AN APPROACH TO DESIGNING FAULT-TOLERANT COMPUTING SYSTEMS*

Richard D. Schlichting
Fred B. Schneider

TR 81-479
November 1981

Department of Computer Science
Cornell University
Ithaca, New York 14853

*This work is supported, in part, by NSF Grant MCS-81-03605

Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems[*]

Richard D. Schlichting[+]
Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

November 19, 1981
Revisions: July 30, 1982

ABSTRACT

A methodology that facilitates the design of fault-tolerant computing systems is presented. It is based on the notion of a fail-stop processor. Such a processor automatically halts in response to any internal failure and does so before the effects of that failure become visible. The problem of implementing processors that, with high probability, behave like fail-stop processors is addressed. Axiomatic program verification techniques are described for use in developing provably correct programs for fail-stop processors. The design of a process control system illustrates the use of our methodology.

[*]This work is supported in part by NSF Grant MCS-81-03605.
[+]Department of Computer Science, University of Arizona, Tucson, Arizona 85721.

1. Introduction

Programming a computer system that is subject to failures is a difficult task. A malfunctioning processor might perform arbitrary and spontaneous state transformations instead of the transformations specified by the programs it executes. Thus, even a correct program cannot be counted on to implement a desired input-output relation when executed on a malfunctioning processor. On the other hand, it is impossible to build a computer system that always operates correctly in spite of failures in its components by using (only) a finite amount of hardware[1]. Thus, the goal of implementing completely fault-tolerant computing systems is unattainable. Fortunately, most applications do not require complete fault-tolerance. Rather, it is sufficient that the system work correctly provided no more than some predefined number of failures occur within some time interval, or provided certain types of failures do not occur. This more modest goal *is* attainable.

In this paper we present an approach to designing fault-tolerant computing systems based on the notion of a *fail-stop processor*, a processor with well-defined failure-mode operating characteristics. Briefly, our approach is as follows. First, software is designed that runs on a computing system composed of one or more fail-stop processors; the number of processors required is dictated by response-time constraints that must be satisfied by the system. Then, a computing system is designed that implements the requisite fail-stop processors.

We proceed as follows. Section 2 describes the characteristics of a fail-stop processor and how such processors can be approximated using present-day hardware. Section 3 discusses extensions to axiomatic verification techniques to facilitate development of provably correct programs for fail-stop processors. Satisfying

[1] *Sed quis custodiet ipsos Custodes*? (Who shall guard the guards themselves?) [Juvenal 130].

response time constraints in the presence of failures is the subject of section 4. Section 5 discusses the application of our approach to a non-trivial problem, the design of a fault-tolerant process-control system. Section 6 contrasts our work with other approaches to designing fault-tolerant systems and section 7 contains some conclusions.

2. Fail-Stop Processors

2.1. Definition

A processor is characterized by the instruction set it supports. Each instruction causes a well-defined transformation on the internal state of the processor and/or the connected storage and peripheral devices. Thus, the effects of executing each instruction can be described by a precise semantic definition, be it a temporal axiomatization of the instruction set [Pnueli 79] or a "Principles of Operation" manual. A *failure* occurs when the behavior of the processor is not consistent with this semantic definition.

A *fail-stop processor* is distinguished by its extremely simple failure-mode operating characteristics. First, the internal state of a fail-stop processor and some predefined portion of the connected storage are assumed to be *volatile*. The contents of volatile storage are irretrievably lost whenever a failure occurs. The remaining storage is defined to be *stable*; it is unaffected by any kind of failure. Secondly, in contrast to a real processor, a fail-stop processor never performs an erroneous state transformation due to a failure. Instead, the processor simply halts. Thus, the only visible effects of a failure in a fail-stop processor are:

FS1: It stops executing.

FS2: The internal state and contents of the volatile storage connected to it are lost.

2.2. Implementation

While the notion of a fail-stop processor is a useful abstraction, it is impossible to implement using a finite amount of hardware. With only a finite amount of hardware, a finite number of failures could disable all the error detection mechanisms and thus allow arbitrary behavior. However, it is possible to construct computing systems that, with high probability, approximate the behavior of a fail-stop processor.

One approach is to construct a system that behaves as specified, unless too many failures occur within some specified time interval, after which no assumptions about its behavior can be made. A k-fail-stop processor is a computing system that behaves like a fail-stop processor unless k+1 or more failures occur in its components.

A k-fail-stop processor can be implemented by exploiting any solution to the Byzantine Generals [Lamport et al 80] (or Interactive Consistency [Pease et al 79]) problem. Such a protocol allows a collection of processors to agree on a value sent by a potentially faulty transmitter ("Commanding General"), so that:

(1) each non-faulty processor agrees on the value sent by the transmitter, and

(2) if the transmitter in non-faulty, each non-faulty processor receives the value sent by the transmitter.

A number of real processors and volatile memory units are interconnected by a communications network to form a single k-fail-stop processor and its attendant stable storage. Each memory unit M_i is read by all processors but written to by only one, P_i. Failures are detected by having each processor run the same program and comparing results. Thus, a copy of each variable is stored in each memory unit. During execution, whenever the value of a variable from stable storage is required, the value of that variable is read from each memory unit and a solution to the Byzantine Generals Problem is employed to distribute the vector of values read to every processor. If all of the values are not identical then a failure has occurred and it is signaled. (Non-faulty processors will halt when the failure is signalled.) A total of $2k+1$ processors are required in order for up to k failures to be tolerated without compromising (1) and (2) above [Lamport et al 80].

Since processors execute asynchronously, execution of the replicated programs must be synchronized to compare results. This can be accomplished, assuming processor clocks run at roughly the same rate, by associating a logical clock [Lamport 78] with each program. This logical clock is incremented whenever a variable that is suppose to be stored in the stable storage of the k-fail-stop processor is read or written. To synchronize, a processor constructs a vector of the values of each of these clocks, again using a solution to the Byzantine Generals problem, and busy-waits until all components in the vector have the same value or a "time-out" period has elapsed. In the latter case, a failure has occurred and it is signaled.

When a collection of k-fail-stop processors are interconnected, it must be possible for one to detect that another has stopped and then to read the contents of that k-fail-stop processor's stable storage. This is accomplished as follows. Each k-fail-stop processor is connected to a communications network that allows it to read the contents of the memory units that make up the other k-fail-stop processors. A special location, $failed_i$, in each memory unit M_i is reserved to record whether processor P_i thinks the fail-stop processor it is a part of has halted due to a failure. A k-fail-stop processor fsp determines that another, fsp', has halted by computing the vector of values $failed_i$ for each memory unit in fsp' using a solution to the Byzantine Generals Problem. If any of the components has value true then fsp' is deemed halted. Should fsp require the values of the variables in the stable storage of fsp', they can be reconstructed as follows. Each processor in fsp reads a different one of the memory units that make up fsp'. Then, using a solution to the Byzantine General Problem these values are exchanged. The majority value is taken to be the value of the variable. Since at most k of the values read from fsp' are wrong, at least 2k+1 different memory units are required to implement stable storage.

While the feasibility of implementing fail-stop processors is established by this argument, the practicality is not. However, recent work in the implementation of highly reliable processors, gives reason to believe that it is indeed practical to implement fail-stop processor approximations. Both FTMP [Hopkins *et al* 78] and SIFT [Wensley *et al* 78] could be configured to behave like a collection of fail-stop processor approximations; both employ replicated processor and memory units. Redundancy can also be introduced at lower-levels in a variety of ways [Avizienis 76] [Siewiorek & Swarz 82]. The level at which redundancy is applied is an important issue and is treated in [Barlow & Proschan 75].

3. Programming a Fail-Stop Processor

3.1. Recovery Protocols

A program executing on a fail-stop processor is halted when a failure occurs. Execution may then be *restarted* on a correctly functioning fail-stop processor. (This may be the original processor if the cause of the failure has been repaired, or it may be another fail-stop processor.) When a program is restarted, the internal processor state and the contents of volatile storage are unavailable. Thus, some routine is needed that can complete the state transformation that was in progress at the time of the failure and restore storage to a well-defined state. Such a routine is called a *recovery protocol*.

Clearly, a recovery protocol (i) must execute correctly when started in any intermediate state that could be visible after a failure and (ii) can only use information that is in stable storage. In addition, because the code for a recovery protocol must be available after a failure, it must be kept in stable storage.

We associate a recovery protocol R with a sequence of statements A called the _action statement_ to form a _fault-tolerant action_ FTA as follows:

```
FTA: action
     A
     recovery
     R
     end
```

Execution of FTA consists of establishing R as the recovery protocol to be in effect when A is executed and then executing A. The recovery protocol in effect when FTA is started is reestablished at its termination. If execution of FTA is interrupted by a failure, upon restart execution continues with the recovery protocol in effect. Subsequent failures cause execution of FTA to be halted and execution of the recovery protocol in effect to begin anew when the program is restarted. Execution of FTA terminates when execution of either A or R is performed in its entirety without interruption.

The following syntactic abbreviation will be used to denote that an action statement A serves as its own recovery protocol:

```
<action-name>: action, recovery
               A
               end
```

Such a fault-tolerant action is called a _restartable action_[2].

[2]As we shall see, any fault-tolerant action can be converted to such a restartable action simply by omitting the action statement.

A program running on a fail-stop processor must at all times have a recovery protocol in effect. This will be the case if the program itself is a single fault-tolerant action. Alternatively, a program can be structured as a sequence of fault-tolerant actions, assuming that establishment of a recovery protocol can be done in such a way that at all times either the old recovery protocol or the new one is in effect. Such an assumption seems quite reasonable.

3.2. Axioms for Fault-Tolerant Actions

Following the Floyd-Hoare axiomatic approach [Hoare 69], an *assertion* is a Boolean-valued expression involving program and logical variables. The syntactic object:

$$\{P\}\ S\ \{Q\}$$

where P and Q are assertions and S is a programming language statement, is called a *triple*. The triple {P} S {Q} is a *theorem* if there exists a proof of it in a specified formal deductive system, usually called a *programming logic*. A programming logic consists of a set of axioms and rules of inference that relate assertions, programming language statements, and triples. Of particular interest are those logics that are sound with respect to execution of programming language statements on the program state -- i.e., deductive systems that are consistent with the operation of a "real" machine. Then, the notation {P} S {Q} is usually taken to mean:

> If execution of S begins in a state in which P is true, and terminates, then Q will be true in the resulting state.

Numerous programming languages have been defined using such logics; a PASCAL-like language [Hoare & Wirth 73] extended with guarded commands [Dijkstra 76] is used in this paper.

It is often more convenient to write a *proof outline* than a formal proof. A *proof outline* is a sequence of programming language statements interleaved with assertions. Each statement S in a proof outline is preceded directly by one assertion, called its precondition and denoted pre(S), and is directly followed by an assertion, called its postcondition and denoted post(S). A proof outline is an abbreviation for a proof if:

PO1: for every statement S, the triple {pre(S)} S {post(S)} is a theorem in the programming logic, and

PO2: whenever {P} and {Q} are adjacent in the proof outline, Q is provable from P.

Let FTA be a fault-tolerant action formed from action statement A and recovery protocol R. We wish to develop an inference rule that will allow derivation of

$$\{P\}\ \mathrm{FTA}\ \{Q\}$$

as a theorem, while preserving the soundness of our programming logic with respect to execution on a fail-stop processor.

First, assume

F1: $\{P'\}\ A\ \{Q'\}$ and $\{P''\}\ R\ \{Q''\}$

have been proved. Then, for execution of A to establish Q, we must have

F2: $P \Rightarrow P'$ and $Q' \Rightarrow Q$.

Similarly, for the recovery protocol R to establish Q, the following (at least) must hold:

F3: $Q'' \Rightarrow Q$.

Recall that R is invoked only following a failure. By definition, the contents of volatile storage are undefined at that time. Therefore, any program variables needed for execution of R must be in stable storage[3]. Thus, we require

[3]If P" is stronger than wp(R,Q") then variables may appear in P" that need not be

F4: All program variables named in P" must be in stable storage.

We must also ensure that whenever the recovery protocol receives control, stable storage is in a state that satisfies P". This will be facilitated by constructing a _replete proof outline_, a proof outline that contains assertions describing those states that could be visible after a failure. Then, we will require that the precondition of the recovery protocol be satisfied in those states.

A _replete proof outline_ is a proof outline in which certain assertions have been deleted so that:

RPO1: No assertion appears between adjacent fault-tolerant actions.

RPO2: Every triple {P} S {Q} in the replete proof outline satisfies either

(a) S is a sequence of fault-tolerant actions, or

(b) $\{P \vee Q\}$ is invariant over execution of S.

RPO1 and RPO2(a) follow because the program state that exists between the execution of two fault-tolerant actions FTA_1 and FTA_2 is never visible to the recovery protocol for the enclosing fault-tolerant action -- either the recovery protocol for FTA_1 or the recovery protocol for FTA_2 will receive control. RPO2(b) follows because if $P \vee Q$ remains true while S is being executed, then either P or Q will be true of the state visible to the recovery protocol should a failure occur and both {P} and {Q} already appear as assertions in the replete proof outline.

For example, if

$$\{P\} \ FTA_1 \ \{P_1\} \ FTA_2 \ \{P_2\} \ \ldots \ FTA_n \ \{P_n\}$$

is a proof outline, then

$$\{P\} \ FTA_1; \ FTA_2; \ \ldots \ FTA_n \ \{P_n\}$$

stored in stable storage. Thus, in the interest of minimizing the amount of stable storage used, the proofs should be in terms of the weakest assertions possible.

is a replete proof outline. If assignment of an integer value to a variable is performed by executing a single, indivisible, (store) instruction -- as it is on most machines -- then

$$\{x=3\}\ x{:=}\ 6\ \{x=6\}$$

is also a replete proof outline. This is because either the precondition or the postcondition of "x:= 6" is true of every state that occurs during execution of the assignment. Even if assignment is not implemented by execution of a single instruction

$$\{val=3\}\ x{:=}\ val\ \{x=3\ \wedge\ val=3\}$$

is a replete proof outline, because the assertion {val = 3} is not destroyed by assignment to x; it is true before, during and after execution of "x:= val".

Therefore, in addition to F1 - F4, correct operation of a recovery protocol requires:

F5: Given a fault-tolerant action with action statement A and recovery protocol R satisfying F1, let $a_1, a_2, \ldots, a_n$ be the assertions that appear in a replete proof outline of {P'} A {Q'}, and $r_1, r_2, \ldots, r_m$ be the assertions that appear in a replete proof outline of {P"} R {Q"}. Then:

(i) $(\forall i\colon 1 \le i \le n\colon a_i \Rightarrow P'')$

(ii) $(\forall i\colon 1 \le i \le m\colon r_i \Rightarrow P'')$

Lastly, it must be guaranteed that failures at processors other than the one executing FTA do not interfere with (i.e., invalidate) assertions in the proof outline of FTA. Suppose an assertion in FTA names variables stored in the volatile storage of another processor.[4] Then should that processor fail, such an assertion

[4]This is often necessary when the actions of concurrently executing processes are synchronized. For example, if it is necessary to assert that a collection of processes are all executing in the same "phase" at the same time, then each would include assertions about the state of the others. See [Schlichting & Schneider 82] for an example of such reasoning.

would no longer be true since the contents of volatile storage would have been lost. Hence, we require that:

F6: Variables stored in volatile storage may not be named in assertions appearing in programs executing on other processors.

Given a fault-tolerant action, a restartable action that implements the same state transformation can always be constructed from the recovery protocol alone. (The proof of this follows from F3 and F5.) Thus, in theory, the action statement is unnecessary. In practice, the additional flexibility that results from having an action statement different from the recovery protocol is quite helpful. Presumably, failures are infrequent enough so that a recovery protocol can do a considerable amount of extra work in order to minimize the amount of (expensive) stable storage used. Use of such algorithms for normal processing would be unacceptable.

3.3. Fault-Tolerant Programs -- A Simple Example

In addition to allowing axiomatic verification of programs written in terms of fault-tolerant actions, F1 - F6 permit a programmer to develop a fault-tolerant program and its proof hand-in-hand, with the proof leading the way, as advocated in [Dijkstra 76] [Gries 81]. F4 allows identification of variables that must be stored in stable storage to be done in a mechanical way from the proof; construction of a replete proof outline provides a mechanical way to determine the intermediate states that could be visible following a failure. To illustrate the use of rules F1 - F6 as an aid in developing a recovery protocol, we consider the following (artificial) problem. (A more substantial example is treated in section 5.)

Periodically, variables x and y are updated based on their previous values. Thus, given a function G, desired is a routine called update that runs on a fail-stop processor and satisfies the following specification:

$$\{P: x = X \wedge y = Y\} \quad \text{update} \quad \{Q: x = G(X) \wedge y = G(Y)\}.$$

Logical variables X and Y represent the initial values of x and y, respectively.

If the possibility of failure is ignored, the following program will suffice:

```
S1:  {P:  $x = X \wedge y = Y$}
     S1a:  x:= G(x);     {P1a: $x = G(X) \wedge y = Y$}
     S1b:  y:= G(y);     {P1b: $x = G(X) \wedge y = G(Y)$}
     {Q:  $x = G(X) \wedge y = G(Y)$}
```

Note that this is a replete proof outline, provided assignment is implemented as an atomic operation: {P $\vee$ P1a} is invariant over execution of S1a and {P1a $\vee$ P1b} is invariant over execution of S1b.

Things become more complicated when the possibility of failure is considered. In particular, S1 could not be the action statement of a restartable action because F5 is violated (assuming G is not the identity function): both P1a $\Rightarrow$ P and P1b $\Rightarrow$ P are false. In order to construct a restartable action, we must find a way to make progress -- compute G(X) and G(Y) -- but without destroying the initial values of x and y until both values have been updated. One way to do this is to modify S1 so that the new values are computed and stored in some temporary variables, giving the following restartable action:

```
U1:  **action, recovery**
     {P:  $x = X \wedge y = Y$}
     U1a:  xnew:= G(x);     {$x = X \wedge xnew = G(X) \wedge y = Y$}
     U1b:  ynew:= G(y);     {$x = X \wedge xnew = G(X) \wedge y = Y \wedge ynew = G(Y)$}
     **end**
     {Q':  $x = X \wedge xnew = G(X) \wedge y = Y \wedge ynew = G(Y)$}
```

Note that in order to satisfy F4, x and y must be stored in stable storage but variables used in computing G need not be. Having established Q', it is a simple matter to establish Q:

```
S2:  {Q':  xnew = G(X)  ∧  ynew = G(Y)}
     S2a:  x:= xnew;    {x = xnew = G(X)  ∧  ynew = G(Y)}
     S2b:  y:= ynew;    {x = xnew = G(X)  ∧  y = ynew = G(Y)}
     {Q:  x = G(X)  ∧  y = G(Y)}
```

This is a replete proof outline, and provided xnew and ynew are stored in stable storage, F1 - F6 are satisfied. So

```
U2:  action, recovery
     {Q':  xnew = G(X)  ∧  ynew = G(Y)}
     U2a:  x:= xnew;    {x = xnew = G(X)  ∧  ynew = G(Y)}
     U2b:  y:= ynew;    {x = xnew = G(X)  ∧  y = ynew = G(Y)}
     end
     {Q:  x = G(X)  ∧  y = G(Y)}
```

is a restartable action. Thus, the desired program is:

U1; U2

4. Termination and Response Time

Most statements in our programming notation are guaranteed to terminate, once started. However, loops (**do**) and fault-tolerant actions are not. Techniques based on the use of variant functions or well-founded sets can be used for proving that a loop will terminate [Dijkstra 76]. Unfortunately, without knowledge about the frequency of failures and statement execution times, termination of a program written in terms of fault-tolerant actions cannot be proved. This is because if failures occur with sufficiently high frequency then there is no guarantee that the component fault-tolerant actions will terminate; neither the action statement nor the recovery protocol of a fault-tolerant action can be guaranteed to run uninterrupted, and so the recovery protocol could continually restart.

Moreover, such liveness properties [Lamport & Owicki 80] cannot even be expressed a Hoare-style programming logic, like the one above. Thus, we must resort to informal means to argue that a program will terminate in a timely manner. Presumably, at some point in the future it will be possible to formalize such arguments. [Harter & Bernstein 81] describe extensions to temporal logic [Lamport & Owicki 80] that allow construction of a proof that a program will meet some specific response-time goals. That work would have to be extended to deal with stochastically defined events for use in this context.

For a given execution of a program S on a fault-free processor, let t(s) be the maximum length of time that elapses once execution of statement s is begun until execution of the next fault-tolerant action in S is started. Define

$$T_{max} = \max_{s \in S} t(s).$$

For an execution of S to terminate at all, it is sufficient that there be (enough) intervals of length T_{max} during which there are no failures. Then, no fault-tolerant action will be forever restarted due to the (high) frequency of failures.

Of course, this gives no bound on how much time will elapse before S completes. Rather, we have argued that S is guaranteed to terminate if the elapsed time between successive failures is long enough, often enough. This should not be surprising. However, it does provide some insight into how to structure a program in terms of fault-tolerant actions if frequent failures are expected: one should endeavor to minimize T_{max}. This can be achieved by making entry into a fault-tolerant action a frequent event -- either by nesting fault-tolerant actions, or composing them in sequence.

Given a collection of fail-stop processors, it is possible to configure a system that not only implements a given relation between input and output, but performs this state transformation in a timely manner despite the occurrence of failures. After the failure of a fail-stop processor fsp, a _reconfiguration rule_ is used to assign programs that were running on fsp to working fail-stop processors. The recovery protocol in effect at the time of the failure facilitates restart of the program. Thus, processor failures are transparent except for possibly increased execution times.

As a result of a failure, execution delays from the following sources are incurred:

(1) Some time t_{detect} will elapse after the fail-stop processor halts until that fact is detected and reconfiguration is begun.

(2) Reconfiguration causes execution delays, as well. First, t_{recon} is required to determine an appropriate assignment of programs to the remaining fail-stop processors. Then, t_{move} might be required to move the program code and contents of its stable storage.

(3) In the worst case, the effects of the last T_A seconds worth of execution before the failure will be lost.

(4) An additional execution delay T_R might be incurred because a recovery protocol is likely to take longer to execute than an action statement.

Both T_A and T_R are defined for the specific execution that was interrupted.

This suggests the following strategy for constructing fault-tolerant systems that will continue to behave correctly in spite of up to k failures, for $k > 0$. A program is developed that (i) implements the desired state transformations when run on fail-stop processors, (ii) satisfies its real-time response constraints provided no failures occur, and (iii) in which no process must respond to an event in less than T_F seconds, where:

$$T_F = k\,(t_{detect} + t_{recon} + t_{move} + T_A) + T_R$$

Suppose R fail-stop processors are required to ensure (i) - (iii) hold. Then, a computing system with R+k fail-stop processors will be able to tolerate up to k fail-stop processor failures and meet its response-time goals. The obvious reconfiguration rule must be used.

Note that if stable storage that can be shared by k fail-stop processors is available, then t_{move} can be made 0. Also, by precomputing various configurations, t_{recon} can be made negligible. This, however, requires a sufficient amount of stable storage to store all possible configurations. Lastly, T_R can be made 0 by using only restartable actions; however, this uniformly degrades execution speed, even if no failures occur.

5. Fault-Tolerant Process-Control Software

We now turn to a more substantial illustration of the application of our methodology: development of a fault-tolerant process control program. First, a correct program for a fault-free computing system is developed. The program is then extended to run correctly on a system of fail-stop processors. While a fair amount of detail is presented, these details are necessary to derive and establish the correctness of the program.

Given are <u>sensors</u> to determine the state of the environment and <u>actuators</u> to exert control over the environment. Correct operation of a process-control system requires that:

PC: The values written to the actuators are related to the values read from the sensors according to a given application-specific function.

It is likely that correct operation also involves a liveness property, like "sensors are read and actuators are updated often enough". We will make no attempt to argue that our program satisfies such real-time response constraints, although informal arguments like those developed in section 4 could be used if timing data were available.

5.1. Assuming No Failures

Our process-control system will be structured as a collection of cyclic processes that execute concurrently. Each process p_i is responsible for controlling some set of actuators act_i. To do so, it reads from some sensors and maintains $state_i$ -- a vector of state variables that reflects the sensor values p_i has read and the actions it has taken. Interprocess communication is accomplished by the disciplined use of shared variables; a process can read and write its state variables, but can only read state variables maintained by other processes. For the moment, we will ignore the problems that arise from concurrent access to state variables.

Each process will consist of a single loop. During execution of its loop body, process p_i: (1) reads from some sensors, (2) computes new values for the actuators it controls and state variables it maintains, (3) writes the relevant values to act_i and (4) updates $state_i$. Presumably, we are given application dependent routines that can be used to compute the values to be written to the actuators and the values to be stored in the state variables.

Without loss of generality, assume that each state variable and sensor is read at most once in any execution of those routines[5]. Then, let var[i,t] denote the value of var read by p_i during the t^{th} execution of its loop body, sensors[i,t] denote the values read by p_i from sensors during the t^{th} execution of its loop body,

[5]Code that satisfies this restriction can be written by using local variables to store state variables and sensor values: each state variable and sensor value is stored in a local variable when it is first read; subsequent references are then made to the local variable.

and $act_i[t]$ denote the values written to act_i by p_i during the t^{th} execution of the loop body.

Behavior satisfying PC is characterized by the following, for each process p_1, p_2, ..., p_n.

First, the values in $state_i$ must correctly encode past actions performed by p_i. That encoding will be denoted here by the function E. Therefore, at the beginning of the $t+1^{st}$ execution of the loop body at p_i:[6]

$$\text{Istate}(i,t):\ t=0\ \textbf{cor}\ state_i = E(sensors[i,t], state_1[i,t] \ldots, state_n[i,t]).$$

Secondly, the values written to actuators by p_i must be computed according to the application-specific function, here called A, based on the sensor values read and the past actions of processes. Therefore, after p_i updates act_i for the t^{th} time,

$$\text{Iact}(i,t):\ t=0\ \textbf{cor}\ act_i[t] = A(E(sensors[i,t], state_1[i,t] \ldots, state_n[i,t])).$$

must be true.

Let T_i be an auxiliary variable defined so that at any time T_i-1 executions of the loop body have completed. Thus, T_i is initialized to 1 and (implicitly and automatically) incremented immediately after the loop body is executed. Then, the correctness criterion PC is satisfied if:

$$I(i):\ \text{Istate}(i,T_i-1)\ \wedge\ \text{Iact}(i,T_i-1)$$

is true at the beginning of each execution of the loop body, for each process p_i.

[6] We use the notation "A **cor** B" to mean "**if** A **then** true **else** B".

In order to construct the loop, variable newstate is introduced. This is necessary so that values used to update $state_i$ and the actuators are consistent with each other. Thus,

$$Vnewstate(i,t):\ newstate = E(sensors[i,t], state_1[i,t] \ldots state_n[i,t]).$$

The loop at process p_i, which has as I(i) as its loop invariant, is:

```
p_i: process
     do true → {I(i)}
          calc: newstate:= E(sensors,state_i, ..., state_n);
               {Vnewstate(i,T_i) ∧ Istate(i,T_i-1) ∧ Iact(i,T_i-1)}
          up_act: act_i:= A(newstate);
               {Vnewstate(i,T_i) ∧ Istate(i, T_i-1) ∧ Iact(i, T_i)}
          up_st: state_i:= newstate;
               {Vnewstate(i,T_i) ∧ Istate(i, T_i) ∧ Iact(i, T_i)}
     od
   end
```

However, because processes execute asynchronously, access to state variables must be synchronized. Otherwise, a process might read state variables while they are in the midst of being updated, which could cause the process to perform the wrong actions. To avoid this problem, the state variables maintained by each process p_i are assumed to be characterized by CC_i, called the consistency constraint for $state_i$. CC_i is kept true of $state_i$ except while p_i is updating those variables -- i.e. performing up_st above. We assume that the code to compute the application dependent functions A and E works correctly as long as values that satisfy the consistency constraints are read. To ensure that only values satisfying the consistency constraints are read, read/write locks [Gray 78] can be used to implement reader-writer exclusion on the state variables maintained by each process. A process trying to read variables in $state_i$ must first acquire a read lock for $state_i$.

Such a lock will not be granted if a write lock is already held for those state variables, hence that process will be delayed if $state_i$ is being updated. A process about to update $state_i$ will be delayed if other processes are reading those values. Such lock operations are not explicitly included in our programs to simplify the exposition; they are part of the routine to compute E in "calc" and "up_st", the routine to update the state variables.

Similarly, we assume that the code to compute A and E requires that sensor values used be consistent. The natural laws that govern our physical world ensure that at any time t the values of the sensors are consistent. Thus, if a process reads all the sensors simultaneously, consistent values would be obtained. Such a simultaneous read operation is not implementable, however. We therefore assume that sensors change values slowly enough and that processes execute quickly enough so that a consistent set of values is obtained by reading each of the sensors in sequence at normal execution speed.

5.2. Allowing Failures

We shall deal with failures by attempting to mask their effects. Thus, we shall endeavor to preserve:

PC': At no time do state variables or actuators have values they could not have had if the failure had not occurred.

Recall that I(i) characterizes values of the state variables and actuators that satisfy PC. Consequently, if it is possible to modify the loop body so that I(i) is true of every state that could be visible after a failure then PC' will be satisfied, as well. Our task, therefore, is to modify the loop body so that it constitutes a restartable action.

I(i) is true except between the time execution of statement up_act begins and when statement up_st completes. Thus, we must either mask intermediate states during execution of up_st and up_act, or devise a way to execute up_st and up_act together as an atomic action. This latter option is precluded by most hardware. Thus, to implement the former, we construct a single fault-tolerant action that updates the actuators and state variables based on the value of newstate:

```
{Vnewstate(i,$T_i$)}
upall
{Vnewstate(i,$T_i$)  ∧  Istate(i,$T_i$)  ∧  Iact(i,$T_i$)}
```

As long as newstate is saved in stable storage, the following replete proof outline satisfies F1 - F6 and accomplishes the desired transformation.

```
upall: **action, recovery**
       {Vnewstate(i,$T_i$)}
       up_act: $act_i$:= A(newstate)
       {Vnewstate(i,$T_i$)  ∧  Iact(i, $T_i$)}
       up_st: $state_i$:= newstate;
       {Vnewstate(i,$T_i$)  ∧  Istate(i, $T_i$)  ∧  Iact(i, $T_i$)}
       **end**
```

A replete proof outline for the code executed at p_i is:

```
p_i: **process**
     **action,recovery**
      **do** true → {I(i)}
           calc:  newstate:= E(sensors,state_i, ...state_n);
           {Vnewstate(i,T_i) ∧ Istate(i,T_i-1) ∧ Iact(i,T_i-1)}
           upall: **action, recovery**
              up_act: act_i:= A(newstate);
              up_st: state_i:= newstate;
              **end**
      **od**
     **end**
```

Notice that following a failure, a process might attempt to acquire a given read/write lock that had already been granted to it. For example, if a failure occurred while up_st were being executed, the recovery protocol would attempt to acquire the write lock on $state_i$, which might already be owned by p_i. Clearly, repeated requests by a given process for the same lock, without intervening release operations, should not delay the invoker. Implementation of read/write locks with this property (binary semaphores do not suffice) is possible and is described in [Schlichting 82].

6. Discussion

6.1. Related Work

Few general techniques have been developed to aid in the design of programs that must cope with operational failures in hardware or support software. One paradigm, based on the use of state machines, was pioneered by Lamport [Lamport 81] [Schneider 82]. A program is viewed as a state machine that receives input, generates actions (output) and has an internal state. A reliable system is constructed by replicating these state machines and running them in parallel. By using a solution to the Byzantine Generals Problem, each machine is guaranteed to receive the same input, despite failures. A second general paradigm, which appears to be promising, is based on the use of nested atomic transactions [Lampson 81].

A variety of protocols for specialized problems have also been developed. Included are: protocols for recovery in data base systems [Gray 78], implementation of highly reliable file systems [Lampson & Sturgis 78] and the use of checkpoint/restart facilities in operating systems [Denning 76].

Despite the apparent similarity between the recovery block construct developed at the University of Newcastle-upon-Tyne [Randell _et al_ 78] and our fault-tolerant actions, the two constructs are intended for very different purposes. A _recovery block_ consists of a _primary block_, an _acceptance test_, and one or more _alternate blocks_. Upon entry to a recovery block, the primary block is executed. After its completion, the acceptance test is executed to determine if the primary block has performed acceptably. If the test is passed, the recovery block terminates. Otherwise, an alternate block -- generally a different implementation of the same algorithm -- is attempted and the acceptance test is repeated. Execution of each alternate block is attempted in sequence until one produces a state in which the acceptance test succeeds. Execution of an alternate block is always begun in the recovery block's initial state.

Recovery blocks are used to mask design errors; fault-tolerant actions are used in constructing programs that must cope with _operational failures_ in the underlying hardware and software. As such, use of recovery blocks to cope with operational failures can only lead to difficulties. For example, a recovery block has only a finite number of alternate blocks associated with it, and therefore a large number of failures in the underlying system can cause the available alternatives to be exhausted. Secondly, the initial states of variables modified by a recovery block must be available when execution of an alternate block is begun. Thus, the model does not admit the possibility of using volatile storage for program variables, since those values cannot be recovered after a failure.

6.2. Whence Fail-Stop Processors

The definition of the fail-stop processor as our underlying computational model, followed from our use of a partial correctness programming logic. In a fail-stop processor all failures are detected and no incorrect state transformations result from failures. Thus, if execution of a statement terminates, by definition the transformation specified by that statement has occurred -- the effect of execution is consistent with the programming logic. On the other hand, failure, by definition, prevents statements from terminating. Thus, the partial correctness (as opposed to total correctness) nature of the programming logic subsumes the consequences of failures.

6.3. Application of the Methodology

We have successfully employed the methodology described in this paper both to verify existing fault-tolerant protocols and to devise new ones. In [Schlichting 82], the two-phase commit protocol as described in [Gray 78] is verified. The process-control example described in section 5 of this paper was developed as part of a project to apply this methodology to design a distributed computing system for navigation in an airplane. The details of that work are discussed in [Schneider & Schlichting 81].

It is natural to ask whether F1 - F6, the components of our proof rule for fault-tolerant actions, are too restrictive. In that case there would exist fault-tolerant actions that would behave correctly, but for which no proof would be possible. While we have not proved the relative completeness of our new rule, the success we have had with its application and the way in which it was derived, suggest F1 - F6 are not too restrictive to allow proof of any "correct" fault-tolerant action.

7. Conclusions

We have described a methodology for constructing fault-tolerant systems. It is based on the notion of a fail-stop processor -- a processor with simple and well defined failure-mode operating characteristics. Fail-stop processors are a very appealing abstract machine to program and can be approximated by real hardware.

We have also shown how axiomatic program verification techniques can be extended for proving the correctness of programs written for fail-stop processors. This allows a programmer to argue convincingly about the correctness of a program *ex post facto*. More importantly, it allows a programmer to develop a fault-tolerant program and its proof hand-in-hand, with the latter leading the way, as advocated in [Dijkstra 76] [Gries 81]. Computing the (weakest) precondition of a recovery protocol is a simple and mechanical way to determine what program variables must be stored in stable storage; constructing a replete proof outline similarly defines what intermediate states could be visible following a failure and thus what states can be seen by a recovery protocol.

Acknowledgements

This work has benefitted from discussions with G. Andrews, R.W. Conway, R. Constable, E.W. Dijkstra, L. Lamport, G. Levin, R. Reitman, D. Wall and S. Worona. The process-control application was first suggested by J. Kemp, W. Comfort and M. Kushner of IBM (FSD/Owego). D. Gries made very helpful comments on an earlier draft of this paper. We would also like to thank the referees for a very careful reading of this paper.

References

[Avizienis 76]
Avizienis, A. Fault-Tolerant Systems. IEEE Transactions on Computers Vol. C-25, No. 12 (December 1976), 1304-1312.

[Barlow & Proschan 75]
Barlow, R.W., F. Proschan. Mathematical Theory of Reliability. Wiley, New York, 1965.

[Denning 76]
Denning, P. Fault Tolerant Operating Systems. Computing Surveys 8, 4 (December 1976), 359-389.

[Dijkstra 76]
Dijkstra, E.W. A Discipline of Programming. Prentice Hall, 1976.

[Gray 78]
Gray, J. Notes on Data Base Operating Systems. Operating Systems An Advanced Course, Lecture Notes in Computer Science, Volume 60, Springer-Verlag, 1978.

[Gries 81]
Gries, D. The Science of Programming. Springer-Verlag, New York, 1981.

[Harter & Bernstein 81]
Harter, P., A. Bernstein. Proving Real Time Properties of Programs with Temporal Logic. Proceedings of SOSP-8, Asilomar, Cailfornia (Dec 1981).

[Hoare 69]
Hoare, C.A.R. An Axiomatic Basis for Computer Programming. CACM 12, 10 (October 1969), 576-580.

[Hoare & Wirth 73]
Hoare, C.A.R., N. Wirth. An Axiomatic Definition of the Programming Language PASCAL. Acta Informatica 2 (1973), 335-355.

[Hopkins et al 78]
Hopkins, A.L., T.B. Smith, J.H. Lala. FTMP -- A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft. Proc. of the IEEE, Vol. 66, No. 10 (October 1978), 1221-1239.

[Juvenal 130]
Juvenal (Decimus Junius Juvenalis, c.50 -c.130). Satires VI, line 347.

[Lamport 78]
Lamport, L. Time, Clock and the Ordering of Events in a Distributed System. CACM 21, 7 (July 1978), 558-565.

[Lamport et al. 80]
Lamport, L., R. Shostak, M. Pease. The Byzantine Generals Problem. Technical Report 54, SRI International, March 1980.

[Lamport 81]
Lamport, L. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. Technical Report 59, SRI International, June 1981.

[Lamport & Owicki 80]
Lamport, L., S. Owicki. Proving the Liveness Properties of Concurrent Programs. Technical Report 57, SRI International, October 1980.

[Lampson & Sturgis 78]
Lampson, B., H. Sturgis. Crash Recovery in a Distributed Data Storage System. Submitted to CACM.

[Lampson 81]
Lampson, B. Atomic Transactions. Distributed Systems -- Architecture and Implementation, Lecture Notes in Computer Science Vol. 105, Springer-Verlag, 1981.

[Pease et al 79]
Pease, M., R. Shostack, L. Lamport. Reaching Agreement in the Presence of Faults. JACM 27, 2 (April 1979).

[Pnueli 79]
Pnueli, A. The Temporal Semantics of Concurrent Programs. Semantics of Concurrent Computation, Lecture Notes in Computer Science Volume 70, Springer Verlag, 1979.

[Randell et al. 78]
Randell, B., P.A. Lee, P.C. Treleaven. Reliability Issues in Computing System Design, Computing Surveys 10, 2 (June 1978), 123-165.

[Schlichting 82]
Schlichting, R.D. Axiomatic Verification to Enhance Software Reliability. Ph.D. Thesis, Dept. of Computer Science, Cornell Univeristy, Jan. 1982.

[Schneider 82]
Schneider, F.B. Synchronization in Distributed Programs. TOPLAS 4 2 (April 1982), 125-148.

[Schneider & Schlichting 81]
Schneider, F.B., R.D. Schlichting. Towards Fault-Tolerant Process Control Software. Proc. Eleventh Annual International Symposium on Fault-Tolerant Computing, IEEE Computer Society, Portland, Maine, (June 1981) 48-55.

[Schlichting & Schneider 82]
Schlichting, R.D, F.B. Schneider. Using Message-Passing for Distributed Programming: Proof Rules and Disciplines. Technical Report TR 82-491, Department of Computer Science, Cornell Univeristy, Ithaca, New York, May 1982.

[Siewiorek & Swarz 82]
Siewiorek, D., R.S. Swarz. The Theory and Practice of Reliable System Design. Digital Press, Bedford, Mass. 1982.

[Wensley et al 78]
Wensley, J., et al. SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. Proc. of the IEEE, Vol. 66, No. 10 (October 1978) 1240-1255.