Sādhanā, Vol. 11, Parts 1 & 2, October 1987, pp. 7–22. © Printed in India.

A tutorial on the principles of fault tolerance

S K SHRIVASTAVA

Computing Laboratory, University of Newcastle upon Tyne NE1 7RU, UK

Abstract. The paper begins by examining the four aspects of fault tolerance – error detection, damage assessment, error recovery and fault treatment – and describes how these aspects can be incorporated in systems. Following this, a methodology for the construction of robust software systems is presented, covering the topics of design fault tolerance and software implemented fault tolerance. Some aspects of modelling faulty behaviour of components is presented and the notion of a family of fault-tolerant algorithms is introduced.

Keywords. Error recovery; fault tolerance; reliability; exception handling; fault classification; atomic actions; replicated processing; real time systems.

1. Introduction

A reliable computing system must be capable of providing normal services in the presence of a finite number of component failures. Faults within a system cause its failure. These faults could be present in either the components of the system or in its design. The paper examines in §2 the nature of systems and their failures and presents a methodology for the construction of robust software modules – modules capable of tolerating both expected and unexpected faults. The next section discusses design fault tolerance and the subsequent section discusses software implemented fault tolerance, describing the principles of constructing algorithms capable of tolerating component failures of specified types. Conclusions from our study are presented in the last section.

2. Systems and their failures

Following Anderson & Lee (1981, 1982), a *system* is defined to consist of a set of components which interact under the control of an algorithm (or design). The *components* of a system are themselves systems as is the *algorithm* (*design*). The phrase 'algorithm of a system' is used here to refer to that part of the system which actually supports the interactions of the components.

The *internal state* of a system is the aggregation of the external states of all its components. The *external state* of a system is an abstraction of its internal state.

During a transition from one external state to another, the system may pass through a number of internal states for which the abstraction, and hence the external state is not defined. We assume the existence of an *authoritative specification* of behaviour for a system which defines the external states of the system, the operations that can be applied to the system, the results of these operations and the transitions between external states caused by these operations.

In our everyday conversations we tend to use the terms 'fault', 'failure' and 'error' (often interchangeably) to indicate the fact that something is 'wrong' with a system. However, in any discussion on reliability and fault tolerance, a little more precision is called for to avoid any confusion. *Failure* of a system is said to occur when the behaviour of the system first deviates from that required by the specification. The *reliability* of the system can then be characterized by a function $R(t)$ which expresses the probability that no failure of the system will have occurred by time t. We term an internal state of a system an *erroneous state* when that state is such that there exist circumstances (within the specification of the use of the system) in which further processing by the normal part of the system will lead to failure. The phrase 'normal part of a system' is used here to admit the possibility of introducing in the system extra components and algorithms to specifically prevent failures. Such additions are referred to as the *redundant* (*exceptional* or *abnormal*) part of the system. The term 'error' is used to designate that part of the internal state that is 'incorrect'. The terms 'error', 'error detection' and 'error recovery' are used as casual equivalents for 'erroneous state', 'erroneous state detection' and 'erroneous state recovery'.

Next we might ask why a system enters an erroneous state (one that leads to a failure). The reason for this could be either the failure of a component or the design (or both). Naturally, a component (or design) being a system, may itself fail because of its internal state being erroneous. It is often convenient to be able to talk about causes of system failure without actually referring to internal states of the system's components and design. We achieve this by referring to the erroneous state of a component or design as a *fault* in the system. A fault could either be a *component fault* or a *design fault*; so a component fault can result in an eventual component failure and similarly a design fault can lead to a design failure. Either of these internal (to a system) failures will cause the system to go from a valid state to an erroneous state; the transition from a valid to an erroneous state is referred to as the *manifestation of a fault*.

To summarize: a system fails because it contains faults; during the operation of a system a fault manifests itself in the form of the system state going into an erroneous state such that – unless corrective actions by the redundant part of the system are undertaken – a system failure will eventually occur.

3. Principles of fault tolerance

Two complementary approaches have been noted for the construction of reliable systems (Avizienis 1976). The first approach, which may be termed *fault prevention*, tries to ensure that the implemented system does not and will not contain any faults. Fault prevention has two aspects:

(i) *fault avoidance* techniques are employed to avoid introducing faults into the

system (e.g. system design methodologies, quality control);
(ii) *fault removal* techniques are used to find and remove faults which were inadvertently introduced into the system (e.g. testing and validation).

The second approach, which has been termed *fault tolerance*, is of special significance to us because of the impracticality of ensuring the complete absence of faults in a system containing a large number of components. Four constituent phases of the fault-tolerance approach have been identified: (i) error detection; (ii) damage assessment; (iii) error recovery; and (iv) fault treatment and continued system service.

3.1 *Error detection*

In order to tolerate a fault, it must first be detected. Since internal states of components are not usually accessible, a fault cannot be detected directly, and hence, its manifestations, which cause the system to go into an erroneous state, must be detected. Thus the usual starting point for fault-tolerance techniques is the detection of errors.

3.2 *Damage assessment*

Before any attempt can be made to deal with the detected error, it is usually necessary to assess the extent to which the system state has been damaged or corrupted. If the delay, identified as the *latency* interval of that fault, between the manifestation of a fault and the detection of its erroneous consequences is large, it is likely that the damage to the system state will be more extensive than if the latency interval were shorter.

3.3 *Error recovery*

Following error detection and damage assessment, techniques for error recovery must be utilized in an attempt to obtain a normal error-free system state. In the absence of such an attempt (or if the attempt is not successful) a failure is likely to ensue. There are two fundamentally different kinds of recovery techniques. The *backward recovery technique* consists of discarding the current (corrupted) state in favour of an earlier state (naturally, mechanisms are needed to record and store system states). If the prior state recovered to, preceded the manifestation of the fault, then an error free state will have been obtained. In contrast a *forward recovery technique* involves making use of the current (corrupted) state to construct an error free state.

3.4 *Fault treatment and continued service*

Once recovery has been undertaken, it is essential to ensure that the normal operation of the system will continue without the fault immediately manifesting itself once more. If the fault is believed to be transient, no special actions are necessary, otherwise, the fault must be removed from the system. The first aspect of fault treatment is to attempt to locate the fault; following this, steps can be taken to either repair the fault or to reconfigure the rest of the system to avoid the fault.

To illustrate the ideas presented so far, let us examine the recovery block mechanism (Horning *et al* 1974; Randell 1975), a well-known method of constructing fault-tolerant software. The syntax of a recovery block construct is,

ensure ⟨ acceptance test ⟩ **by** P_0 **else-by** P_1 **else** fail;

which depicts a software system with four components, the two procedures P_0 (the primary) P_1 (the alternative), the acceptance test and the set of global variables accessible to the procedures (not shown above). The algorithm of the system is the control structure implied by the syntax. If we assume that the acceptance test is 'perfect' (i.e. detects all violations of the specification) then the recovery block shown can tolerate faults within procedure P_0 (if any) that could lead to its failure, provided of course, that P_1 passes the test. Regarding P_0 as a system, its faults are essentially design faults. So, when it is said that 'a recovery block can tolerate design faults', what is really meant is that it can tolerate faults in some of its components (P_0 in our case) which could fail due to design faults in them. We shall next see how the four aspects of fault tolerance are embodied in a recovery block. The acceptance test (a boolean expression) is used for detecting errors. Damage assessment is particularly simple: only the component in execution is assumed to be affected. (We are assuming the simple case of a single sequential process; when interacting processes are involved, damage assessment can be quite difficult, Randell 1975.) Error recovery – backward in this case – consists of recovering the state of the executing program to that at the beginning of the recovery block. Finally, the program in execution (primary or alternative) is assumed to be faulty, so its faults are avoided by executing the next alternative (if any).

The four aspects of fault tolerance form the basis for all fault-tolerance techniques and provide a sound foundation for design and implementation of reliable systems (Anderson & Lee 1981).

4. Software design methodology

In this section we will present a methodology for the construction of robust software systems based on the treatment presented in Anderson & Lee (1981) and Cristian (1982). Following generally accepted software engineering concepts, we shall assume the use of data abstractions (abstract data types) in program development. This leads to software systems that are structured into a hierarchy of modules (or components). Such a hierarchy may be represented by an acyclic graph (figure 1) where modules are represented by nodes and an arrow from a node A to a node B means that A is a *user* of B; that is, there are one or more operations in A such that a successful completion of one such operation depends on the successful

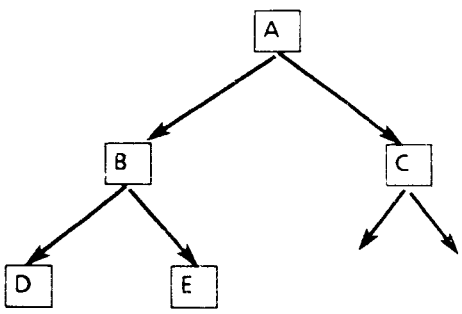

Figure 1. Hierarchy of modules.

completion of some operation provided by B – in other words, B provides certain *services* to A.

4.1 *Expected events*

The specified services provided by a given module can be classified into *normal services* (expected and desired) and *abnormal* or *exceptional services* (expected but undesired). In programming language terms, when a user module *calls* a procedure exported by a lower level module, then either the call terminates normally (expected desired service is obtained) or an exceptional return is obtained. Let us now consider the design of an intermediate module such as B (see figures 1 & 2).

A normal chain of events might consist of some procedure of A making a call on B, as a result of which B calls a lower level module (say E), this call returns normally, and subsequently A's call returns normally. We examine now the two cases that could lead to A's call returning exceptionally.

(i) A call to a lower level module (such as E) by B returns exceptionally. In such a case we say that an *exception* is *detected* in B (this is synor ymous to saying that an error is detected in B; we will use the term 'exception' here because it is more commonly used when talking about software). If this exception is not 'handled', then module B would certainly fail to provide the specified service to A. To cope with the detected exceptions, module B therefore contains *exception handlers* (the handlers thus represent the 'abnormal' part of a system mentioned earlier). If, despite the occurrence of a lower level exceptional return, module B provides a normal service to A, we say that the lower level exception has been *masked* by the handler in B. On the other hand, if B is unable to mask a lower level exception and provides an exceptional return to A, we say that the lower level exception has been *propagated* to a higher level.

(ii) A boolean expression in B – inserted specifically for detecting an error (exception) – evaluates to false. The treatment of this exception by its handler is similar to the previous case: either that exception is masked, in which case (provided no further exceptions are encountered) B will return normally to A, otherwise an exceptional return is obtained by A.

We thus see that the construction of a robust module requires the provision of (a) exception handlers for coping with exceptions propagated from lower levels; and (b) boolean expressions for detecting exceptions arising in the module itself,

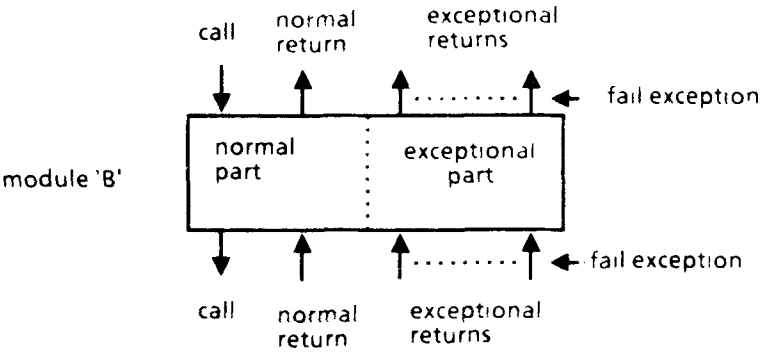

Figure 2. Structure of a module.

and their exception handlers. Note that it is possible (and often desirable for the sake of simplicity) to map several exceptions onto a single handler.

The need for exception handling facilities in programming languages has now been recognised and many modern languages such as CLU (Liskov & Snyder 1979) and ADA (Luckham & Polak 1980) contain specific features for exception handling. We shall use here some simple notations which will enable us to illustrate these ideas with the help of a few examples. The following notation will be used to indicate that a procedure P, in addition to the normal return, also provides an exceptional return E:

procedure P(- -) **signals** E;.

The invoker of P can define the exceptional continuation to be some operation H which will be termed the *handler* of E:

P(- -) [$E \Longrightarrow H$];.

In the body of P, the designer of P can insert the following syntactic constructs (where the braces indicate that the signal operation is optional):

(a) [$B \Longrightarrow \ldots.$; {**signal** E}];

(b) Q[$D \Longrightarrow \ldots.$; {**signal** E}];.

Construct (a) represents the case whereby an exception is detected by a run time test; whilst the second construct represents the case when invocation of an operation Q results in an exceptional return D which in turn could lead to the signalling of exception E. When an exception is signalled using construct (a) or (b), the control passes to the handler of that exception (H in this case).

Example: We consider the design of a procedure P which adds three positive integers. The procedure uses the operation '+' (typically provided by the hardware interpreter) which can signal an overflow exception OV.

procedure P(**var** i: integer; j, k: integer) **signals** OW;

begin

$i := i+j$[$OV \Longrightarrow$ **signal** OW];

$i := i+k$[$OV \Longrightarrow i := i-j$; **signal** OW];

end;.

It is assumed above that no assignment is performed if an exception is detected during the execution of the operation '+'.

The above example also illustrates an important aspect of exception handling, which is that before signalling an exception it is often necessary to perform a 'clean up' operation. The most sensible strategy is to 'undo' any side effects produced by the procedure. If all the procedures of a module follow this strategy, we get a module with the following highly desirable property: either the module produces results that reflect the desired normal service to the caller, or no results are produced and an exceptional return is obtained by the caller.

Example: A file manager module exports a procedure CREATE whose function is to create a file containing n blocks. Assume that the file manager employs two discs

for block allocation such that a given file has its blocks on either disc d_1 or d_2, and that M_1 and M_2 are the disc manager modules for d_1 and d_2 respectively.

procedure CREATE (n: integer) **signals** NS;

begin

$M_1 \cdot AL(n)[DO \Longrightarrow M_2 \cdot AL(n)[DO \Rightarrow$ restore; **signal** $NS]]$;

---

end;.

The above procedure illustrates how an exception may be masked. The AL procedure of a disc manager allocates n blocks, but if the number of free blocks is less than requested, a disc overflow exception (DO) is signalled. The first handler of this exception tries to get space from the second disc manager. If a second DO exception is detected then the procedure is exited with a 'no space' exception NS. The procedure 'restore' recovers the state of global variables accessible to CREATE to that at the beginning of the call (this follows from our philosophy of undoing any side effects before signalling an exception).

4.2 *Unexpected events*

So far we have considered the treatment of 'expected events' (desired or undesired); we turn our attention to the treatment of unexpected (and therefore undesired) events. Let us assume that the hardware interpreter over which the software under consideration is executing is behaving according to the specification. Then, any unexpected behaviour from a software module must be attributed to the existence of one or more design faults in that module or any of its lower level modules. In general, during the execution of a procedure P of a module, a design fault can manifest itself in any of the following ways:

1) the execution of P does not terminate;
2) a lower level exception is detected for which there is no exception handler in P;
3) the execution of P terminates normally (the invoker obtains a normal return) but the results produced by P are not in accord with the specification.

It is clear that situations (1) and (2) will eventually cause a failure of the module; situation (3) represents the case where the module has failed but this event has not yet been detected by the system. To cope with such cases, we can employ a *default exception handler*:

procedure P(- -) **signals** E;

begin

---

end [$\Longrightarrow$ "default handler"];.

The control goes to this handler during the execution of P whenever an exception is detected for which there is no handler. Thus, to cope with situation (1) it is possible to start a 'timer' concurrently with the invocation of P; the 'time out' exception will then be handled by the default handler. All the lower level exceptions with no programmed handlers will similarly be handled by the default handler. Finally we

make use of run time checks (assertions) to detect possible violations of specifications to minimise the danger of undetected failures (case 3).

What should be the strategy adopted by a default handler? The simplest thing to do is to undo any side effects produced by the procedure and to signal a *fail* exception (see figure 2). When the invoker receives a fail exception, it means that the called module has failed to provide the specified service. Nevertheless, the called module has failed 'cleanly' since no side effects have been produced. It is also possible for the default handler to mask the (unanticipated) exception by calling an alternative procedure in the hope of circumventing the design fault(s). The similarity with the recovery block approach is not accidental, as the example below shows how a recovery block can be modelled by making use of default exception handlers:

ensure ⟨ acceptance test ⟩ **by** P_0 **else-by** P_1 **else** fail;.

The above construct is equivalent to the following one:

$P_0'[\Longrightarrow$ restore; $P_1'[\Longrightarrow$ restore; **signal** fail]];

where, P_i', $i = 0, 1$, is given by:

procedure P_i'

begin

body of P_i;

assert ⟨ acceptance test ⟩;

end [$\Longrightarrow$ **signal** fail];.

The following design methodology has then emerged. During the design of a given module, we carefully analyse the cases that could prevent the module from providing the desired normal services. We make use of specific exception handlers to either mask the effects of such undesired but expected exceptions or to signal an appropriate exception to the caller of the module; the purpose of signalling an exception is to indicate to the caller that the normal service cannot be provided and also to give an indication of the reason (e.g. arithmetic overflow, disc full, fail etc.). We make use of default exception handlers or recovery blocks to obtain a measure of tolerance against design faults. The capability of tolerating design faults rests largely on the 'coverage' of run time checks (such as acceptance tests) for detecting errors. Often, for reasons of efficiency, it is not possible to check completely within a procedure that the results produced have been according to the specification (e.g. for a routine that sorts its input, the check that the output has been sorted would be almost as complex as the routine itself); hence run time checks are often limited to checking certain critical aspects of the specification (hence the name 'acceptance test'). This means that the possibility of undetected failures cannot be ruled out entirely.

5. Tolerance for design faults

The difficulty in providing tolerance for design faults is that their consequences are unpredictable. As such, tolerance can only be achieved if *design diversity* has been

built into the system (figure 3). In the last section, recovery blocks (or default exception handlers) were mentioned as a mechanism for introducing design diversity. In this section the concept of design diversity is explored further. One more proposal, in addition to recovery blocks, has been made for tolerating design faults in software, and is known as *N-version programming* (Avizienis 1985). Both the approaches can be described uniformly using the diagram given below (Lee & Anderson 1985; pp. 64–77).

Each redundant module has been designed to produce results acceptable to the adjudicator. Each module is independently designed and may utilize different algorithms as chosen by its designer. In the *N*-version approach, the adjudicator is essentially a majority voter (the scheme is analogous to the hardware approach known as the *N*-modular redundant technique). The recovery block scheme has an adjudicator which applies an acceptance test to each of the outputs from the modules in turn, in a fixed sequence.

The operational principles of the *N*-version approach are straightforward: all of the *N* modules are executed in parallel and their results are compared by a voting mechanism provided by the adjudicator. The implementation of this scheme requires a *driver program* which is necessary for: (i) invoking each of the modules; (ii) waiting for the modules to complete their execution; and (iii) performing the voting function. Each module must be executed without interference from other modules. One way of achieving this goal is to physically separate the modules – each module is run on a separate processor.

A special case of *N*-version programming is when the degree of replication is just two. In this case the adjudicator provides a comparison check. The Airbus A310 slat and flap control system (Martin 1982) uses this approach, for driving stepping motors via a comparator. In the event of a discrepancy, the motors are halted, the control surfaces locked and the flight crew alerted.

Experiments conducted at UCLA and elsewhere on *N*-version programming (Avizienis 1985; Knight & Leveson 1986) and at Newcastle on recovery blocks (Anderson *et al* 1985) have produced encouraging results indicating that tolerance to design faults is certainly possible.

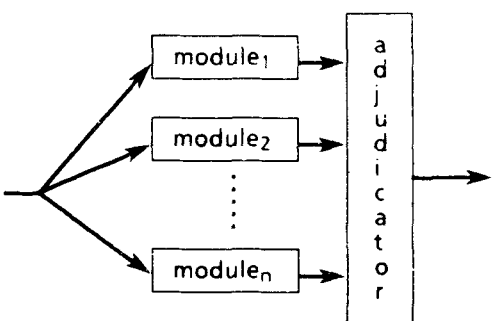

Figure 3. Design diversity

6. Software implemented fault tolerance

6.1 *Tolerance to hardware faults*

The techniques presented in §4 can be applied to the case when lower level modules have been implemented in hardware (e.g. disc units, processors). If these modules also provide normal and exceptional services (which is usually the case) then higher level software modules which use them can employ the fault-tolerance techniques discussed previously to either mask a lower level hardware exception or to propagate it as a higher level exception. The term *software implemented fault tolerance* is often used to refer to software techniques for tolerating hardware faults. The resulting algorithms will be termed *fault-tolerant algorithms*. When dealing with hardware the following points must be borne in mind:
(i) An exceptional response is often obtained due to a transient fault in the hardware; thus simply retrying the operation may prove to be sufficient.
(ii) All hardware components eventually fail (due to ageing and wearout); so when a failure of a hardware module is suspected, steps might be required to permanently remove the module from the system (reconfigure the system).
(iii) Diagnostic techniques can be used in an operational system to detect possible failures of components and to repair (replace) them before these components are utilised for services.

The function of a fault-tolerant algorithm of a system is to detect failures of the system's components and to attempt to tolerate these failures so as to provide specified services.

Example. Construction of reliable disc storage out of unreliable discs. A disc can fail (permanently) due to defective disc surface conditions, failure of the disc drive system or failure of the read-write electronics. In addition, various other accidents can occur which can cause data stored in one or more pages of a disc to be corrupted. Here we will briefly discuss tolerance to these latter kind of failures.

We assume the existence of the following two hardware procedures for accessing a disc:

procedure write (at: address; data: page);

procedure read (at: address; **var** data: page) **signals** looksbad;.

The exception 'looksbad' indicates that the data read could be corrupted. This could either be because the page is really corrupted or some transient failure has occurred – in which case a bounded number of retries should eventually result in good data being read. The effect of a write operation is that either (i) the addressed page gets the data; or (ii) the addressed page remains unchanged or gets corrupted data.

We next construct fault-tolerant read and write operations using the unreliable operations mentioned above:

procedure careful-read (at: address; **var** data: page) **signals** bad-page;

begin

use read operation at most n number of times to obtain good data (i.e. not looksbad) **else** signal bad-page;

end;

procedure careful-write (at: address; data: page) **signals** bad-page

begin

perform 'write' and then 'careful-read' on the same page to check written data = read data; **if** the check fails even after *n* retries **then** signal bad-page

end;.

One way we can guard against accidental corruption of a page is by making sure that an uncorrupted copy of the page is available somewhere. This can be achieved by employing two discs (with independent failure modes) and by maintaining pairs of pages on these discs. It is then necessary to check at regular intervals that the pairs of pages have identical uncorrupted data stored in them; if not, the corrupted page of a pair is updated by performing a careful read on the paired page followed by a careful write on the corrupted page. The interval of running this checking process is chosen so as to reduce the probability of both the pages of a pair becoming corrupted to an acceptably small quantity (Lampson & Sturgis 1981).

6.2 *Modelling faulty behaviour of components*

The simple example of the previous sub-section illustrates how, given a specification of abnormal behaviour of components, specific measures can be employed in fault-tolerant algorithms. The fault-tolerance measure employed by 'careful-read' (namely, repeated retries) will only be effective, when a read operation fails by reading the addressed page in a detectably incorrect manner (exception looksbad is signalled). Clearly, the employed measure will not be effective if a disc fails, say, by correctly reading a page other than the intended one. Thus, design of a fault-tolerant algorithm of a system entails making assumptions about the behaviour of faulty components of the system. A given faulty component can behave in many different ways, some of which will be easier to tolerate than others; furthermore, certain patterns of faulty behaviour are likely to be more probable than others.

Suppose we can classify faulty behaviour of a component starting from those that are relatively restricted breaches of the specification (caused by simple faults) to those that are increasingly more general breaches of the specification (caused by complex faults). Then we can design a *family of fault-tolerant algorithms* – from simple ones tolerating simple faults to increasingly more complex ones tolerating larger classes of more general faults. Given such a family of algorithms, one can select a particular one depending upon the stated reliability requirements – choosing an algorithm tolerating larger classes of faults (or in the extreme, all types of faults) for a system requiring a very high degree of reliability. In this section, such a fault classification is presented. The treatment presented here is based on that by Ezhilchelvan & Shrivastava (1986) where more details can be found.

Following Kopetz (1985, pp. 91–101), the response of a component for a given input will be said to be correct if the output value is not only as expected, but also produced on time. Formally, the correct response of a component is defined as follows.

Correct response of a component: Let a component receive at time t_i an input requiring a non-null response from the component and as a result produce an output value v_j at time t_j. For that input, the response v_j at time t_j is correct iff:
(i) the value is correct: $v_j = w_j$, where w_j is the expected value consistent with the specification; and,
(ii) the response is correct: $t_j = t_i + t_d + \delta_t$, where t_d is the minimum delay time of the component, and δ_t is the unpredictable delay such that $0 \leq \delta t \leq t_{\max}$, and $t_{\max}$ is the maximum unpredictable delay time of the component.

The values t_d and $t_{\max}$ are constants for a given component. First of all, we note that the response of a component cannot be instantaneous to a given input but must experience a finite minimum amount of delay which is specified by the parameter t_d. Secondly, it is usual in engineering specifications to indicate a time interval during which a response is required; according to our definition, this interval is from $t_i + t_d$ to $t_i + t_d + t_{\max}$.

A correctly functioning component does not arbitrarily produce responses. In particular, when there is no input (null input) or when no response is expected for an input, there is naturally no output value produced (output is null). The values t_d and $t_{\max}$ are meaningful only when non-null output values are produced.

If $v_j \neq w_j$, then the output value will be termed *incorrect*; similarly, if $t_j < t_i + t_d$ (output produced too early) or $t_j > t_i + t_d + t_{\max}$ (output produced too late), then the response time will also be termed *incorrect*.

Given the above definitions of correct and incorrect responses, there can be *at most three* possible ways by which a response can deviate from that specified. This leads to the following three types of faults.

(*i*) *Timing fault*: A fault that causes a component to produce the expected value for a given input either too early or too late will be termed a *timing fault* and the corresponding failure a *timing failure.* Using our notation:

$$\text{(i) } v_j = w_j, \text{ and (ii) either } t_j < t_i + t_d \text{ or } t_j > t_i + t_d + t_{\max}.$$

(ii) *Value fault*: A fault that causes a component to respond, for a given input, within the specified time interval, but with a wrong value will be termed a *value fault* and the corresponding failure a *value failure*:

$$\text{(i) } v_j \neq w_j, \text{ and (ii) } t_j = t_i + t_d + \delta_t.$$

(iii) *Commission fault*: A *fault of commission* is responsible for a *commission failure* with the following property:

$$v_j \neq w_j, \quad \text{and/or} \quad t_j \neq t_i + t_d + \delta_t.$$

A commission failure is *any* violation from the specified behaviour. In particular, it includes the possibility of a component producing a response when no input was supplied.

(iv) *Omission fault*: Many fault-tolerant algorithms are designed under a particularly simple failure mode assumption, which is that a component can fail only by producing no response. A fault which causes a component, for a given input requiring a non-null response, not to produce any response will be termed an *omission fault* and the corresponding failure an *omission failure.*

We could regard 'not producing a response' as equivalent to 'producing a null value on time', thereby treating an omission fault as a special case of a value fault. We can also treat an omission fault as a special case of a timing fault by regarding 'not producing a response' as equivalent to 'producing a correct value at infinite time'.

Fault/failure lattice: A commission fault (failure) subsumes all the other three types of faults (failures). The relationships among these four types of faults (failures) can be expressed by the following fault (failure) lattice (figure 4), where an arrow from A to B, indicates that fault (failure) type A is a special case of fault (failure) type B. (The relation '->' is transitive.) An important observation can now be made which is that a fault-tolerant algorithm designed to tolerate m, $m > 0$, timing failures (value failures) can also tolerate m omission failures and further that an algorithm designed to tolerate m commission failures can tolerate m failures of any type. The top of the lattice represents the simplest and the bottom, the most general fault (failure).

Examples: We will next give some examples of various types of failures. A self-checking component (e.g. a processor) that stops functioning as soon as an error is detected within itself can be regarded as suffering from omission failures. On the other hand, a self-checking component, which upon detecting an error, responds within time by producing a 'fail signal' can be said to fail due to a value fault. If the signal is produced too late or too early, then the failure would be classed as a commission failure. A software module that produces correct output values but too late (perhaps because the processor executing the program was overloaded) will fail in a timing manner. Similarly, late delivery of an uncorrupted message will be termed a timing failure, while a late delivery of a corrupted message will be a failure of commission. Delivery of a corrupted message within time will be a value failure. A component that produces values arbitrarily will have a commission fault.

The above classification is based on the behaviour of a component with respect to an individual response. Each type of fault (and failure) can be further subclassified when a *sequence* of responses is considered. If a particular faulty behaviour persists for a 'sufficiently lengthy' response sequence, then that failure type can be classified as *permanent* (as against *transient*). The ideas presented here are discussed at length by Ezhilchelvan & Shrivastava (1986) where a family of agreement protocols has also been developed.

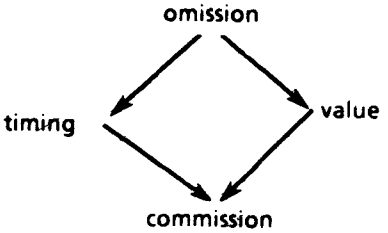

Figure 4. Fault/failure lattice.

7. Concluding remarks

We began by examining the nature of systems and their faults and developed basic concepts of fault tolerance. These concepts were utilized in a methodology for the development of robust software modules – the building blocks of any software system. The concepts presented here can be applied to the design of a wide variety of computing systems. We present two examples from distributed systems composed of a number of nodes (computer systems) connected by a communications system (e.g. a local area network).

(1) *Robust distributed programs:* Let us consider the reliability aspects of *distributed programs*: programs that have been composed out of modules residing on different nodes of a distributed system. We will consider a specific class of applications such as banking and office information systems, where maintaining the integrity of stored data is of considerable importance. Imagine that the nodes of the system provide various services which can be invoked from any node. A typical distributed program might be thought of as composed out of a 'root' program (running at a user's node) that contains service calls to some remote services and routines at nodes that provide the services. The execution of such a program will involve a group of cooperating processes distributed over the system. When a program running at some node makes a legitimate service call to some other node, there can be many reasons why that service might not be available; for example, the communication link between the nodes might be faulty or the server node may have 'crashed' and so on. For these, and many other reasons, it is quite possible for the computation of a distributed program to arrive at a state from which further meaningful progress is not possible. Under such circumstances it is preferable that the computation be terminated without producing any results (side effects). Various reliability mechanisms are necessary for supporting such 'cleanly' terminating programs that maintain the integrity of stored data. In addition to the integrity requirement, we also require the property of *durability of results*: once results have been produced by a terminated program, the results should survive system failures with a high probability of success. Finally, it is required that the stored data be made available despite system failures.

It is well-known that the above reliability requirements can be tackled within the framework of *atomic actions* (*atomic transactions*) (Lampson & Sturgis 1981; Gray 1986). The methodology presented here provides an ideal set of structuring concepts (Randell 1985). System design will require making fault models of components such as communication media, nodes and storage media. Specific fault-tolerant algorithms are then constructed for reliable interprocess communications (e.g. remote procedure calls, Lin & Gannon 1985; Panzieri & Shrivastava 1987), reliable storage, maintenance of replicated data, concurrency control and so forth. The most convenient way of introducing design diversity in such a system is at the level of atomic actions; for example, by executing each primary or alternative of recovery blocks as an atomic action.

There is a large body of literature on the topic of atomic actions in distributed systems. The interested reader may find the tutorial presented in Shrivastava (1985a, pp. 102–121) a useful starting point.

(ii) *Replicated distributed processing*: Many real-time systems require a very high degree of reliability, for which utilization of modular redundancy in the form of replication of processing modules with majority voting provides a very attractive possibility. An added advantage of replicated processing for real-time systems is that the time critical nature of processing often means that masking of failures by majority voting is the most appropriate fault treatment strategy. We will consider our system to be composed of a number of nodes fully connected by means of redundant communication channels. A node will represent a functional processing module, constructed as a number of processors and voters in a classical NMR (N-modular redundant) configuration. Fault-tolerant scheduling algorithms are required for properly executing real-time tasks in a replicated manner (Shrivastava 1987). In particular it is necessary to ensure that all the non-faulty processors of a node execute incoming tasks in an identical order. An interesting aspect of the work reported by Shrivastava (1987) is that the exception handling framework reported here can be applied to the development of voting algorithms for detecting certain types of component failures (see Mancini & Shrivastava 1986 for more details). Thus, voters enhanced in this manner can be exploited for passing on component failure information to the reconfiguration sub-system.

The replicated distributed processing architecture, briefly mentioned here, provides a suitable framework for executing N-version programs. The system developed at UCLA (Avizienis 1985) has many similarities to the architecture described here.

Many of the ideas presented in this paper have been developed over the years by a well-established research group at the author's institution. Some of the work of this group is available in book form (Shrivastava 1985b) and may be of interest to readers wishing to delve further into the exciting subject of fault-tolerant computing.

The work reported here has been supported in part by research grants from the Science and Engineering Research Council and the Ministry of Defence. Comments from Tom Anderson on a previous version of the paper are gratefully acknowledged.

References

Anderson T, Barrett P A, Halliwell D N, Moulding M R 1985 *IEEE Trans. Software Eng.* SE-11: 1502–1510

Anderson T, Lee P A 1981 *Fault tolerance: Principles and practice* (Englewood Cliffs, NJ: Prentice Hall)

Anderson T, Lee P A 1982 *Proc. of 12th Fault-tolerant Computing Symposium, Santa Monica* (Silver Spring, MD: IEEE Comput. Soc. Press) pp. 29–33

Avizienis A 1976 *IEEE Trans. Comput.* C-25: 1304–1312

Avizienis A 1985 *IEEE Trans. Software Eng.* SE-11: 1491–1501

Cristian F 1982 *IEEE Trans. Comput.* C-31: 531–540

Ezhilchelvan P, Shrivastava S K 1986 *Proc. 5th Symp. on reliability in distributed software and database systems, Los Angeles* (Silver Spring, MD: IEEE Comput. Soc. Press) pp. 215–222

Gray J N 1986 *IEEE Trans. Software Eng.* SE-12: 684–689

Horning J J, Lauer H C, Melliar-Smith P M, Randell B 1974 *Lect. Notes Comput. Sci.* 16: 177–193

Knight J C, Leveson N G 1986 *Proc. of 16th Int. Symp. on Fault Tolerant Computing, Vienna* (Silver Spring, MD: IEEE Comput. Soc. Press) pp. 165–170
Kopetz H 1985 in *Resilient computing systems* (London: Collins)
Lampson B, Sturgis H 1981 *Lect. Notes Comput. Sci.* 105: 246–265
Lee P A, Anderson T 1985 in *Resilient computing systems* (London: Collins)
Lin K J, Gannon J D 1985 *IEEE Trans. Software Eng.* SE-11: 1126–1135
Liskov H, Snyder A 1979 *IEEE Trans. Software Eng.* SE-5: 546–558
Luckham D C, Polak W 1980 *ACM Trans. Program. Lang. Syst.* 2: 225–233
Mancini L, Shrivastava S K 1986 *Proc. of 16th Int. Symp. on fault-tolerant Computing, Vienna* (Silver Spring, MD: IEEE Comput. Soc. Press) pp. 384–389
Martin D J 1982 *AFARD Symp. on software for Avionics, the Hague*, 36: 1
Panzieri F, Shrivastava S K 1987 *IEEE Trans. Software Eng.* (to appear)
Randell B 1975 *IEEE Trans. Software Eng.* SE-1: 220–232
Randell B 1985 in *Reliable computer systems* (ed.) S K Shrivastava (Berlin: Springer-Verlag) chap. 7
Shrivastava S K 1985a in *Resilient computing systems* (London: Collins)
Shrivastava S K (ed.) 1985b *Reliable computer systems. Texts and monographs in computer science* (Berlin: Springer-Verlag)
Shrivastava S K 1987 *Lecture Notes Comput. Sci.* 248: 325–337