Proof Rules for Fault Tolerant Distributed Programs

Mathai Joseph
Abha Moitra
Neelam Soundararajan
TR 84-643
October 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853
ABSTRACT

Proving properties of fault tolerant distributed programs is a complex task as such proofs must take into account failures at all possible points in the execution of individual processes. The difficulty in accomplishing this is compounded by the need also to cater for simultaneous failures of two or more processes. In this paper, we consider programs written in a version of Hoare's CSP and define a set of axioms and inference rules by which proofs can be constructed in three steps: proving the properties of each process when its communicants are prone to failure, establishing the effects of failure of each process, and combining these proofs to determine the fault tolerant properties of the whole program.
1. Introduction

In some earlier work [3,4], we had studied the problem of building fault-tolerant distributed programs and shown that it is possible to prove the properties of such programs when they are subject to a variety of failures. Programs were written in an slightly extended version of CSP [2] and it was assumed that faults in a process or in a communication channel could be detected, and that recovery could be initiated by 're-making' a faulty process, causing it to 'repair' its channels and resume execution from its initial state. In general, recovery required the cooperative action of several processes and could then be performed without the use of stable storage [5]. The proof techniques were based on the proof system devised by Apt, Francez and de Roever [1], using local invariants for individual processes, a global invariant for the whole program and a proof of cooperation.

Apart from their use in proving properties of fault-tolerant programs, local and global invariants can be analyzed to provide guidelines for constructing fault-tolerant versions of correct distributed programs [4]. Such guidelines are quite useful for simple programs, but we have found that they do not always provide enough information to help in constructing larger and more complex fault-tolerant programs. Informally, we can say that local and global invariants allow characterization of the 'low level' behaviour of a program and that this is sometimes inadequate when 'higher level' properties of a fault-tolerant program must be established; for example, a global invariant provides links between communication commands in pairs of processes but it is difficult to both represent the effects of failure in one process on other processes and account for the simultaneous failure of more than one process.
Consider a program $S_1$ in which process $R_1$ sends process $R_2$ an ascending sequence of integers and $R_2$ sums successive pairs of integers and sends the results to process $R_3$ to be printed.

$$S_1 :: [ R_1 :: i := 1; *[{\text{LI}(R_1)}] R_2 ! i \rightarrow i := i + 1 ]$$

$$| |$$

$$R_2 :: R_1 ? x; *[{\text{LI}(R_2)}] R_1 ? y \rightarrow R_3 ! (x + y); R_1 ? x]$$

$$| |$$

$$R_3 :: *[{\text{LI}(R_3)}] R_2 ? t \rightarrow \text{"print t"}$$

Assume process $R_2$ fails (and recovers) and that this failure is detected by $R_1$. If the failure is detected when $i > 2$ and even, the local invariants $\text{LI}(R_1)$, $\text{LI}(R_2)$ and $\text{LI}(R_3)$ and a suitable global invariant will permit the inference that the sum of the two integers ($i$-3) and ($i$-2) has been sent by $R_2$ to $R_3$. But if the value of $i$ is odd, no assertion can be made about whether $R_2$ failed after or before it communicated the new sum to $R_3$. In a fault tolerant version of the program, $R_1$ must then assume the worst and decrement $i$ by 2 before re-sending values to $R_2$. And this can lead to the same value being printed twice (or more often, if there are repeated failures in $R_2$). Unfortunately, even with as few as three processes, the reasoning that leads to this conclusion must be based on properties of the whole program and the assertion describing this condition is no longer simple. It can easily be seen that the situation can get rapidly out of hand as the number of processes increases.

If $R_3$ is modified to print only those values received from $R_2$ that are in strictly ascending order, such repetitions can be avoided. But if $R_2$ and $R_3$ fail
simultaneously, i.e. within the same interval of time, it can no longer be assured that values are printed in this order and we must accept a weaker global invariant that admits the possibility of repetitions. In general, in all such cases we must either rely on the intuition of the programmer in specifying these effects and their combinations correctly, or look for a methodology by which the effects can be systematically derived. Even a limited acquaintance with the construction of fault tolerant programs will show that intuition, by itself, is not sufficient: we have found building such programs and proving their properties to be an extremely complex task unless it is supported by a sound methodology.

We have therefore been investigating other techniques for constructing fault-tolerant programs. A more promising approach than dealing with the program as a whole would be to decompose the main problem into the sub-problems of:

a. Specifying the behaviour of each process when its communicants fail and are re-made,
b. Specifying the effects of failure and re-making for each process, and
c. Combining these specifications to prove the properties of the program as a whole.

Steps (a) and (b) can then be performed locally (i.e., in isolation) for each process and step (c) requires the use of proof rules for communication between processes. It should be noted that the properties of the same program with and without failures can be rather different and this should emerge from the application of step (c), rather than from informal reasoning by the programmer.
The CSP proof system of Soundararajan [8] follows similar steps in reasoning about programs: properties of individual processes are proved in isolation and then combined by a rule of parallel composition where 'compatibility' between the assumptions made in each process about communications from other processes is established. We shall extend the axioms and inference rules of this proof system for our purposes, adding clauses to account for detected failures, and then prove the fault-tolerant properties of a simple program.

2. A System Model

Assume every process in a CSP program executes on a separate processing node, and that the nodes are interconnected by a communication medium. A node consists of processors and memory, and the processors independently execute the instructions for a process. Failure of a node occurs when there is a discrepancy in the actions of the processors: when there are two processors, this can be detected by 'matching' circuits, and if there are three processors the error is detected by simple majority logic. For our purposes, the processors need not execute in step as it is only necessary that errors be detected at synchronization points, i.e. before communication takes place with another node. When an error is detected in a node, execution of the resident process must cease and the node will fail to respond to any communication requests. A failed node (and its resident process) is said to be 'withdrawn'. (A node with these properties is similar to the 'fail-stop' processors of [7]).

A process attempting to communicate with another process on a withdrawn node will receive an error signal, so failures of a node can be detected by attempts at
communication from other nodes. This simple model is adequate to deal with failure of a single node at a time. But consider what will happen if two (or more) communicating nodes fail the same interval of time. In this case, the multiple failures of a set of nodes will not be detected unless some other node attempts communication with the failed nodes. It is therefore necessary to add to the system an independent means of failure detection and this is done using a communication checker process executing on a separate node. The communication checker regularly interrogates each node and detects a failure by the lack of (or an incorrect) response from a node. A process $P_0$ will be designated as the communication checker.

The communication checker has an additional function: when it detects failure of a node, it re-starts execution of the resident process on a new node. Thus we assume there is adequate hardware redundancy, and that the communication checker keeps a table of the status of all nodes and of the assignment of processes to nodes. To complete the model, we must also assume there are a number of communication checkers (one master and several standbys) each in communication with the others, so that failure of a communication checker does not compromise the reliability of the whole system. In general, failure of a communication checker must be determined using a ‘Byzantine general’ solution [6] and we shall assume this is done by some underlying mechanism.

The system model outlined here has been kept as simple as possible, because the purpose of this paper is really to examine issues of fault tolerance at the program level. Nevertheless, it is useful to know that a number of implementations of such a model are possible, e.g. on a shared memory system, or on a local area network where the communication interface of each node stores the identity of
the process executing at the node. A generous amount of hardware redundancy is required, but its actual extent will be determined by the degree of failure resilience required.

3. Fault Detection and Recovery

We shall make the following assumptions about failures:

1. All failures in a process are detected by the process $P_0$ and a failed process is 'withdrawn' from active service, i.e., it attempts no further communication, until it is re-made; one way to ensure such fault detection is to execute each process on a 'fail-stop' processor [7].

2. A failed process is re-made by a re-make command executed by communication checker process $P_0$ and a re-made process resumes execution from its initial state. (A technique for implementing this is described in [3].) At times it may be more desirable to use checkpoints and to re-start the execution of a failed process from a previous checkpoint; it will turn out that the proof rules we will develop can be easily tailored to handle this model.

3. The communication statements of CSP are altered to deal with failures. Using the symbol '.' to mean either input ('?') or output ('!'), a statement in process $P_1'$ to communicate with process $P_2'$ appears as

$$P_2'.x <<S'>>$$

and in a guarded command as

$$b; P_2'.x \rightarrow S <<S'>>$$

where $b$ is a boolean guard.
As in CSP, a communication command such as $P_2'.x$ may be selected for execution in two ways: deterministically, as in (1), and non-deterministically (subject to $b$ evaluating to true), as in (2). When such a statement is selected for execution in a correct program, there are two possibilities:

(a) Process $P_2'$ may be ready to reciprocate the communication, and the command $P_2'.x$ is executed,

(b) Process $P_2'$ may have failed since it last communicated with $P_1'$, and the statement $<<S'>>$ is executed: if $P_2'$ is still in a failed state, it is implicitly re-made before $S'$ is executed.

Note that exactly one of these possibilities will be selected: either $P_2'.x$ is correctly executed, or $<<S'>>$ is executed. In our further discussion, we shall call $<<S'>>$ the 'fault' alternative. To keep the control flow simple, we do not allow any communication statement to be included in a fault alternative.

To illustrate the use of such statements, consider the following example of the well known bounded buffer. A producer process $P_1'$ sends a sequence of numbered lines to a process $P_2'$ with a buffer of size $n$. The buffer process in turn sends a sequence of lines to the consumer process $P_3'$ which sends each line to a printer process $P_4'$. 
\[ S_2 :: [P_1' \mid | P_2' \mid | P_3' \mid | P_4'] \]

\[ P_1' :: \text{pseq : integer; ready : boolean; nextline : line;} \]
\[ \text{pseq}:=0; \text{ready}:=\text{false}; \]
\[ \text{if ready} \rightarrow \text{nextline}:=\text{Line(pseq+1)}; \text{ready}:=\text{true} \]
\[ \text{ready; } P_2' \mid (\text{nextline, pseq+1}) \rightarrow \text{pseq}:=\text{pseq}+1; \text{ready}:=\text{false} \]

\[ P_2' :: \text{in, out : integer;} \]
\[ \text{A : [0..(n-1)] of record ln : line; linenum : integer end;} \]
\[ \text{in}:=0; \text{out}:=0; \]
\[ \text{if in < out + n-1; } P_1'?A[\text{in mod n}] \rightarrow \text{in}:=\text{in+1} \]
\[ \text{out < in; } P_3' \mid A[\text{out mod n}] \rightarrow \text{out}:=\text{out+1} \]

\[ P_3' :: \text{ln : line; num : integer;} \]
\[ \text{if } P_2' \mid \text{ln, num} \rightarrow P_4' \mid \text{ln} \]

\[ P_4' :: \text{ln : line;} \]
\[ \text{if } P_3' \mid \text{ln} \rightarrow \text{skip} \]

Let us assume for simplicity that the producer process, \( P_1' \), and the printer process, \( P_4' \), never fail. A failure in the buffer process, \( P_2' \), will be detected by \( P_1' \) and \( P_3' \) and will cause the loss of \((\text{out} - \text{in})\) lines. As \text{out} and \text{in} are local variables of \( P_2' \), it must then be assumed by \( P_1' \) that up to \( n \) lines may be lost. Thus, a solution is for \( P_1' \) to re-send these lines to \( P_2' \). If, in fact, \((\text{out} - \text{in})\) were less than \( n \) at the time of failure, this may lead to up to \( n \) repetitions in
lines sent to $P_3'$. To suppress such duplicate lines, $P_3'$ can be altered to forward to $P_4'$ only those lines that have numbers in strictly ascending order; e.g., if $lastnum$ is the number of the last line printed, the next line to be printed must have $num > lastnum$. Failure of $P_3'$ will be detected by $P_2'$ and $P_4'$ and may result in the loss of a line taken from $P_2'$ but not yet printed. $P_2'$ must therefore make provision for repeating the last line sent to $P_3'$ and, if this line had been printed, this will lead to a duplicated line. The last case to be considered is when $P_2'$ and $P_3'$ fail within an interval of time such that the value of $lastnum$ in $P_3'$ is lost and duplicate lines sent from $P_2'$ reach the printer.

This informal analysis suggests that failure of $P_2'$ and $P_3'$ will, in general, lead to duplicated printing. The extent of such duplication depends on the points of failure, on the number of individual failures in $P_2'$ and $P_3'$, and on the interaction between the effects of failures in $P_2'$ and $P_3'$. We shall later formally prove the fault tolerant properties of $S_3$, which is a fault-tolerant version of $S_2$. 
$S_3 :: [P_1 \mid \cdot P_2 \mid \cdot P_3 \mid \cdot P_4$

$P_1 :: pseq, sent : integer; ready : boolean; nextline : line;$

\[ pseq:=0; \; sent:=0; \; ready:=false; \]

\[ \square \]

\[ \neg ready \rightarrow nextline:=Line(pseq+1); \; ready:=true \]

\[ \square \]

\begin{align*}
\text{ready; } P_2 \mid \cdot (\text{nextline, pseq+1}) & \rightarrow pseq:=pseq+1; \; ready:=false; \\
\text{sent:=sent+1} & \rightarrow \neg ready:=false; \\
\end{align*}

\[ \square \]

\[ \begin{align*}
\text{sent} & < n \rightarrow pseq:=pseq - sent \\
\text{sent} & \geq n \rightarrow pseq:=pseq - n \\
\end{align*} \]

\[ \square \]

\[ \begin{align*}
; \; \text{sent:=0} & \rightarrow \square \]

$P_2 :: \text{in, out : integer; output : boolean;}$

A : [0..(n-1)] of record ln : line; linenumber : integer end;

\[ \begin{align*}
in:=0; \; out:=0; \; output:=false; \\
\text{in < out+ n - 2; } P_1?A[in \mod \; n] & \rightarrow in:=in+1 \\
\text{out < in; } P_3 \mid \cdot A[\text{out mod n}] & \rightarrow out:=out+1; \; output:=true \\
\end{align*} \]

\[ \square \]

\[ \neg output \rightarrow \square \]

\[ \begin{align*}
\neg output \rightarrow \square \\
\text{; output:=false} & \rightarrow \square \\
\end{align*} \]
$P_3 ::= \text{ln : line; num,lastnum : integer;}
\text{lastnum}:=0;
\quad *[P_2? \text{ln,num} \rightarrow \text{[num > lastnum} \rightarrow P_4 \text{! ln; lastnum}:=\text{num}}
\quad \Box
\quad \text{num} \leq \text{lastnum} \rightarrow \text{skip}
\quad ]
\quad \langle\langle \text{skip} \rangle\rangle
\]

$P_4 ::= \text{ln : line;}
\quad *[P_3? \text{ln} \rightarrow \text{skip}
\quad \langle\langle \text{skip} \rangle\rangle
\]

Note that the program takes into account the possibility of repeated failures of $P_2$ and $P_3$ at all points in their execution, including the cases where they fail more than once before engaging in any communication.

4. Communication Sequences

Let every communication in an (extended) CSP program be characterised by a triple of the form $<i,j,m>$, where $i$ is an integer index for the sender process, $j$ a similar index for the receiver process and $m$ the communication (or message). Thus the communication resulting from the simultaneous execution of the statement $P_3!5$ in process $P_1$ and $P_1?x$ in process $P_3$ would be represented by the triple $<1,3,5>$. We shall refer to communications resulting from the execution of input and output commands as 'explicit' communications.
'Implicit' communication takes place without the execution of input and output commands. There are two kinds of implicit communication and we shall characterise them by 'messages' received implicitly by a process:

\[ \delta \] received by \( P_0 \) when some other process \( P_i \) fails

\[ \rho \] received by a process before it communicates with a failed and re-made process

But it should be emphasised that implicit communications do not necessarily represent the transmission of real messages from a sender to a receiver; in particular, the message \( \delta \) originates from the detection of failure in a process by the communication checker process \( P_0 \), rather than from any action assumed to be taken by the failed process. When a process \( P_i \) fails then before any other process \( P_j, j \neq 0 \), can start communication with \( P_i \), \( P_j \) must receive the message \( \rho \).

With every process \( P_i \) of a CSP program, we associate a communication sequence \( h_i \) which consists of triples denoting communications sent to or received by \( P_i \). Thus the execution of the command \( P_3!5 \) in process \( P_1 \) and the command \( P_1?x \) in process \( P_3 \) will result in two identical triples, both equal to \( \langle 1,3,5 \rangle \), being concatenated to the communication sequences of \( P_1 \) and \( P_3 \). A failure of a process \( P_i \) is recorded as \( \langle i,0,\delta \rangle \) in \( h_i \); and before any other process \( P_j \) can communicate with \( P_i \), \( P_j \) records this failure as \( \langle i,j,\rho \rangle \).

The following operations are defined over sequences.
\(|h|\) is the length of the sequence \(h\)

\(h_1 + h_2\) concats \(h_2\) to the end of \(h_1\)

\(h|i\) is the subsequence of all elements of \(h\) which are of the form \(<i,j,m>\) or \(<j,i,m>\)

\(h[k]\) is the \(k^{th}\) element of \(h\) from the beginning

\(h[j : k]\) is the subsequence of \(h\) from its \(j^{th}\) element to its \(k^{th}\) element

\(h_i \subseteq h_j\) if for \(1 \leq k \leq |h_i|\), \(h_i[k] = h_j[k]\)

Much of this has been taken from Soundararajan [8], except for the introduction of implicit communications. We can therefore use the general form of the axioms and rules of inference defined there, with adaptations to deal with the extensions to CSP described above.

5. Axioms and Rules of Inference

Hoare-style proof systems are characterized by rules of the form \(\{p\} S \{q\}\) which are interpreted as saying that if the predicate \(p\) is true before the execution of \(S\), then the predicate \(q\) will be true if and when execution of \(S\) is completed. Consider now the execution of the process \(P_5\):

\[ P_5 ::= [ \text{true} \rightarrow P_6 ! 1; \star[ \text{true} \rightarrow \text{skip}] \]

\[\square\]

\[\text{true} \rightarrow P_6 ! 2 \]
Assume in an execution of $P_5$, the first guarded command is chosen in the alternative statement, so that 1 is output to $P_6$ and $P_5$ then loops in the repetitive command. When considering $P_5$'s normal execution (i.e. without faults), it would still be correct to annotate $P_5$ with the assertions

$$\{h_5 = \epsilon\} P_5 \{h_5 = \langle 5,6,2 \rangle\}$$

because the post-condition is indeed provably true for the only case when $P_5$ does terminate; when $P_5$ does not terminate, a partial-correctness proof system will allow any arbitrary post-condition to be asserted.

That is not the case for the execution of $P_5$ in an environment where faults may appear. Taking the execution of $P_5$ described above, assume $P_5$ fails when executing its repetitive command. When $P_5$ is re-made, let its execution be such that this time the second guarded command is chosen. The sequence $h_5$ will then consist of

$$\langle 5,6,1 \rangle \langle 5,0,\delta \rangle \langle 5,6,2 \rangle$$

The post-condition for $P_5$ should thus be such that it is satisfied by any sequence of partial execution each ending in failure followed by recovery, followed by one ending in termination.

In our proof system, we shall specify axioms and rules of inference corresponding to the normal execution of a program. A final rule will allow us to obtain the behaviour of the failure-prone program from its normal behaviour. To do this, we shall use the notation $(r)\{p\} S \{q\}$ which stands for the following: in the process $P_i$, if $p$ is satisfied initially, then throughout the execution of $S$ the sequence $h_i$ will satisfy $r$, and if and when $S$ terminates, the predicate $q$ will
hold; \( r \) is a predicate over \( h_i \) only, and does not refer to any (other) variables of \( P_i \).

The following axioms and rules of inference refer to statements executed in a process \( P_i \) whose communication sequence is denoted by \( h_i \), \( i > 0 \). We shall not define the actions of process \( P_0 \), which is the communication checker and is assumed to be part of the underlying implementation.

R1. Skip

\[
\frac{\ \ p \Rightarrow r \ \ \ }{ (r \{p\} \text{skip}\{p\})}
\]

R2. Output

\[
\frac{\ \ p \Rightarrow r, p \Rightarrow q_{h_i}^{h_i} + \langle i, j, i, e \rangle , \ q \Rightarrow r \ \ \ }{ (r \{p\} P_j ! e \{q\})}
\]

R3. Input

\[
\frac{\ \ p \Rightarrow r , \ p \Rightarrow \forall m . \ q_{m,h_i}^{x,h_i} + \langle j, i, m \rangle , \ q \Rightarrow r \ \ \ }{ (r \{p\} P_j ? x \{q\})}
\]

R4. Fault Tolerant Communication

\[
(r \{p\} C_j \{q\} \text{ where } C_j \text{ is either } P_j ? x \text{ or } P_j ! e
\]

\[
\frac{\ \ p \Rightarrow q_{h_i}^{h_i} + \langle j, i, i, e \rangle \ \ }{ (r \{q\} S \{q\})}
\]

\[
\frac{\ \ (r \{p\} C_j \langle S \rangle \{q\} \ \ )}{(r \{p\} C_j \langle S \rangle \{q\})}
\]
R5. Assignment

\[
p \Rightarrow r, p \Rightarrow q, q \Rightarrow r
\]

\[
(r) \{ p \} x := e \{ q \}
\]

R6. Composition

\[
p \Rightarrow r, (r) \{ p \} S_1 \{ q' \}, (r) \{ q' \} S_2 \{ q \}
\]

\[
(r) \{ p \} S_1; S_2 \{ q \}
\]

R7. Alternative Command

\[
p \Rightarrow r
\]

\[
(r) \{ p \} A B(g_k) C(g_k); S_k \{ q \}, k = 1,..,m
\]

\[
[p \land B(g_k)] \Rightarrow q^{k_i}_{k_i} + <CP(g_k), i, \rho>,
\]

\[
(r) \{ q^{k_i} \} S_k' \{ q \}, k \in IO
\]

\[
(r) \{ p \} [\square (k = 1,..,m) g_k \rightarrow S_k <<S_k'>> \} \{ q \}
\]

where \( B(g_k) \) is the boolean part and \( C(g_k) \) the communication part of the guard \( g_k \) \((C(g_k) \) is skip if \( g_k \) is purely boolean), \( IO \) is the set of indices of the input/output guards, and \( CP(g_k) \) is the index of the process with which \( P_i \) is trying to communicate in \( C(g_k) \). \(<<S_k'>>\) will be present only if \( g_k \) is an input or output guard.

R8. Repetitive Command

\[
p \Rightarrow r
\]

\[
(r) \{ p \land \bigvee_{k=1}^{m} B(g_k) \} [\square (k = 1,..,m) g_k \rightarrow S_k <<S_k'>> \} \{ p \}
\]

\[
(r) \{ p \} \ast [\square (k = 1,..,m) g_k \rightarrow S_k <<S_k'>> \} \{ p \land \bigvee_{k=1}^{m} \neg B(g_k) \}
\]

Note that to simplify the presentation we assume a loop terminates only when the boolean parts of all the guards evaluate to false.
R9. Consequence

\[
\frac{r' \Rightarrow r, p \Rightarrow p', (r') \{p'\} \ S\{q'\}, \ q' \Rightarrow q}{(r) \{p\} \ S\{q\}}
\]

R10. Strengthening

\[
\frac{(r_1) \{p\} \ S\{q\}, \ (r_2) \{p\} \ S\{q\}}{(r_1 \land r_2) \{p\} \ S\{q\}}
\]

R11. Disjunction

\[
\frac{(r) \{p_1\} \ S\{q\}, \ (r) \{p_2\} \ S\{q\}}{(r)\{p_1 \lor p_2\} \ S\{q\}}
\]

R12. Conjunction

\[
\frac{(r) \{p\} \ S\{q_1\}, \ (r) \{p\} \ S\{q_2\}}{(r) \{p\} \ S\{q_1 \land q_2\}}
\]

6. Failure of a Process

We now need to see how to obtain the behaviour of the failure-prone execution of a process \(P_i\) from the rules given above. Let \([P_i]\) denote such an execution of \(P_i\); \([P_i]\) then consists of a series of partial executions of \(P_i\) which end in failure and re-making of \(P_i\), followed by a final and complete execution. The behaviour of \([P_i]\) can be defined by a rule.
R13. Failure-Prone Process Execution

\[(r) \{p\} P_i \{q\}\]
\[q \Rightarrow q'\]

\[
\frac{[q' \land r_{h_i}^{h_i'}] \Rightarrow q'_{h_i'} + <i,0,\delta> + h_i}{\{p\} [P_i] \{q'\}}
\]

Note that in general, \(h_i\) in \(r_{h_i}^{h_i'}\) and \(q'_{h_i'} + <i,0,\delta> + h_i\) does not refer to the same sequence. This is because all predicates involved in the annotation of process \(P_i\) are described in terms of a general but arbitrary sequence named \(h_i\). We could have written the third clause of R13 as

\[
[q'(h_i) \land r(h_i')] \Rightarrow q'(h_i' + <i,0,\delta> + h_i)
\]

where \(t(H)\) would be defined to be true if and only if the predicate \(t\) is satisfied for the sequence \(H\). We have not adopted this notation since it would make the presentation of most of the other rules much more complicated.

The second clause of R13, \(q \Rightarrow q'\), ensures that the results of the executions of \([P_i]\) that proceed to completion without encountering an error satisfy \([P_i]'\)'s post-condition. The next implication ensures that the results of those executions of \([P_i]\) that encounter \(n+1\) errors before going through one fault-free execution will satisfy the post-condition \(q'\) provided the results of those executions of \([P_i]\) that encounter \(n\) errors satisfy \(q'\). This may be seen as follows.

Consider an execution of \([P_i]\) which encounters \(n+1\) errors, after each of which the process is re-made with its variables set to their initial values and \(<i,0,\delta>\) concatenated to \(h_i\). When the final execution begins, the variables of \(P_i\) will have their initial values and the value of \(h_i\) will have the form
\[ h_i^1 + <i,0,\delta> + h_i^2 + <i,0,\delta> + \cdots + h_i^{n+1} + <i,0,\delta> \]

where \( h_i^j \) is the record of communications that the process goes through during its \( j^{th} \) partial execution. Let \( h''_i \) be the sequence of communications by the process during its final execution, and let the final state of the local variables of the process be denoted by \( S_i^f \). Since after each error, the local state of \( [P_i] \) is reset to its initial value, a possible execution of \( [P_i] \) would be one in which \( n \) (rather than \( n+1 \)) errors were encountered, with the \( n+1^{st} \) execution proceeding without error and reaching the same final state \( S_i^f \), and with its communication sequence being

\[ h_i^2 + <i,0,\delta> + \cdots + h_i^{n+1} + <i,0,\delta> + h''_i. \]

Thus the \( n \)-error execution of \( [P_i] \) is identical to the \( n+1 \)-error execution except that it avoids the first error of that execution. Also, \( h_i^1 \) will satisfy \( r \) (i.e., \( r_{h_i^1} \) will be true if \( h_i' = h_i^1 \)). Then if the results obtained following the \( n \)-error execution of \( [P_i] \) satisfy \( q' \), and the second implication in R13 is true, the results obtained following a \( n+1 \)-error execution will also satisfy \( q' \).

It would appear that it should be possible to obtain the predicate \( r \) directly from the partial correctness post-condition \( q \) of \( P_i \), rather than by building it up during the proof of \( P_i \). One way of doing this would be to define

\[ r \equiv \exists h_i'. [h_i \subseteq h_i' \land q_{h_i'}^h] \]

and to argue that any sequence \( h_i \) that satisfies \( r \) must be the initial subsequence of some sequence \( h_i' \) that will satisfy \( q \). Recall, however, the example given earlier where we indicated why a simple Hoare-style rule was inadequate for proving properties of fault tolerant programs. If \( P_5 \) starts execution with the pre-
condition $h_5 = \epsilon$, and no errors occur, we can obtain the post-condition $[h_5 = <5,6,2>]$. $r$ could then be

$$r \equiv [h_5 = \epsilon \lor h_5 = <5,6,2>]$$

Using this to derive the post-condition when errors do occur would give

$$[\forall k, 1 \leq k \leq |h_5|, [h_5[k] = <5,6,2> \lor h_5[k] = <5,0,\delta>]]$$

This is incorrect, as some execution of $P_5$ may choose the first guard, send 1 to $P_6$, fail, recover (i.e. send $\delta$ to $P_6$), choose the next guard, send 2 to $P_6$ and terminate. For such an execution of $P_5$, we will have

$$h_5 = <5,6,1><5,0,\delta><5,6,2>$$

which does not satisfy the post-condition. We must therefore adopt the alternative task of proving

$$(r) \{h_5 = \epsilon\} P_5 \{h_5 = <5,6,2>\}$$

with $r \equiv [h_5 = \epsilon \lor h_5 = <5,6,1> \lor h_5 = <5,6,2>]$

7. Parallel Composition of Fault-Tolerant Processes

R14. Parallel Composition

$$\{p_i \land h_i = \epsilon\}[P_i\{q_i\}, i = 1, \ldots, n]$$

$$\{p_1 \land \ldots \land p_n\}[P_1 || \ldots || P_n\{q_1 \land \ldots \land q_n \land Compat(h_1, \ldots, h_n)\}]$$

R14 seems identical to the rule for parallel composition in [8]; however, the
definition of \textit{Compat} is slightly different, to take care of the \( \delta \) and \( \rho \) type elements that may appear in communication sequences.

\[
\text{Compat}(h_1,\ldots,h_n) \equiv \exists h . \left( \forall i, 1 \leq i \leq n, h |, i = h_i \land R1(h) \land R2(h) \right)
\]

where

\[h |, j\]

is the subsequence of all elements of \( h \) which are of the form \( <i,j,m> \), \( <j,i,m> \), \( <j,0,\delta> \) or \( <i,j,\rho> \) where \( m \notin \{\rho, \delta\} \)

Informally,

\( R1(h) \equiv \forall P_1, \ldots, P_n \) are informed of all faults

\( \equiv \) if the \( k^{th} \) element of \( h \) is an explicit communication between \( i \) and \( j \)

and if there exists \( k' < k \) such that \( h[k'] = <i,0,\delta> \)

then there exists \( k'', k' < k'' < k \) such that \( h[k''] = <i,j,\rho> \)

Formally,

\[
R1(h) \equiv \forall k . 1 \leq k \leq |h| .
\[
\left( (h[k] = <i,j,m> \lor h[k] = <j,i,m>) \land m \notin \{\rho, \delta\} \land \exists k' . k' < k . h[k'] = <i,0,\delta> \right) \Rightarrow \exists k'' . k' < k'' < k . h[k''] = <i,j,\rho>
\]

Informally,

\( R2(h) \equiv P_0 \) detects all faults

\( \equiv \) if the \( k^{th} \) element of \( h \) is \( <i,j,\rho> \)

then there exists \( k', k' < k \) such that \( h[k'] = <i,0,\delta> \)

and in \( h[k' + 1 : k] \) there is no explicit communication between \( i \) and \( j \)

Formally,

\[
R2(h) \equiv \forall k . 1 \leq k \leq |h| .
\[
h[k] = <i,j,\rho> \Rightarrow \exists k' . k' < k .
\[
\{h[k'] = <i,0,\delta> \land (h[k' + 1 : k] | j) | i = \epsilon}\]
The definition ensures that if $P_i$ fails (and this is recorded by a $<i,0,\delta>$ in $h_i$), then before $P_i$ and $P_j$ can communicate with each other, $P_j$ must 'register' that $P_i$ had failed and recovered (as recorded by a $<i,j,\rho>$ in $h_j$). Similarly, it ensures that if $P_j$ has registered failure and recovery of $P_i$ (i.e. if there is a $<i,j,\rho>$ in $h_j$), $P_i$ must indeed have failed and recovered (recorded by a $<i,0,\delta>$ in $h_i$) since the last communication between $P_i$ and $P_j$. Note that even when a process $P_i$ fails several times before communicating with some other process $P_j$, exactly one triple $<i,j,\rho>$ will be appended to $h_j$ when $P_j$ next attempts to communicate with $P_i$. As an example, if $P_1$ fails and then sends a value 3 to $P_2$ the various sequences will be as follows:

$$h = <1,0,\delta><1,2,\rho><1,2,3>$$
$$h_1 = <1,0,\delta><1,2,3>$$
$$h_2 = <1,2,\rho><1,2,3>$$

8. Non-terminating Processes and Programs

The axioms and rules of the last section can be used to obtain the post-condition of fault tolerant programs; frequently, however, we wish to consider the behaviour of non-terminating programs, such as the bounded buffer program given earlier. In this section we explain how we can use the approach of this paper to deal with such programs.

Consider a fault tolerant program

$$[P_1 || ... || P_n]$$

Suppose for each of the processes $P_i$ ($i=1,...,n$), we have shown, using the axioms and rules of the last section, the following results:
\[(r_i) \{ p_i \land h_i = \epsilon \} P_i \{ q_i \}\]

If \( P_i \) is a non-terminating process, \( q_i \) will presumably be the predicate \( \text{false} \); however, we are interested not in the post-condition of \( P_i \) (or of the other processes), but in the predicates it satisfies at certain points during its execution. Frequently, for instance, we are interested in the ‘invariant’ relation that is satisfied by the variables of \( P_i \) (in particular by its communication sequence \( h_i \)) at all times during \( P_i \)’s execution. If each of the processes is non-terminating, as is often the case (and is the case in the bounded buffer), we are usually interested in the invariant relation that the communication sequences of the various processes satisfy; in particular, in the case of the bounded buffer we would probably like to prove that the lines produced by \( P_1 \) are received by \( P_4 \) in proper order, possibly with some repetitions, and that the number of repetitions is bounded by some function of the number of failures of the processes \( P_2 \) and \( P_3 \).

If we are able to prove
\[(r_i) \{ p_i \land h_i = \epsilon \} P_i \{ q_i \}\]
then throughout a \( \text{fault-free} \) execution of \( P_i \), its communication sequence \( h_i \) will satisfy the predicate \( r_i \) (recall that \( r_i \) can only involve \( h_i \) -- no other variables of \( P_i \) or of any other process may appear in \( r_i \)). However an arbitrary execution of \( P_i \) will not be fault-free, and we have to find a relation that will be satisfied by \( h_i \) even if the particular execution of \( P_i \) that we are considering has a number of faults (each, of course, followed by a recovery). In other words, we wish to find a relation \( r_i' \) that will be satisfied during executions of \( P_i \) that may include faults. Clearly then \( r_i' \) may be obtained in more or less the same way that \( q' \) of rule \( R_{13} \) was obtained in the last section.

Thus, following the analogy of \( R_{13} \), \( r_i' \) must be such that
\[r_i \Rightarrow r_i'\]
\[\left[ r_i' \land r_i h_i \right] \Rightarrow r_i' h_i' + <i,0,\delta> + h_i\]
More formally, we can consider the following rule of inference:

R15. Process Execution With Faults

\[
( r_i \{ p_i \land h_i = \epsilon \} P_i \{ q_i \} ) \Rightarrow r_i'
\]

\[
\begin{align*}
[ r_i' \land r_i^{h_i} ] & \Rightarrow r_i'^{h_i} + <i,0,\delta> + h_i \\
\{ p_i \} [ P_i ] ( r_i' ) & 
\end{align*}
\]

where the notation \( \{ p_i \} [ P_i ] ( r_i' ) \) means the following: if the initial values of the variables of \( P_i \) (not including \( h_i \)) will satisfy \( p_i \), then throughout the execution of \([ P_i ]\) (i.e., including those executions in which \( P_i \) goes through a number of faults and recoveries), the communication sequence \( h_i \) will satisfy the relation \( r_i' \).

Next, suppose we have been able to prove

\[
\{ p_i \} [ P_i ] ( r_i' ) , \; i=1,\ldots,n 
\]

It should then be clear that throughout the execution of the program \([ P_1 \ || \ \ldots \ || \ P_n ]\), the various communication sequences \( h_1,\ldots,h_n \) will satisfy the following relation:

\[
r_1' \land r_2' \land \ldots \land r_n' \land Compat( h_1,\ldots,h_n ) 
\]

The predicate \( Compat \) will be satisfied at all times during the execution of the program, since it merely expresses the requirement that the communications of the various processes, as recorded in their individual communication sequences, must be mutually compatible, and that this requirement must be met at all times during the execution of the program -- not merely when the program finishes execution.
Thus, an appropriate rule of inference would be

\[
\{p_i\} [P_i] (r'_i), i = 1, \ldots, n
\]

\[
\{p_1 \land \cdots \land p_n\} \parallel [P_1] \parallel \cdots \parallel [P_n]\parallel (r'_1 \land \cdots \land r'_n \land \text{Compat}(h_1, \ldots, h_n))
\]

We shall use the notations and rules of this and the last section to show, in the next section, that the bounded buffer program does indeed behave the way we might expect. The reader may recall that in the bounded buffer program we assumed that \(P_1\) and \(P_4\) do not fail; clearly, then, if we can prove

\[
(r_i) \{p_i \land h_i = \epsilon\} P_i \{q_i\}, \quad i = 1, \ldots, 4
\]

the appropriate invariant for the whole program, allowing for faults in \(P_2\) and \(P_3\) but not in \(P_1\) and \(P_4\), would be

\[
r_1 \land r_2' \land r_3' \land r_4 \land \text{Compat}(h_1, \ldots, h_4)
\]

rather than

\[
r_1' \land r_2' \land r_3' \land r_4' \land \text{Compat}(h_1, \ldots, h_4)
\]

In the next section we shall prove

\[
(r_i) \{p_i \land h_i = \epsilon\} P_i \{q_i\}, \quad i = 1, \ldots, 4
\]

with appropriate \(r_i\) (\(p_i\) being identically \textit{true}, and \(q_i\) identically \textit{false}, and each of the processes being in an infinite loop), and show that

\[
[r_1 \land r_2' \land r_3' \land r_4 \land \text{Compat}(h_1, \ldots, h_4)] \Rightarrow [T_1 \land T_2 \land T_3 \land T_4]
\]

where

\[
\text{Compat}(h_1, \ldots, h_4)
\]
\[ T_1 \equiv \forall k . \ 1 \leq k \leq |h_4| . \ \{ h_4[k] \in \{ \mathbf{<3,4,\rho>, \mathbf{<3,4,Line(m)>} \} \} \] (4)

that is \( h_4 \in \{ \mathbf{<3,4,\rho>, \mathbf{<3,4,Line(m)>} \} \)∗

Thus \( T_1 \) in (3) will essentially show that the only “normal” values the printer process \( P_4 \) will receive are the “lines.” (Recall that \( P_4 \) prints all the normal values it receives.)

\[ T_2 \equiv [\forall k . \ 1 \leq k \leq |h_4| . \ \{ h_4[k] = \mathbf{<3,4,\rho>} \} \]
\[ \lor [\exists k . \ 1 \leq k \leq |h_4| . \ \{ h_4[k] = \mathbf{<3,4,Line(1)>} \}
\]
\[ \land \forall k' . \ 1 \leq k' < k . \ \{ h_4[k'] = \mathbf{<3,4,\rho>} \} \} ] \] (5)

that is \( h_4 \in \{ \mathbf{<3,4,\rho>} \} \)∗

or \( h_4 \in \{ \mathbf{<3,4,\rho>} \} \)∗ \( \mathbf{<3,4,Line(1)>} \) \{ \mathbf{<3,4,\rho>, \mathbf{<3,4,Line(m)>} \} \)∗

\( T_2 \) ensures that the first normal value that \( P_4 \) receives is \( \text{Line}(1) \).

\[ T_3 \equiv \forall k,k'. \ 1 \leq k \leq k' \leq |h_4| . \]
\[ \{ h_4[k] = \mathbf{<3,4,Line(m)>} \land h_4[k'] = \mathbf{<3,4,Line(m')>} \]
\[ \Rightarrow \forall m'' . \ m < m'' < m' . \]
\[ \{ \exists k'' . k' < k'' < k. \]
\[ \{ h_4[k''] = \mathbf{<3,4,Line(m'')>} \} \} \} \] (6)

\( T_3 \) ensures that if at some time \( t \) the \( m^{th} \) line is printed and at a later time \( t' \) the \( m'^{th} \) line is printed then all lines from \( m + 1 \) to \( m' - 1 \) are printed between time \( t \) and \( t' \) (possibly with duplications).

\[ T_4 \equiv f_2(h_4) \leq (n-1) * f_1(h_2) \] (7)

where
\[ f_1(h_2) = \text{number of elements of the kind } <2,0,\delta> \text{ in } h_2, \text{ i.e., the number of failures of } P_2 \]

\[ f_2(h_4) = \text{number of repetitions in elements of the kind } <3,4,\text{Line}(m)> \text{ in } h_4.\]

\(f_1(h_2)\) and \(f_2(h_4)\) can be defined in a straightforward manner and we leave that for the reader. Thus \(T_4\) specifies an upper bound on the number of duplications in the lines printed; it is possible to get a tighter bound involving \(h_2, h_3\) and \(h_4\); however, this would be much more complex than \(T_4\), since it would involve the relative times at which \(P_2\) and \(P_3\) failed (not just the number of failures of \(P_2\) and \(P_3\)).

With \(T_1, T_2, T_3\) and \(T_4\) as defined in (4), (5), (6) and (7), (3) will clearly show that the program does indeed behave as we expect it to.

9. Proof of the Bounded Buffer Program

Our proof of the bounded buffer program will be quite informal; we shall begin by informally proving the results

\[(r_i) \{h_i = \in\} P_i \{false\}, \quad i=1,\ldots,4\]  

(8)

The formalization of these proofs (appealing to the various axioms and rules applicable to the statements in \(P_i\)) will be left to the interested reader. Our informal arguments will be rather like the semi-formal proofs of sequential programs, omitting most of the intermediate details and all but the key assertions such as loop invariants; such informal proofs are justified since our formalism allows (or rather requires) us to consider one process at a time, and the validity of the various assertions in the proof of a process depends entirely on what the
process does - and not at all on what the other processes do or on how they interact with this process or with each other. Thus once the intuition behind the proof rules is understood, it is as easy to informally prove a result such as (8) as it is to informally prove the partial correctness of sequential programs.

However, in the current case, we have to prove an additional result, after proving (8):

\[ r_1 \land r_2' \land r_3' \land r_4 \land \text{Compat}(h_1, \ldots, h_4) \Rightarrow [T_1 \land T_2 \land T_3 \land T_4] \]

Let us begin by considering the process \( P_4 \) of the bounded buffer since it is the simplest. (The processes \( P_1 \) through \( P_4 \) have been introduced in section 3 but we will repeat them here for convenience.)

\[ P_4 ::= \text{ln : line; } \]
\[ *P_3 ? \text{ln} \rightarrow \text{skip} \]
\[ <<<\text{skip}>>> \]

The reader should easily be able to see the following result:

\[(r_4) \{ h_4 = \epsilon \} P_4 \{ \text{false} \} \] (9)

where,

\[ r_4 \equiv \forall k . 1 \leq k \leq |h_4| . \{ h_4[k] \in \langle 3,4,m \rangle, \langle 3,4,\rho \rangle \} \land m \notin \{ \rho, \delta \} \}

(10)

that is \( h_4 \in \{ \langle 3,4,m \rangle, \langle 3,4,\rho \rangle \}^* \)

(the loop invariant will be identical to \( r_4 \)).

The post-condition in (9) merely expresses the fact that the loop does not terminate; \( r_4 \) will be true at all times during the execution of \( P_4 \) since the only communications \( P_4 \) participates in are those in which it receives an input from \( P_3 \) or
a signal that \( P_3 \) has failed and recovered since the previous communication between \( P_3 \) and \( P_4 \).

Next consider \( P_1 \):

\[
P_1:: \text{pseq,sent : integer; ready : boolean; nextline : line;}
\]
\[
pseq:=0; \text{sent}:=0; \text{ready}:=\text{false};
\]
\[
\text{*[\neg ready \rightarrow nextline:=Line(pseq+1); ready:=true
\]
\]
\[
\square
\]
\[
\text{ready; } P_2 \vdash (\text{nextline,pseq+1} \rightarrow \text{pseq}:=\text{pseq+1}; \text{ready}:=\text{false};
\]
\[
\text{sent}:=\text{sent}+1
\]
\[
\text{<<ready}:=\text{false};
\]
\[
[\text{sent} < n \rightarrow \text{pseq}:=\text{pseq} - \text{sent}
\]
\[
\square
\]
\[
\text{sent} \geq n \rightarrow \text{pseq}:=\text{pseq} - n
\]
\[
]; \text{sent}:=0 >>
\]

The loop invariant for \( P_1 \) is:

\[
\text{LI}_1 \equiv h_1 \in \{<1,2,\rho>,<1,2,(\text{Line}(m),m)>\}^*
\]
\[
\Lambda \text{ready} \Rightarrow \text{nextline} = \text{Line}(\text{pseq}+1)
\]
\[
\Lambda \text{pseq} = g_1(h_1) \land \text{sent} = g_2(h_1)
\]
\[
\forall k . 1 \leq k \leq |h_1| . [h_1[k] = <1,2,(p,q)> \Rightarrow p = \text{Line}(q) \land q = g_1(h_1[1:k-1]+1)]
\]

where

\[
g_1(\epsilon) = 0
\]
\[
g_1(h + <1,2,(p,q)>) = g_1(h) + 1
\]
\[
g_1(h + <1,2,\rho>) = g_1(h) - \min(n,g_2(h))
\]
\[
g_2(\epsilon) = 0
\]
\[ g_2(h + \langle 1,2, (p,q) \rangle) = g_2(h) + 1 \]
\[ g_2(h + \langle 1,2, \rho \rangle) = 0 \]

The relation \( r_1 \) is

\[ r_1 \equiv \forall k \cdot 1 \leq k \leq |h_1| \cdot [h_1[k] = \langle 1,2, (p,q) \rangle \]
\[ \Rightarrow p = \text{Line}(q) \land q = g_1(h_1[1:k-1]) + 1 \]

As stated earlier, we will leave it to the reader to formally verify \((r_1) \{ h_1 = \in \}\) \(P_1 \{ \text{false} \}\), using \(LI_1\) as the loop invariant, and introducing appropriate assertions as necessary.

Next consider \(P_2\).

\[
P_2 :: \text{in, out} : \text{integer}; \text{output} : \text{boolean};
\[
A : [0..(n-1)] \text{ of record } \text{ln} : \text{line}; \text{linenum} : \text{integer end};
\[
in:=0; \text{out}:=0; \text{output}:=\text{false};
\[
*[\text{in} < \text{out}+n-2; P_1?A[\text{in mod n}] \rightarrow \text{in}:=\text{in}+1
\]
\[
\square
\]
\[
\text{out} < \text{in}; P_3!A[\text{out mod n}] \rightarrow \text{out}:=\text{out}+1; \text{output}:=\text{true}
\]
\[
<\text{output} \rightarrow \text{out}:=\text{out} - 1
\]
\[
\square
\]
\[
\neg \text{output} \rightarrow \text{skip}
\]
\[
]; \text{output}:=\text{false} >>
\]

We will only specify \(r_2\), leaving the other assertions including the loop invariant to the reader.
\( r_2 = \) what \( P_2 \) receives from \( P_1 \) is sent out to \( P_3 \)

and the number of values received from \( P_1 \) but not sent to \( P_3 \) can

be atmost \( n - 1 \)

More formally,

\[
\begin{align*}
    r_2 & = \left[ f_3(h_2 | 3) \right]_{1,2}^{2,3} \subseteq h_2 | 1 \\
    & \land \left[ | f_3(h_2 | 3) | \leq | h_2 | 1 \right) \leq | f_3(h_2 | 3) | + n - 1 \right]
\end{align*}
\]

where

\( f_3(h_2 | 3) = \) sequence obtained from \( h_2 | 3 \) by dropping from it all

subsequences of the form \( <2,3,m> < 3,2,\rho> \) for all \( m \).

\( = h_2 | 3 \) if in that execution there were no failures in \( P_3 \)

\( = h_2 | 3 \) in the execution of \( P_2 \) where the fault alternative

\( <<..>> \) has been removed

More formally \( f_3(h_2 | 3) \) is defined as follows:

\[
\begin{align*}
    f_3(\emptyset) &= \emptyset \\
    f_3(\langle<3,2,\rho> + h\rangle) &= f_3(h) \\
    f_3(\langle<2,3,m> + <2,3,m'> + h\rangle) &= <2,3,m> + f_3(\langle<2,3,m'> + h\rangle) \text{ for } m, m' \neq \rho \\
    f_3(\langle<2,3,m> + <3,2,\rho> + h\rangle) &= f_3(h)
\end{align*}
\]

In writing down \( r_2 \), we have allowed for failures in \( P_3 \), but not in \( P_1 \) since \( P_1 \) is

assumed not to fail; failures in \( P_2 \) will be accounted for when we write down \( r_2' \).

We still need to consider \( P_3 \):
\( P_3 :: \text{ln} : \text{line}; \text{num}, \text{lastnum} : \text{integer}; \)

\[
\text{lastnum} := 0; \\
*[P_2!\text{ln}, \text{num} \rightarrow [\text{num} > \text{lastnum} \rightarrow P_4!\text{ln}; \text{lastnum} := \text{num} \langle< \text{skip} >>] \\
\]

Again, we will only specify \( r_3 \), leaving the formal verification of \( (r_3) \{ h_3 = \emptyset \} \)

\( P_3 \{ f_{\text{false}} \} \) to the reader. First define

\[
f_4(\{<*,*,(m_1,m_2)>\}^*, i) = \text{return only those triples } <*,*,m_1> \text{ whose second component of value are in increasing order starting from greater than } i
\]

Note that the difference between \( P_3' \) and the fault-tolerant \( P_3 \) given above is that some 'evasive action' is taken in the fault-tolerant version to compensate for possible failures in \( P_2 \). Then, \( f_4(h_3|2,0) \) of fault-tolerant \( P_3 = h_3|2 \) of \( P_3' \).

\[
r_3 \equiv \left[ h_3 \mathrel{|} 4 \right]_{2,3}^{3,4} \subseteq f_4(h_3 \mathrel{|} 2,0) \\
\Lambda [ \left| h_3 \mathrel{|} 4 \right| \leq \left| f_4(h_3 \mathrel{|} 2,0) \right| \leq \left| h_3 \mathrel{|} 4 \right| + 1 ]
\]

More formally

\[
f_4(\emptyset, i) = \emptyset \]
\[
f_4(\langle<2,3,\rho> + h, i \rangle) = f_4(h,i) \]
\[
f_4(\langle<2,3,(m_1,m_2)> + h, i \rangle) = \text{if } m_2 \leq i \text{ then } f_4(h, i) \]
\[
\text{else } <2,3,m_1> + f_4(h,m_2) \]

\( r_3 \) expresses the fact that \( [h_3 \mathrel{|} 4]_{2,3}^{3,4} \) is a prefix of \( f_4(h_3 \mathrel{|} 2,0) \) and that the length
of \( h_3 \mid 4 \) can be at most 1 less than that of \( f_4(h_3 \mid 2,0) \).

That completes the informal proofs of

\[
(r_i) \{ h_i = \varepsilon \} P_i \{\text{false} \}, \ i=1,...,4
\]

Before trying to prove

\[
[r_1 \land r_2' \land r_3' \land r_4 \land \text{Compat}(h_1,..,h_4)] \Rightarrow [T_1 \land T_2 \land T_3 \land T_4]
\]  

(11)

we write down \( r_2' \) and \( r_3' \) (from \( r_2 \) and \( r_3 \) respectively):

\[
r_2' \equiv \forall k \cdot 1 \leq k \leq \text{Num}(h_2, \delta) + 1 \cdot \{ r_2^h_{\text{subseq}}(h_2, k-1, k, \delta) \}
\]

where

\( \text{Num}(h_i, x) = \) number of triples of the kind \( <j,i,x> \) in \( h_i \)

\( \text{subseq}(h_i, m1,m2,x) = \) The subsequence of \( h_i \) from just after the \( m1^{th} \) element of \( h_i \) of the form \( <j,i,x> \) (from the beginning of \( h_i \) if \( m1 = 0 \)) to just before the \( m2^{th} \) element of \( h_i \) of the form \( <j,i,x> \) (to the end of \( h_i \) if \( m > \text{Num}(h_i, x) \)).

Essentially \( r_2' \) says that \( h_2 \) looks like a concatenation of a number of (smaller) sequences each of which satisfies \( r_2 \), with a \( <2,0,\delta> \) element sandwiched between each consecutive pair of these smaller sequences.

\( r_3' \) is similar:

\[
r_3' \equiv \forall k \cdot 1 \leq k \leq \text{Num}(h_3, \delta) + 1 \cdot \{ r_3^h_{\text{subseq}}(h_3, k-1, k, \delta) \}
\]

**Proof of** \( T_1 \)

From \( r_1 \) and \( r_2' \) we can infer that

\[
h_2 \mid 1 \in \{ <1,2, (\text{Line}(m),m) > \}^*
\]
This together with $r_3'$ gives us

$$h_3 \mid 2 \in \{ <2,3, (\text{Line}(m),m)>, <2,3, \rho> \}^*$$

Combining this with $r_4$ we get

$$h_4 \in \{ <3,4, \text{Line}(m)>, <3,4, \rho> \}^*$$

**Proof of $T_2$**

From $r_1$ and $r_2'$ we know that

$$h_2 \mid 3 \in \{ <2,3, (\text{Line}(1),1)>, <3,2, \rho>, <3,2, \rho> \}^*$$

$$+ <2,3, (\text{Line}(1),1)>, <2,3, (\text{Line}(2),2)>, \text{+ some arbitrary trace.}$$

Combining this with $r_3'$ gives us

$$h_3 \mid 4 \in <3,4, \text{Line}(1)> + \text{+ some arbitrary trace}$$

and hence

$$h_4 \in \{ <3,4, \rho> \}^* <3,4, \text{Line}(1)> + \text{+ some arbitrary trace.}$$

**Proof of $T_3$**

From $r_1$ and $r_2'$ we obtain

$$\forall k \cdot 1 \leq k \leq |h_2|. \{ h_2[k] = <1,2,(\text{Line}(m),m)> \wedge m > 1$$

$$\Rightarrow \ \exists k'. \ k' < k . \{ h_2[k'] = <1,2,(\text{Line}(m-1),m-1)>$$

$$\wedge h_2[k'+1 : k-1] \mid 1 \in \{ <1,2,(\text{Line}(p),p)> \}^*$$

$$\text{where } p \geq m \}$$

Combining this with $r_2'$ gives us

$$\forall k \cdot 1 \leq k \leq |h_3|. \{ h_3[k] = <2,3,(\text{Line}(m),m)> \wedge m > 1$$

$$\Rightarrow \ \exists k'. \ k' < k . \{ h_3[k'] = <2,3,(\text{Line}(m-1),m-1)>$$

$$\wedge h_3[k'+1 : k-1] \mid 2 \in \{ <2,3,(\text{Line}(p),p)>, <2,3, \rho> \}^*$$

$$\text{where } p \geq m \}$$

From the above and $r_4$ we can infer that
\begin{align*}
\forall k \quad 1 \leq k \leq |h_4|. \quad \{h_4[k] = <3,4,\text{Line}(m)> \wedge m > 1 \\
\Rightarrow \exists k'. \quad k' < k. \quad \{h_4[k'] = <3,4,\text{Line}(m-1)> \\
\wedge h_4[k' + 1 : k-1] \in \{<3,4,\text{Line}(p)>, <3,4,\rho>\}^* \\
\text{where } p \geq m\}
\end{align*}

which can be rewritten as

\begin{align*}
\forall k, k'. \quad 1 \leq k \leq k' \leq |h_4|. \\
\{h_4[k] = <3,4,\text{Line}(m)> \wedge h_4[k'] = <3,4,\text{Line}(m')> \\
\Rightarrow \forall m'' . \quad m < m'' < m'. \\
\exists k'' . \quad k' < k'' < k. \\
\{h_4[k''] = <3,4,\text{Line}(m'')> \} \}
\end{align*}

**Proof of T_4.**

From \(r_1\) we obtain

number of repetitions in \(h_1|2 \leq (n-1) \times \text{number of failures of } P_2\)

Combining this with \(r_2'\) we get

number of repetitions in \(h_2|3 \leq (n-1) \times \text{number of failures of } P_2 + \text{number of failures of } P_3\)

From the above and \(r_3'\) we get

number of repetitions in \(h_3|4 \leq (n-1) \times \text{number of failures of } P_2\)

Since

number of repetitions in \(h_3|4 = \text{number of repetitions in } h_4\)

we have proved \(T_4.\)
10. Discussion

The proof rules defined here provide a means for formally proving the properties of fault tolerant programs written in extended CSP. They are naturally more complex than proof rules for programs that are not subject to faults but, with some familiarity, their interpretation becomes no more difficult than the task of constructing fault tolerant programs. In fact, during the course of proving the properties of even simple programs we have sometimes found errors in programs that earlier passed fairly thorough but informal inspection.

The rules are limited to proofs of partial correctness and it is tempting to consider how they may be extended to deal with total correctness. For example, if it can be proved that all loops will terminate, the function $Compat$ can be extended to detect deadlock and to prove process termination. Unfortunately, proof of loop termination cannot in general be done in isolation and requires re-inspection of the proof outlines of all processes. This militates against the basic intention of this proof system that once the proofs of individual processes are over, proof of the program should directly result from the application of the rule of parallel composition. The possibility of carrying forward loop termination predicates in post-conditions and proving their ‘compatibility’ at the end must be rejected because it results in extremely unwieldy proofs. So the proof system remains one of partial correctness.

It is important to prove that the proof system is consistent and complete with respect to an operational model for extended CSP. Though we shall not attempt to demonstrate this here, it appears quite possible for the proof given in [9] for the proof system of [8] to be adapted for our version of extended CSP.
Throughout this paper we have assumed that a failed process is restarted from it initial state. Such an assumption allows the possibility of running the system of processes using read only stable storage. On the other hand, the use of checkpoints requires that the processor states be stored at various checkpoints and a failed process along with all the other necessary processes must be restarted from some previous checkpoint. Such a technique requires the usage of stable storage but may at times simplify the recovery action involved. Checkpoints can be handled by suitable altering axiom R13 for Failure-Prone Process Execution.

REFERENCES


