP : bool,\ inQ : bool\};\\
&\mathcal{T} = \{(p_0,\ (inP, T) := (true, q),\ p_w),\quad (q_0,\ (inQ, T) := (true, p),\ q_w),\\
&\qquad (p_w,\ T = p \longrightarrow skip,\ p_c),\qquad (q_w,\ T = q \longrightarrow skip,\ q_c),\\
&\qquad (p_c,\ inP := false,\ p_0),\qquad (q_c,\ inQ := false,\ q_0)\}.
\end{aligned}
$$

Obviously, the invariant I_{11} associated with the system $\mathcal{S}_{11}$ will be

$$I_{11} =_{def} [I_{00} \wedge (at\ p_0 \equiv \neg inP) \wedge (at\ q_0 \equiv \neg inQ)].$$

Now, we would like to weaken the guards $T = p$ and $T = q$ into $(\neg inQ \vee T = p)$ and $(\neg inP \vee T = q)$, respectively. This will lead to the refined system $\mathcal{S}_{22}$, but an invariant I_{22} has to be discovered for it. Let us note that this transformation is less trivial than the previous one; in particular, the new invariant I_{22} is not bound to be $(I_{11} \wedge F)$ for some formula F. A method for discovering I_{22} is explained in the sequel.

As $\mathcal{S}_{22}$ contains double assignments, sequential refinements should be attempted. However, such transformations require that new "intermediate" labels are introduced first. In fact, two new labels p_i and q_i are introduced in P and Q respectively; this leads to the refined system $\mathcal{S}_{33}$ and to a refined invariant I_{33} for it. As no transition evokes the new labels, the new invariant will be

$$I_{33} =_{def} (I_{22} \wedge \neg at\ p_i \wedge \neg at\ q_i).$$

As a last step, the sequential refinements themselves can be attempted; for instance, the transition

$$(p_0,\ (inP, T) := (true, q),\ p_w)$$

will be replaced either by the transitions

$$(p_0,\ inP := true\ ,\ p_i),$$
$$(p_i,\ T := q\ ,\ p_w),$$

or by the transitions

$$(p_0,\ T := q\ ,\ p_i),$$
$$(p_i,\ inP := true\ ,\ p_w).$$

A similar replacement will be done for the transition

$$(q_0,\ (inQ, T) := (true, p),\ q_w).$$

These transformations will lead to the refined system $\mathcal{S}_{44}$ and, hopefully, to an appropriate invariant I_{44} for this system. (I_{44} will be appropriate if the formula $\neg(atp_c \wedge atq_c)$ is still a logical consequence of it.) Once again, the derivation of I_{44} from I_{33} is not trivial; especially, the way the assignments about inP (or inQ) and T are ordered may be important.

As a conclusion, we see that two very different kinds of refinements must be performed. First, the refinements from $\mathcal{S}_{00}$ to $\mathcal{S}_{11}$ and from $\mathcal{S}_{22}$ to $\mathcal{S}_{33}$ are rather trivial. The role of these transformations is simply to introduce new objects (secondary variables or labels), in order to prepare more substantial and less trivial refinements. For this reason, they are called *preliminary refinements*. Their main characteristic is that they respect the invariant of the system; with self-explaining notation, we have the relation

$$I_{new} \equiv (I_{old} \wedge F),$$

where the formula F describes the new objects. No specific methodology is needed to deal with such transformations; they are also called *non-semantical refinements*, since they do not alter the invariant (that is, the semantics) of the system.

On the other hand, the refinements from $\mathcal{S}_{11}$ to $\mathcal{S}_{22}$ and from $\mathcal{S}_{33}$ to $\mathcal{S}_{44}$ are not trivial. They involve the introduction of no new object, but some transitions are replaced by new ones. In order not to rule out any kind of program transformation, no restriction is placed on the nature of the new transitions, except that they must be syntactically acceptable; as a consequence, they cannot evoke undefined labels and variables.[2] As such transformations are likely to alter deeply the invariant (the semantics) of the system, they are called *semantical refinements*. The *sequential refinement* (or *atomicity refinement*) is the most elementary case of semantical refinement.

3.2 Sequential refinement

The purpose of this paper is to evaluate the impact on the invariant of any kind of semantical refinement but, as the particular case of sequential refinement has already been studied [7,18], it is helpful to recall it first.

A sequential refinement $\mathcal{S}'$ of an FCS $\mathcal{S}$ is obtained by replacing a transition $\tau = (\ell, C \longrightarrow A, m)$ of $\mathcal{S}$ by two transitions $\tau' = (\ell, C' \longrightarrow A', n)$ and $\tau'' = (n, C'' \longrightarrow A'', m)$, where τ' and τ'' satisfy the following compatibility conditions. First, n is a new label; one can suppose that it has been introduced by a preliminary refinement, and that the invariant of the system $\mathcal{S}$ is $(I \wedge \neg at\ n)$. Second, for all assertions P and Q, if the triple $\{P\}\tau\{Q\}$ is true, then there exists an assertion R such that the triples $\{P\}\tau'\{R\}$ and $\{R\}\tau''\{Q\}$ are also true. The second condition can be rewritten as

$$wlp[\tau;\ Q] \Rightarrow wlp[\tau';\ wlp[\tau'';\ Q]],\ \text{for all } Q.$$

[2] If new labels and variables are needed, preliminary refinements must be performed first to introduce them.

Comment. The definition of a sequential refinement given here agrees with the formal notion of refinement introduced in [6,36,37]. This notion is convenient in the framework of sequential programming but, as already mentioned, insufficient in the framework of parallel programming.

We briefly recall the method for dealing with sequential refinements. (More details are given in [18] and will be given, for the general case, in the next paragraph.) A definition is introduced first. A *transient state* of the sequentially refined system $\mathcal{S}'$ is a state where the place predicate $at\ n$ is true (n is the new label). It is considered that a sequential refinement is acceptable, or valid, when the invariant of the initial system $\mathcal{S}$ remains true throughout the computations of the refined system $\mathcal{S}'$, except maybe in transient states. This is the case if and only if an assertion J exists such that

$$I' =_{def} [(\neg at\ n \Rightarrow I) \wedge (at\ n \Rightarrow J)]$$

is an invariant of the system $\mathcal{S}'$. As a consequence, to validate a refinement means to discover an adequate J. Such a J is a solution of the constraint

$$\{(\neg at\ n \Rightarrow I) \wedge (at\ n \Rightarrow J)\}\ \mathcal{T}'\ \{(\neg at\ n \Rightarrow I) \wedge (at\ n \Rightarrow J)\}, \tag{10}$$

where $\mathcal{T}' = (\mathcal{T} \setminus \{\tau\}) \cup \{\tau', \tau''\}$ is the set of transitions of the refined system $\mathcal{S}'$. It is helpful to decompose this constraint into four parts, depending on the fact that $at\ n$ can be true or false in the precondition or in the postcondition. The four resulting constraints are listed below.

$$\begin{array}{l} \{\neg at\ n \wedge I\}\ \mathcal{T}'\ \{\neg at\ n \Rightarrow I\}, \\ \{\neg at\ n \wedge I\}\ \mathcal{T}'\ \{at\ n \Rightarrow J\}, \\ \{at\ n \wedge J\}\ \mathcal{T}'\ \{\neg at\ n \Rightarrow I\}, \\ \{at\ n \wedge J\}\ \mathcal{T}'\ \{at\ n \Rightarrow J\}. \end{array} \tag{11}$$

The first constraint is easy to check, since J does not occur in it. The second and the third constraints are rewritten respectively into

$$\begin{array}{l} (\mathit{slp}[(\neg at\ n \wedge I);\ \mathcal{T}'] \wedge at\ n) \Rightarrow J, \\ J \Rightarrow (at\ n \Rightarrow \mathit{wlp}[\mathcal{T}';\ (\neg at\ n \Rightarrow I)]), \end{array}$$

and give respectively a strongest possible choice and a weakest possible choice for J.[3] Last, the fourth constraint contains two occurrences of J; as a consequence, it cannot be solved easily. The proposed strategy is to repeatedly select "candidates" in the set of formulas determined by the *slp*- and *wlp*-constraints, and to use the fourth constraint as an acceptance/rejection filter.

3.3 The general case

Our purpose is to identify the semantical consequences of the replacement of a transition by an arbitrary set of new transitions. The starting point is the constraints (10,11). The conditions $\neg at\ n$ and $at\ n$ have no longer any meaning here; they are replaced respectively by a *refinement condition* R and its negation $\neg R$. The problem is to determine an appropriate refinement condition. Our guideline for doing that will be to simplify the constraints as much as possible; we suppose that a refinement condition R has been chosen, and we look for the properties of R inducing simplification of the constraints. As a starting point, the invariant of the refined system $\mathcal{S}'$ is

$$I' \equiv [(R \Rightarrow I) \wedge (\neg R \Rightarrow J)].$$

In practice, the invariant I is the conjunction of several assertions. All of them are true when R is true, but some of them are false when R is false. This suggests a decomposition like $I =_{def} (I^- \wedge I^+)$, where the invariant I^- contains only non-altered assertions, whereas the formula I^+ may contain altered ones. More formally, we have $\{I\}\ \mathcal{T}'\ \{I^-\}$, and even $\{I^-\}\ \mathcal{T}'\ \{I^-\}$, but we have not $\{I\}\ \mathcal{T}'\ \{I^+\}$. This decomposition suggests to rewrite

[3] A further simplification is possible: $\mathcal{T}'$ can be replaced by τ' in the *slp*-constraint, and by τ'' in the *wlp*-constraint. Let us also mention that it is sufficient to find $J[at\ n]$.

$$I' \equiv [I^- \wedge (R \Rightarrow I^+) \wedge (\neg R \Rightarrow K)], \tag{12}$$

for appropriate I^- and I^+; formulas R and K are still unknown. (Note that J is $(I^- \wedge K)$.) If I is the conjunction $(I_1 \wedge \cdots \wedge I_n)$, it is convenient to evaluate first $\{I\}\, \mathcal{T}' \setminus \mathcal{T}\, \{I_j\}$, for $j = 1, \ldots, n$: the conjunction I^0 of the I_j's for which the triple is true is a good approximation of I^-. More often than not, I^0 is an invariant, so $I^- =_{def} I^0$ is appropriate. It is also convenient to choose $R =_{def} I^+$, for obtaining a simplified form of formula (12), that is:

$$I' \equiv [I^- \wedge (I^+ \vee K)]. \tag{13}$$

In order to discover K, the constraint

$$\{I'\}\, \mathcal{T}'\, \{I'\}$$

will be made explicit and simplified. We first observe that I^- is true in every relevant state; it is therefore convenient to discard any state not satisfying I^-. Formally, this means that I^- is taken as an additional axiom of the deduction system. The constraint is rewritten into

$$\{I^+ \vee K\}\, \mathcal{T}'\, \{I^+ \vee K\}.$$

The set $\mathcal{T}'$ of the transitions of the refined system $\mathcal{S}'$ can be partitioned into the set $\mathcal{T}' \cap \mathcal{T}$ of old transitions and the set $\mathcal{T}' \setminus \mathcal{T}$ of new transitions. Furthermore, we distinguish the case where I^+ is true in the precondition and the case where K is true in the precondition. The constraint is therefore split into the set of constraints listed below.

$$\begin{array}{l} \{I^+\}\, \mathcal{T}' \cap \mathcal{T}\, \{I^+ \vee K\}, \\ \{K\}\, \mathcal{T}' \cap \mathcal{T}\, \{I^+ \vee K\}, \\ \{I^+\}\, \mathcal{T}' \setminus \mathcal{T}\, \{I^+ \vee K\}, \\ \{K\}\, \mathcal{T}' \setminus \mathcal{T}\, \{I^+ \vee K\}. \end{array}$$

As I^+ is respected by old transitions, a further reduction leads to

$$\begin{array}{l} \{I^+\}\, \mathcal{T}' \setminus \mathcal{T}\, \{I^+ \vee K\}, \\ \{K\}\, \mathcal{T}'\, \{I^+ \vee K\}. \end{array} \tag{14}$$

As K occurs in the definition (13) of I' only in the term $(I^+ \vee K)$, it is not a real restriction to add the constraint $(K \Rightarrow \neg I^+)$. As a consequence, K must be stronger than formula $\neg I^+$ whereas, on the other hand, the first constraint of (14) asserts that $(I^+ \vee K)$ must be weaker than the formula $slp[I^+;\, \mathcal{T}' \setminus \mathcal{T}]$.
The set $\mathcal{C}$ of appropriate choices for K is therefore a subset of

$$\mathcal{C}_0 =_{def} \{X : [(K_U \Rightarrow X) \wedge (X \Rightarrow K_D)]\},$$

where

$$\begin{array}{l} K_U =_{def} \neg I^+ \wedge slp[I^+;\, \mathcal{T}' \setminus \mathcal{T}], \\ K_D =_{def} \neg I^+. \end{array} \tag{15}$$

The determination of the whole set $\mathcal{C}$ is usually intractable but, fortunately, we are satisfied with a single element of it. Furthermore, practice shows that, when an appropriate K exists, the formula K_U often turns out to be acceptable. If it is not, the strategy proposed for the sequential refinement still holds in the general case: repeatedly select "candidates" inside $\mathcal{C}_0$ and test them against the second constraint of (14); any candidate satisfying this test can be accepted.

4 A worked-out elementary example

In this section, the system $\mathcal{S}_{11}$ introduced in paragraph 3.1 is incrementally transformed into Peterson's algorithm.

4.1 Weakening a guard

The first step is to refine the system $\mathcal{S}_{11}$ (§ 3.1) by weakening the guard $T = p$ into the guard $(\neg inQ \vee T = p)$. This will lead to the system $\mathcal{S}_{21}$. In order to simplify the notation, we rename $\mathcal{S}_{11}$ into $\mathcal{S}$ and $\mathcal{S}_{21}$ into $\mathcal{S}'$. The latter is obtained from the former by replacing the transition

$$\tau : (p_w,\ T = p \longrightarrow skip,\ p_c),$$

by the transition

$$\tau' : (p_w,\ \neg inQ \vee T = p \longrightarrow skip,\ p_c).$$

The invariant I of the system $\mathcal{S}$ is

$$\begin{array}{l}(at\ p_w \Rightarrow (T = q \vee at\ q_w)) \wedge (at\ q_w \Rightarrow (T = p \vee at\ p_w)) \\ \wedge\ (at\ p_c \Rightarrow T = p) \wedge (at\ q_c \Rightarrow T = q) \\ \wedge\ (at\ p_0 \equiv \neg inP) \wedge (at\ q_0 \equiv \neg inQ);\end{array}$$

The triple $\{I\}\,T'\,\{A\}$ is checked for each assertion A; this leads to the decomposition $I = (I^- \wedge I^+)$, where

$$\begin{array}{lll} I^- & : & (at\ p_w \Rightarrow (T = q \vee at\ q_w)) \wedge (at\ q_w \Rightarrow (T = p \vee at\ p_w)) \wedge \\ & & (at\ q_c \Rightarrow T = q) \wedge (at\ p_0 \equiv \neg inP) \wedge (at\ q_0 \equiv \neg inQ), \\ I^+ & : & (at\ p_c \Rightarrow T = p). \end{array}$$

Formula I^- is easily checked to be an invariant of the refined system (but, obviously, this invariant does not ensure mutual exclusion). Definitions (15) give rise to

$$\begin{array}{lll} K_D & = & \neg I^+ \\ & = & (at\ p_c \wedge T = q), \\ K_U & = & \neg I^+ \wedge slp[I^+;\ T' \setminus T] \\ & = & \neg I^+ \wedge slp[I^+;\ (p_w,\ \neg inQ \vee T = p \longrightarrow skip,\ p_c)]) \\ & = & (at\ p_c \wedge T = q) \wedge [at\ p_c \wedge (\neg inQ \vee T = p)] \\ & = & at\ p_c \wedge T = q \wedge \neg inQ\,. \end{array}$$

The "candidate set" for K is

$$\mathcal{K}_0 \ = \ \{X \ : \ [(at\ p_c \wedge T = q \wedge \neg inQ) \Rightarrow X] \wedge [X \Rightarrow (at\ p_c \wedge T = q)]\}.$$

We tentatively select $K =_{def} K_U$ and check that the triple $\{K\}\,T'\,\{I^+ \vee K\}$ holds (recall that, in the present context, I^- is assumed to be true in all states). The formula $I^+ \vee K$ reduces to $(at\ p_c \Rightarrow (T = p \vee \neg inQ))$. The formula $I' =_{def} [I^- \wedge (I^+ \vee K)]$ can be simplified into

$$\begin{array}{lll} I_{21} & =_{def} & (at\ p_w \Rightarrow (T = q \vee at\ q_w)) \wedge (at\ q_w \Rightarrow (T = p \vee at\ p_w)) \\ & & \wedge\ (at\ p_c \Rightarrow (T = p \vee \neg inQ)) \wedge (at\ q_c \Rightarrow T = q) \\ & & \wedge\ (at\ p_0 \equiv \neg inP) \wedge (at\ q_0 \equiv \neg inQ). \end{array}$$

It is straightforward to check that the formula

$$I_{21} \Rightarrow \neg(at\ p_c \wedge at\ q_c)$$

is valid; as a consequence, the refinement preserves the mutual exclusion.

A similar work leads to the system $\mathcal{S}_{22}$ and the invariant

$$\begin{array}{lll} I_{22} & =_{def} & (at\ p_w \Rightarrow (T = q \vee at\ q_w)) \wedge (at\ q_w \Rightarrow (T = p \vee at\ p_w)) \\ & & \wedge\ (at\ p_c \Rightarrow (T = p \vee \neg inQ)) \wedge (at\ q_c \Rightarrow (T = q \vee \neg inP)) \\ & & \wedge\ (at\ p_0 \equiv \neg inP) \wedge (at\ q_0 \equiv \neg inQ). \end{array}$$

4.2 Sequential decomposition

We consider now a sequential refinement. The initial version is $\mathcal{S} =_{def} \mathcal{S}_{33}$ and the refined version $\mathcal{S}' =_{def} \mathcal{S}_{43}$ is obtained by replacing the transition

$$\tau : \ (p_0,\ (inP, T) := (true, q),\ p_w),$$

by the transitions

$$\tau' : \ (p_0,\ inP := true,\ p_i),$$
$$\tau'' : \ (p_i,\ T := q,\ p_w).$$

As usual, the invariant $I =_{def} I_{33}$ is split into

$$I^- : \ (at\ p_w \Rightarrow (T = q \ \vee\ at\ q_w)) \wedge (at\ q_w \Rightarrow (T = p \ \vee\ at\ p_w)) \wedge$$
$$(at\ p_c \Rightarrow (T = p \ \vee\ \neg inQ)) \wedge$$
$$(at\ p_0 \equiv \neg inP) \wedge (at\ q_0 \equiv \neg inQ) \wedge \neg at\ q_i,$$

$$I^+ : \ (at\ q_c \Rightarrow (T = q \ \vee\ \neg inP)) \wedge \neg at\ p_i.$$

Formulas K_D and K_U are easily computed; once again, the fact that I^- is always true in relevant states (it is an invariant of I_{43}) induces some simplification. The results are

$$\begin{aligned} K_D &= \neg I^+ \\ &= (at\ q_c \wedge T = p \wedge inP) \vee at\ p_i, \end{aligned}$$

$$\begin{aligned} K_U &= \neg I^+ \wedge slp[I^+;\ T' \setminus T] \\ &= (\neg I^+ \wedge slp[I^+;\ (p_0 \rightarrow p_i)]) \vee (\neg I^+ \wedge slp[I^+;\ (p_i \rightarrow p_w)]) \\ &= (\neg I^+ \wedge at\ p_i \wedge slp[I^+[at\ p_0];\ inP := true]) \vee (\neg I^+ \wedge at\ p_w \wedge slp[I^+[at\ p_i];\ T := q]) \\ &= (at\ p_i \wedge slp[true;\ inP := true]) \vee (at\ q_c \wedge T = p \wedge inP \wedge at\ p_w \wedge slp[false;\ T := q]) \\ &= (at\ p_i \wedge inP) \vee false \\ &= (at\ p_i \wedge \neg at\ p_0) \\ &= at\ p_i. \end{aligned}$$

We have to select a K within the set

$$\mathcal{K}_0 \ = \ \{X \ : \ [at\ p_i \Rightarrow X] \wedge [X \Rightarrow ((at\ q_c \wedge T = p \wedge inP) \vee at\ p_i)]\}.$$

Once again, $K = K_U$ is an appropriate choice, such that $\{K\}\, t\, \{I^+ \vee K\}$ holds for each transition t of the refined system. The formula $(I^+ \vee K)$ reduces to $[(at\ q_c \Rightarrow (T = q \vee \neg inP)) \vee at\ p_i]$, and further to $[at\ q_c \Rightarrow (T = q \vee \neg inP \vee at\ p_i)]$. The formula $I' =_{def} [I^- \wedge (I^+ \vee K)]$ is

$$\begin{aligned} I_{43} =_{def}\ & (at\ p_w \Rightarrow (T = q \ \vee\ at\ q_w)) \wedge (at\ q_w \Rightarrow (T = p \ \vee\ at\ p_w)) \\ & \wedge (at\ p_c \Rightarrow (T = p \ \vee\ \neg inQ)) \wedge (at\ q_c \Rightarrow (T = q \ \vee\ \neg inP \ \vee\ at\ p_i)) \\ & \wedge (at\ p_0 \equiv \neg inP) \wedge (at\ q_0 \equiv \neg inQ) \\ & \wedge \neg at\ q_i. \end{aligned}$$

A similar sequential refinement leads to $\mathcal{S}_{44}$; an invariant of this system is

$$\begin{aligned} I_{44} =_{def}\ & (at\ p_w \Rightarrow (T = q \ \vee\ at\ q_w)) \wedge (at\ q_w \Rightarrow (T = p \ \vee\ at\ p_w)) \\ & \wedge (at\ p_c \Rightarrow (T = p \ \vee\ \neg inQ \ \vee\ at\ q_i)) \\ & \wedge (at\ q_c \Rightarrow (T = q \ \vee\ \neg inP \ \vee\ at\ p_i)) \\ & \wedge (at\ p_0 \equiv \neg inP) \wedge (at\ q_0 \equiv \neg inQ), \end{aligned}$$

and the mutual exclusion is still satisfied.

Comments. In [18,19], we proposed a slightly different technique; for the first sequential refinement, we could have chosen $R =_{def} \neg at\ p_i$ instead of $R =_{def} I^+$. One can check that both choices lead to the same refined invariant I_{43}.

One can also check that reverting the order in the sequential refinement, that is, replacing the transition

$$\tau : \ (p_0,\ (inP, T) := (true, q)\ ,\ p_w)\ ,$$

by the transitions

$$\mu' : \ (p_0,\ T := q\ ,\ p_i),$$
$$\mu'' : \ (p_i,\ inP := true\ ,\ p_w),$$

leads to an unsatisfactory version (see [39,15,17]).

5 Development of a data transfer protocol

Even the development of small to medium-sized systems involves rather many steps but most of them reduce to routine symbolic manipulation. The interesting, non trivial steps are concerned with the specific algorithmic idea(s) of the system in development.

We address in this section the development of a variant of Stenning's data transfer protocol [42]. In order to keep the presentation within a short size, only the crucial step of the design will be considered (see [20] for the complete development).

5.1 The initial version

A data transfer protocol must ensure reliable transmission of information from a station to another. The problem is that transmission channels are not safe: they can lose, corrupt, duplicate and reorder messages. It is assumed that any corruption is detected by the receiving station, which simply discards corrupted messages. The sequence of data is to be transmitted without loss, alteration, duplication or permutation.

The information to be sent along is represented by a sequence $X =_{def} (X[n] : n = 1, 2, \ldots)$ of messages, whereas a similar sequence Y records the already (and correctly) transmitted part of X. A simple transmission strategy is as follows. The sending station repeatedly transmits a message $X[n]$. The receiving station discards corrupted copies of $X[n]$ until a correct copy is received; then, "acknowlegments" are repeatedly sent back to the sending station. Upon receiving such an acknowledgment, the sending station can begin to send copies of the next message $X[n+1]$.

Our initial version will implement this elementary strategy, at a rather abstract level: first, no process is introduced (like in UNITY [10], a formal concurrent system can be represented by a set of global actions) and, second, the (unreliable) channels are supposed to be synchronous. Three counters are used. The sequence $X[1 : LA]$ has been successfully transmitted and acknowledged ("LA" means "Last Acknowledged"); the sequence $X[1 : LR]$ has been correctly received ("LR" means "Last Received") and the sequence $X[1 : HS]$ has been transmitted ("HS" means "Highest Sent"). As a consequence of the transmission stategy, the relation $LA \leq LR \leq HS \leq LA+1$ is maintained. Here is the formal representation of the initial system $\mathcal{S}_0 = (\mathcal{P}_0, \mathcal{M}_0, \mathcal{T}_0)$.

$$\mathcal{P}_0 \ = \ \emptyset\,;$$

$$\mathcal{M}_0 \ = \ \{LA, LR, HS : nat;\ \ X, Y : array[nat]\ of\ string\}\,;$$

$$\begin{aligned}
\mathcal{T}_0 \ = \ \{ & 1.\ \ (LA = HS \ \longrightarrow \ (HS, Y[HS+1]) := (HS+1, X[HS+1]))\,,\\
& 2.\ \ (LA = HS \ \longrightarrow \ HS := HS+1)\,,\\
& 3.\ \ (LA < HS \ \longrightarrow \ Y[HS] := X[HS])\,,\\
& 4.\ \ (LA < HS \ \longrightarrow \ skip)\,,\\
& 5.\ \ (Y[LR+1] \neq NIL \ \longrightarrow \ (LA, LR) := (LA+1, LR+1))\,,\\
& 6.\ \ (Y[LR+1] \neq NIL \ \longrightarrow \ LR := LR+1)\,,\\
& 7.\ \ (Y[LR+1] = NIL \ \longrightarrow \ LA := LR)\,,\\
& 8.\ \ (Y[LR+1] = NIL \ \longrightarrow \ skip)\}\,.
\end{aligned}$$

The invariant is

$$\begin{aligned} I_0 =_{def} \quad & (LA \leq LR \leq HS \leq LA+1) \wedge \\ & \forall s\,(Y[s] \in \{X[s], NIL\}) \wedge \\ & \forall s\,(1 \leq s \leq LR \Rightarrow Y[s] = X[s]) \wedge \\ & \forall s\,(HS < s \Rightarrow Y[s] = NIL)\,, \end{aligned}$$

and simply expresses the strategy informally introduced above. Any state satisfying I_0 is an adequate initial state; the standard initial state is characterized by

$$C_0 =_{def} \quad LA = LR = HS = 0 \wedge \forall s\,(0 < s \Rightarrow Y[s] = NIL)\,.$$

Transitions 1 and 2 model message transmission. When message $X[HS]$ has been acknowledged, message $X[HS+1]$ can be transmitted; this transmission can succeed (transition 1) or fail (transition 2). The value NIL is a "dummy" value; it models the initial empty value of elements of the recording sequence Y, and also any corrupted value. As a result, unsuccessful transmission is simply modelled by *skip*.
If message $X[HS]$ has not been acknowledged within some delay (this delay is not modelled here), it has to be transmitted again, and retransmission can succeed (transition 3) or fail (transition 4).
When a new, uncorrupted message arrives, it is acknowledged; the transmission of the acknowledgment can succeed (transition 5) or fail (transition 6). The acknowledment corresponding to some message is sent repeatedly, until the next message arrives. The acknowledgment retransmission can succeed (transition 7) or fail (transition 8).

5.2 Correctness of the initial version

The invariant expresses that, at every time, the prefix $X[1 : LR]$ of the sequence has been correctly transmitted; it is also clear that the counters LA, LR and HS cannot decrease. These properties are the interesting invariance properties of the system. Although we are mainly concerned by invariance properties, let us also mention an interesting liveness property of the system: the counters increase, provided that some fairness requirements are satisfied. This property is formally specified and proved in [20]; only a graphical proof outline is given here (Fig. 2), with the following notation:

$$\begin{aligned} A_n &: \; LA = LR = HS = n\,, \\ B_n &: \; LA = LR = n \wedge HS = n+1 \wedge Y[HS] = X[HS]\,, \\ C_n &: \; LA = LR = n \wedge HS = n+1 \wedge Y[HS] = NIL\,, \\ D_n &: \; LA = n \wedge LR = HS = n+1\,. \end{aligned}$$

The self loops in Fig. 2 correspond to useless moves, whereas the other arcs correspond to useful moves. Fairness requirements are that self looping cannot last forever. These requirements can be informally expressed as follows.

The receiver cannot delay the sender forever, (loop A),
The sender cannot delay the receiver forever, (loop B),
The transmission channel cannot fail forever, (loop C),
The acknowledgment channel cannot fail forever, (loop D).

5.3 Stenning's window principle

The next step will be the implementation of Stenning's technique. For now, the transmitter may be ahead of the receiver, but only by one message. The synchrony between the transmitter and the receiver can be decreased by allowing a "window" of already sent but still unacknowledged messages. Let $W \geq 1$ be the finite maximal size of the window. Notice that the candidates for retransmission are no longer $X[HS]$ only, but each $X[r]$ such that $LA < r \leq HS$. All the messages belonging to this

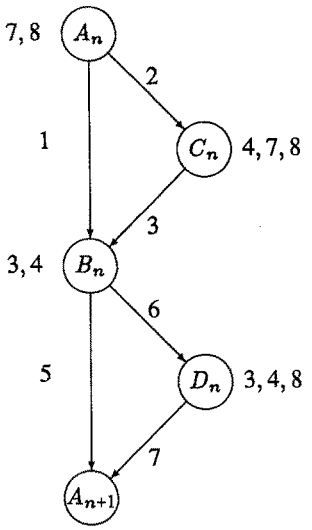

Figure 2: Graphical proof outline

window will be retransmitted after some delay. (In our model, the delay is left unspecified; any finite delay is acceptable, as far as only correctness is concerned.) A preliminary refinement is needed to introduce a window index r; that leads to the system S_1, with invariant $I_1 = I_0$. The window principle is now introduced by a semantical refinement. The set of transitions of the resulting system S_2 is

$$
\begin{aligned}
T_2 = \{ & 1.\ (HS - LA < W \longrightarrow (HS, Y[HS+1]) := (HS+1, X[HS+1])), \\
& 2.\ (HS - LA < W \longrightarrow HS := HS+1), \\
& 3.\ (LA < r \leq HS \longrightarrow (Y[r], r) := (X[r], r+1)), \\
& 4.\ (LA < r \leq HS \longrightarrow r := r+1), \\
& 5.\ (\neg(LA < r \leq HS) \longrightarrow r := LA+1), \\
& 6.\ (Y[LR+1] \neq NIL \longrightarrow (LA, LR) := (LA+1, LR+1)), \\
& 7.\ (Y[LR+1] \neq NIL \longrightarrow LR := LR+1), \\
& 8.\ (Y[LR+1] = NIL \longrightarrow LA := LR), \\
& 9.\ (Y[LR+1] = NIL \longrightarrow skip)\}.
\end{aligned}
$$

Transitions 1 and 2 correspond to first message transmission, transition 3, 4 and 5 implement message retransmission, transitions 6 and 7 correspond to message reception and acknowledgment, and transitions 8 and 9 implement acknowledgment retransmission.

As usual, some parts of the initial invariant are respected by the new transitions but others are not; this gives rise to the decomposition $I_1 =_{def} (I^- \wedge I^+)$, where

$$
\begin{aligned}
I^- =_{def}\ & (LA \leq LR \leq HS) \wedge \\
& \forall s\,(Y[s] \in \{X[s], NIL\}) \wedge \\
& \forall s\,(1 \leq s \leq LR \Rightarrow Y[s] = X[s]) \wedge \\
& \forall s\,(HS < s \Rightarrow Y[s] = NIL),
\end{aligned}
$$

$$
I^+ =_{def}\ HS \leq LA + 1.
$$

Formulas K_D and K_U are determined as follows.

$$\begin{aligned} K_D &\equiv \neg I^+ \\ &\equiv HS > LA+1. \end{aligned}$$

$$\begin{aligned} K_U &\equiv \neg I^+ \wedge slp[I^+;\ T' \setminus T] \\ &\equiv HS > LA+1 \wedge slp[HS \leq LA+1;\ \{1,2,3,4,5\}] \\ &\equiv HS > LA+1 \wedge (slp[HS \leq LA+1;\ \{1,2\}] \vee slp[HS \leq LA+1;\ \{3,4,5\}]) \\ &\equiv HS > LA+1 \wedge (HS \leq LA+2 \wedge HS \leq LA+W \vee HS \leq LA+1) \\ &\equiv HS = LA+2 \wedge HS \leq LA+W. \end{aligned}$$

(In this computation, I^- is assumed to be true and only relevant variables have been considered.) The choice $K =_{def} K_U$ is too strong but, as K_U is a conjunction, it is natural to try one of the conjunct. An adequate choice is $(I^+ \vee K) =_{def} HS \leq LA+W$. The resulting invariant is

$$\begin{aligned} I_2 =_{def}\ & (LA \leq LR \leq HS \leq LA+W) \wedge \\ & \forall s\,(Y[s] \in \{X[s], NIL\}) \wedge \\ & \forall s\,(1 \leq s \leq LR \Rightarrow Y[s] = X[s]) \wedge \\ & \forall s\,(HS < s \Rightarrow Y[s] = NIL). \end{aligned}$$

The correctness proof given for S_0 is easily adapted to S_2.

6 Related work and further work

The abstract programming language used in this paper is based on the classical notion of a transition system, frequently used in parallel programming (see e.g. [25,33]). Our variant has been obtained by adding labels to the formalism presented in [41]. The adaptation of Hoare logic to this language is similar in spirit to the adaptations for concurrency presented in [26,28] (although some technical differences exist). The rules (3) comes from [16].

The stepwise refinement approach has been widely used in parallel programming. Early contributions are mainly concerned with the refinement of the grain of parallelism [14]; [27] introduces the concept of a state function, which connects the invariant of the refined version with the invariant of the initial version. This concept can lead to concise proofs, but the discovery of appropriate state functions is not always trivial.

In this paper, a (semantical) refinement is the replacement of an old transition by a set of new transitions. This definition is too general, since any transformation can be achieved by a sequence of such replacements. However, the distinction between a refinement and a general transformation is somewhat arbitrary; several authors have demonstrated that many non-trivial transformations of concurrent systems can be considered as refinements [43,4,9]. The notion of refinement deserves further theoretical study.

Mainly invariance properties have been considered in this paper. Temporal logic has been used to specify and prove other kinds of program properties [34,35]. We have observed that when a refined version maintains a liveness property, the proof of this property is easily adapted, but this is not a general result. A refinement method taking into account the liveness properties has been proposed in [7], but it is restricted to atomicity refinements. Another weakness of the method proposed here is that it is sound but not complete [12]. More precisely, if the invariant of the refined version does not imply some wanted invariance property, one cannot deduce that this property does not hold. From the theoretical point of view, completeness can be achieved by using predicate transformers like the "weakest safe invariant" introduced in [43], or the "weakest invariant" introduced in [29]. The problem is that these predicate transformers are not easily computed. However, this approach seems interesting for finite state systems since, in this case, weakest and strongest invariants can be computed in a mechanical way.

References

[1] E.A. ASHCROFT and Z. MANNA, "Formalization of Properties of Parallel Programs", Machine Intelligence, **6**, pp. 17-41, 1970

[2] K.R. APT, "Ten years of Hoare logic", ACM Toplas, **3**, pp. 431-483, 1981

[3] K.R. APT, "Correctness Proofs of Distributed Termination Algorithms", ACM Toplas, **8**, pp. 388-405, 1986

[4] R.J.R. BACK and R. KURKI-SUONIO, "Decentralization of Process Nets with Centralized Control", Proc. 2nd ACM Symp. on Principles of Distributed Computing, pp. 131-142, 1983

[5] R.J.R. BACK and R. KURKI-SUONIO, "Distributed Cooperation with Action Systems", ACM Toplas, **10**, pp. 513-554, 1988

[6] R.J.R. BACK, "A Calculus of Refinements for Program Derivations", Acta Informatica, **25**, pp. 593-624, 1988

[7] R.J.R. BACK, "A Method for Refining Atomicity in Parallel Algorithms", LNCS, **366**, pp. 199-216, 1989

[8] E. BEST, "A Note on the Proof of a Concurrent Program", IPL, **9**, pp. 103-104, 1979

[9] M. CHANDY and J. MISRA, "An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection", ACM Toplas, **8**, pp. 326-343, 1986

[10] M. CHANDY and J. MISRA, "Parallel Program Design: A Foundation", Addison-Wesley, 1988

[11] J. de BAKKER, "Mathematical theory of program correctness", Prentice-Hall, 1980

[12] J.W. de BAKKER and L.G.L.T. MEERTENS, "On the completeness of the inductive assertion method", JCSS, **11**, pp. 323-357, 1975

[13] E.W. DIJKSTRA, "A discipline of programming", Prentice Hall, New Jersey, 1976

[14] E.W. DIJKSTRA and al., "On-the-Fly Garbage Collection: An Exercise in Cooperation", CACM, **21**, pp. 966-975, 1978

[15] E.W. DIJKSTRA, "An assertional proof of a program by G.L. Peterson", EWD 779, 1981

[16] R. GERTH, "Transition logic", Proc. 16th ACM Symp. on Theory of Computing, pp. 39-50, 1984

[17] E.P. GRIBOMONT, "Synthesis of parallel programs invariants", LNCS, **186**, pp. 325-338, 1985

[18] E.P. GRIBOMONT, "Development of concurrent programs : an example" LNCS, **352**, pp. 210-224, 1989

[19] E.P. GRIBOMONT, "Stepwise refinement and concurrency : a small exercise" LNCS, **375**, pp. 219-238, 1989

[20] E.P. GRIBOMONT, "Stenning's protocol", in "Formal methods for parallel programming", Internal report, 1989

[21] D. GRIES, "The Science of Programming", Springer-Verlag, Berlin, 1981

[22] C.A.R. HOARE, "An axiomatic basis for computer programming", CACM, **12**, pp. 576-583, 1969

[23] C.A.R. HOARE, "Communicating Sequential Processes", CACM, **21**, pp. 666-677, 1978

[24] C.A.R. HOARE, "Communicating Sequential Processes", Prentice-Hall, 1985

[25] R.M. KELLER, "Formal Verification of Parallel Programs", CACM, **19**, pp. 371-384, 1976

[26] L. LAMPORT, "The 'Hoare Logic' of Concurrent Programs", Acta Informatica, **14**, pp. 21-37, 1980

[27] L. LAMPORT, "An Assertional Correctness Proof of a Distributed Algorithm", SCP, **2**, pp. 175-206, 1983

[28] L. LAMPORT and F.B. SCHNEIDER, "The 'Hoare Logic' of CSP, and All That", ACM Toplas, **6**, pp. 281-296, 1984

[29] L. LAMPORT, "*win* and *sin*: Predicate Transformers for Concurrency", DEC SRC Report 17, 1987

[30] L. LAMPORT, "A Theorem on Atomicity in Distributed Algorithms", DEC SRC Report 28, 1988

[31] R.J. LIPTON, "Reduction: a method of proving properties of parallel programs", CACM, **18**, pp. 717-721, 1975

[32] N.A. LYNCH and M.R. TUTTLE, "Hierarchical Correctness Proofs for Distributed Algorithms", Proc. 6th ACM Symp. on Principles of Distributed Computing, pp. 137-151, 1987

[33] Z. MANNA and A. PNUELI, "How to cook a temporal proof system for your pet language", Proc. 10th ACM Symp. on Principles of Programming Languages, pp. 141-154, 1983

[34] Z. MANNA and A. PNUELI, "Adequate proof principles for invariance and liveness properties of concurrent programs", SCP, **4**, pp. 257-289, 1984

[35] Z. MANNA and A. PNUELI, "Specification and verification of concurrent programs by $\forall$-automata", Proc. 14th ACM Symp. on Principles of Programming Languages, pp. 1-12, 1987

[36] C. MORGAN, "The Specification Statement", ACM Toplas, **10**, pp. 403-419, 1988

[37] J.M. MORRIS, "A theoretical basis for stepwise refinement and the programming calculus", SCP, **9**, pp. 287-306, 1987

[38] S. OWICKI and D. GRIES, "An Axiomatic Proof Technique for Parallel Programs", Acta Informatica, **6**, pp. 319-340, 1976

[39] G.L. PETERSON, "Myths about the mutual exclusion problem", IPL, **12**, pp. 115-116, 1981

[40] R.D. SCHLICHTING and F.D. SCHNEIDER, "Using Message Passing for Distributed Programming: Proof Rules and Disciplines", ACM Toplas, **6**, pp. 402-431, 1984

[41] J. SIFAKIS, "A unified approach for studying the properties of transition systems", TCS, **18**, pp. 227-259, 1982

[42] N.V. STENNING, "A data transfer protocol", Computer Networks, **1**, pp. 99-110, 1976

[43] A. van LAMSWEERDE and M. SINTZOFF, "Formal derivation of strongly correct concurrent programs", Acta Informatica, **12**, pp. 1-31, 1979