On Failures and Faults [1]

Brian Randell

School of Computing Science
University of Newcastle upon Tyne

Abstract: Real computer-based systems fail, and hence are often far less dependable than their owners and users need and desire. Individuals, organisations and indeed the world at large are becoming more dependent on such systems, so there has been much work on trying to gain increased understanding of the many and varied types of faults that need to be prevented or tolerated in order to reduce the probability and severity of system failures. In this paper I analyze the concept of system faults and failures, and discuss the assumptions that are often made by computing system designers regarding faults, and a number of continuing research issues related to fault tolerance.
Keywords: Dependability, formal concepts, fault assumptions.

1 On Fault-Tolerant Computing

The direct origins of modern fault-tolerant computing lie in John von Neumann's influential work in the early 1950s on "Probabilistic Logic and the Synthesis of Reliable Organisms from Unreliable Components" [22]. In the 1950s and 1960s much work was done on hardware fault tolerance, from the (widespread) use of error detecting and correcting codes, to the more exotic realms of replicated processors, automatic reconfiguration, etc., used in highly demanding environments, e.g. in aerospace. However, in the software world, the notion of dependability was still equated to that of correctness – indeed, of perfecting the software development process.

In 1968 I participated in the first NATO Software Engineering Conference at Garmisch in Bavaria [17]. The participants constituted a broad international cross-section of industry, government and academia. What was special and novel about this conference was the readiness of these participants to face up to the at times very serious faults in the whole process by means of which software was then specified, designed, implemented and deployed. The impact of this conference, particularly on many of the attendees, was therefore immense. For example, both Edsger Dijkstra and I later went on record as to how the discussions at this conference on the "software crisis" had strongly influenced our thinking and our subsequent research activities. In his case the discussions prompted his study of formal approaches to producing high quality, indeed formally validated, programs. In my case they led me to the belief that large software systems would essentially always contain residual design faults and, following my move to Newcastle soon after the Garmisch Conference, to the then novel and controversial idea that it was worth trying to find means by which such

[1] Much of this paper is based closely on some of the material in my BCS/IEE 1999 Turing Memorial Lecture [19]

systems could nevertheless be made adequately reliable. (As I've remarked before, our respective choices of research problems suitably reflect our relative skills as programmers.)

A detailed study that I and my colleagues were commissioned to make in 1970 of a number of large on-line computer systems confirmed that software faults were a major cause of undependability in these systems, and more importantly, resulted in our finding that:

(i) a significant fraction of the code in these systems was aimed at detecting and recovering from errors caused by hardware and operational faults,
(ii) this code was ad hoc and limited in its capability, e.g. concerning the possibility of concurrent faults, or of further errors being detected while error recovery was already being attempted, yet
(iii) nevertheless, somewhat fortuitously, these error recovery facilities did in fact help to provide a useful measure of software fault tolerance.

This study marked the start of a still-continuing, and indeed now greatly-expanded, programme of research at Newcastle on system dependability, and in particular fault tolerance (for various types of fault), which has been funded by a succession of research grants from UK and European government sources, and from industry. The subject of fault tolerance continues, even thirty years later, to fascinate me – my aim in this talk is to try to explain why.

2 On Dependability Concepts

The concept of a "fault" is surprisingly subtle – or, as I would prefer to put it, "gloriously recursive". Indeed, clarifying the concepts related to dependability is difficult – and hence vitally important – when one is talking about systems in which

(i) there is potential confusion regarding the placement and nature of system boundaries
(ii) the very complexity of the systems (and their specification, if they have one) is a major problem,
(iii) judgements as to possible causes or consequences of failure can be very subtle, and
(iv) there are (fallible) provisions for preventing faults from causing failures.

From early on in our work at Newcastle on software fault tolerance we realised the inadequacy, with regard to residual design faults, of the definitions of terms such as fault and error used at that time by hardware designers. The problem was that they took as the basis of their definitions a set of terms for a few well-known forms of fault, such as "stuck-at-zero" faults, "bridging" faults, etc. This approach did not seem at all appropriate for thinking about residual design faults, given the huge variety, and the lack of any useful classification, of such faults. In fact, we eventually realised that we could achieve the generality we needed by starting not from faults, but from the concept of a system "failure" [20].

The ensuing generality of our definitions led us to start using the term "reliability" in a much broader sense than was then common, since a system might fail in all sorts of

ways – it might deliver the wrong results, work too slowly, fail to protect confidential information, lead to someone's death, or whatever. Our over-generalisation of the term "reliability" was not well received, and it was a French colleague, Jean-Claude Laprie of LAAS-CNRS, who came to our linguistic rescue by proposing the use of the term "dependability" [13] for the concept underlying our broadened definition. The term dependability thus can be seen as including, as special cases, such properties as availability, reliability, safety, confidentiality, integrity, etc. These are illustrated in Figure 1, taken from [14], as being attributes of dependability.

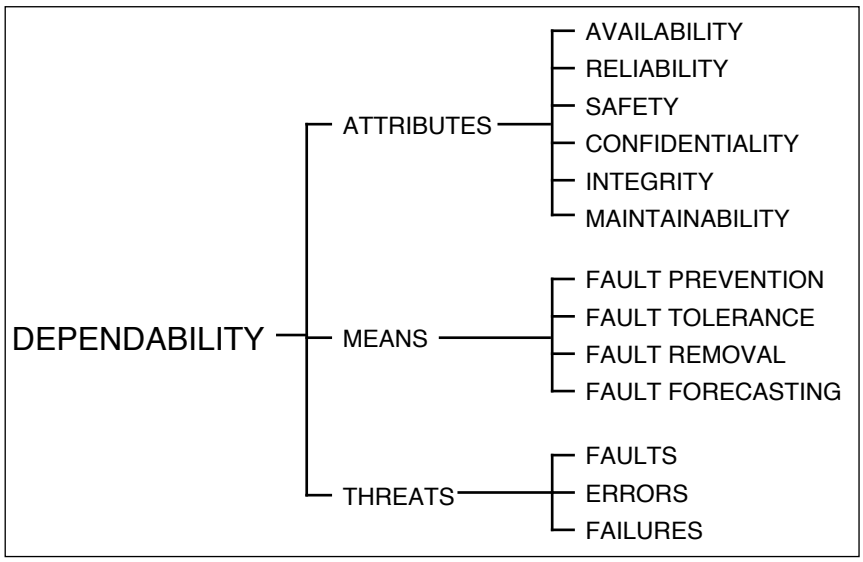


Fig. 1. The dependability tree

Quoting from the latest published version of the dependability definitions [3][2].

> "A system **failure** occurs when the delivered service deviates from fulfilling the system **function**, the latter being what the system is *aimed at*.."

The phrase "what the system is aimed at" is a means of avoiding reference to a system "specification" – since it is not unusual for a system's lack of dependability to be due to inadequacies in its documented specification. (I return to the issue of inadequate specifications below.)

Systems of interest will possess an internal state:

> "An **error** is that part of the system state which is *liable to lead to subsequent failure*: an error affecting the service is an indication that a

[2] A revised edition of the definitions in [13] and [14]

failure occurs or has occurred. The *adjudged or hypothesised cause* of an error is a **fault**."

Note that an error may be judged to have multiple causes, and does not necessarily lead to a failure – for example error recovery might be attempted successfully and failure averted.

"A failure occurs when an error 'passes through' the system-user interface and affects the service delivered by the system – a system of course being composed of components which are themselves systems. Thus the manifestation of failures, faults and errors follows a "fundamental chain":

. . . → failure → fault → error → failure → fault → . . ."

One example of this fundamental chain is as follows:

"the result of a programmer's error is a (dormant) fault in the written software (faulty instruction(s) or data); upon activation (invoking the component where the fault resides and triggering the faulty instruction, instruction sequence or data by an appropriate input pattern) the fault becomes active and produces an error; if and when the erroneous data affect the delivered service (in value and/or in timing of their delivery), a failure occurs." [12].

The recognition of the importance of this chain – which takes the form of

. . . → event → cause → state → event → cause → . . .

led to a great increase in our ability to understand, and to design means of ameliorating, all sorts of complex manifestations of undependability. This chain can go from one system to:

(i) some enclosing system of which it is a component,
(ii) another essentially separate system with which it is deployed, or from
(iii) a further system(s) that it creates

Let me illustrate some of these possibilities by a further example:

A fighter plane crashed killing the pilot – it turned out that it had for a period before this failure (i.e. the crash) been calculating its position erroneously, and that this was due to it having been fitted with the wrong (albeit correctly functioning) inertial navigation subsystem. One could describe this as the fault. In fact, this fault had arisen as the result of a failure of the (largely human) system responsible for maintaining the plane. But this failure (i.e. the act of installing the wrong inertial navigation subsystem) could in part be blamed on a much earlier failure of the system that had specified and designed the whole plane maintenance system; it had created a situation in which two functionally distinct inertial navigation subsystems had identical mechanical interfaces, and catalogue numbers that differed by only one in the least significant digit! This was surely a situation that was a positive invitation

> to disaster. In fact it was eventually determined that the erroneous catalogue number had been generated as a result of a hitherto un-noticed failure by the computerised inventory control system. This failure was due to the fact that an overflow had occurred from a quantity field into the catalogue number field, in a COBOL program that contained no checks against overflow. So, here we have a whole set of different systems, and a complicated chain, in which failures in one system constituted faults in other systems that created erroneous states which were not corrected but instead led to further failures.

This did actually happen. The good news is that the overall inventory control process, which was part-manual, part-automated, was in other respects so well designed and managed that it was possible to determine when the overflow had occurred and which other planes also had been fitted with the wrong inertial navigation subsystem – so other impending fatalities were averted. In fact, some of the most important sources of the whole subject of database transactions and integrity controls derive in large part from this work, which was carried out by C.T. Davies, first for the U.S. Air Force, and later at IBM, and led to the creation of the very influential "spheres of control" concept [6].

The wording that has been in use for some time as a definition of computer system dependability *per se* is:

> "**Dependability** is defined as that property of a computer system such that *reliance can justifiably be placed on the service* it delivers. (The service delivered by a system is its behaviour *as it is perceptible* by its user(s); a user is another system (human or physical) which *interacts* with the former.)"

I now feel it possible, and worthwhile, to improve on the definition of "failure" in order to make explicit the judgement that is involved, and to use this more directly in the definition of dependability. First the alternative definition of failure:

> A given system, operating in some particular environment (a wider system), may fail in the sense that some other system makes, or could in principle have made, a *judgement* that the activity or inactivity of the given system constitutes **failure**.

The second system, the judgemental system, may be an automated system, a human being, the relevant judicial authority or whatever. (It may or may not have a documented system specification to guide it.) Different judgemental systems might, of course, come to different decisions regarding the given system. Moreover, such a judgemental system might itself fail – in the eyes of some other judgemental system – a possibility that is well understood by the legal system, with its hierarchy of courts. So, we can have a (recursive) notion of "failure" which is defined merely in terms of what are taken as the fundamental, dictionary-defined, concepts of "system" and "judgement", and which clearly is a relative rather than an absolute notion. So then is the concept of dependability:

> The concept of **dependability** can be simply defined as "the quality or characteristic of being dependable", where the adjective "dependable" is

attributed to a system whose failures are judged sufficiently rare or insignificant.

It should be noted that these definitions, and the four basic means of obtaining and establishing high dependability, namely fault prevention, fault tolerance, fault removal and fault forecasting, are as applicable to human and industrial systems, as they are to computer systems. In particular they are applicable to the part-manual part-automated systems, i.e. "computer-based systems", including those that are used to design and implement computer systems. This generality, and the explicit role given to judgement, are important given the subtleties that are sometimes involved in identifying the exact boundaries of the various systems of concern, of resolving disagreements regarding the acceptability of a system's specified and/or actual behaviour, and of determining how blame should be apportioned after a system fails.

2.1 Concept Formalisation

In fact, since a first version of this discussion was published [19], my colleague Cliff Jones has, to my great delight, taken up very seriously the problem of providing a formalisation of these basic dependability concepts. I have long regretted the lack of such a formalisation, since I recall how much benefit I, at least, obtained from my collaboration many years ago with Jim Horning on a paper for Computing Surveys on process structuring concepts [10]. (This paper interlaced informal and formal definitions of a large number of concepts related to processes, process combination and process abstraction, using a few rather basic mathematical concepts, such as sets, sequences, relations, and functions.) Subsequently, much of my research, in particular my recent involvement in work on Co-ordinated Atomic Actions (see below), has been greatly helped by the efforts of my more formal colleagues,

The formalisation of the basic dependability concepts in [11] is introduced in the following terms:

> "The idea here is to offer definitions of [the terms *fault, error, failure*] with respect to a particular notion of what constitutes a system. . . . The intention here is not to offer formalism for its own sake. In fact the details of the particular notation, etc., are unimportant. The hope is that understanding can be increased by employing a firm foundation. [. . .] some interesting relationships between systems are explored. The propagation of *fault, error, failure* chains where one system is *built on* another system are well-understood. Many of the failure propagation systems of interest in socio-technical systems arise when one system is *created by* another. Lastly, the idea of one system being *deployed with* another is considered."

The systems that are dealt with are thus both technical and socio-technical (e.g. computer-based) systems, and include systems (e.g. "real-time" control systems) that are linked to processes that evolve autonomously. The formalism Cliff uses is in fact VDM, though he points out that Z and B would be equally appropriate.

One of the main divergences between Cliff and myself concerns the role played by a "specification". Quoting again from his paper:

> "[My view] is that the judgement that a system fails can only be made against a specification. What if the "specification is wrong"? Presumably, this means that the specification is in some sense inappropriate; the specification might be precise, but it can be seen to result in faults and failures in a bigger system. For example, a specification might state that a developer can assume that the user might respond in one micro-second but failing to so do can result in fatal consequences. The developer writes a program which "times out" after one micro-second and an accident occurs. It surely is not right to say that the software system (which meets its specification) is failing. Nor of course is it reasonable to blame "operator error" with such an unreasonable assumption. The only reasonable conclusion is that it is an earlier system which exhibited erroneous behaviour: the act of producing the silly specification is the failure that caused a fault in the combined system of software and operator. The judgement that a specification is "silly" must of course be made by another (external) system. A similar argument can be made for missing specifications: an engineering process requires a reference point."

I remain to be convinced on this point. I have seen systems whose facilities and interfaces are so intuitive and well-chosen that users can immediately understand how to operate them, and have to turn to the manual, assuming there is one, only in extremis. Moreover, they can with little difficulty recognize any such failures as do occur as being failures by their inconsistency with respect to other aspects of the perceived behaviour of the system. On the other hand, in my experience specifications of large computer or computer-based systems are rarely complete, in the sense that one can guarantee that any implementation that satisfies the specification will be regarded as fully satisfactory by the people who are in a position to judge the system. Rather, specifications should act as constraints, possibly extremely detailed constraints, that enable one to state various (pre-conceived) ways in which a system would be regarded as inadequate.

The notion of the superiority of such "negative" specifications is in fact one of the important ideas in Alexander's very influential book "Notes on the Synthesis of Form" [1]:

> ". . . every design problem begins with an effort to achieve fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem. In other words, when we speak of design, the real object of discussion is not the form alone, but the ensemble comprising the form and its context. Good fit is a desired property of this ensemble which relates to some particular division of the ensemble into form and context. . . . It seems as though in practice the concept of good fit, describing only the absence of such failures and hence leaving us nothing concrete to refer to in explanation, can only be explained indirectly; it is, in practice, the disjunction of all possible misfits.
>
> In the case of a design problem which is truly problematical, we . . . have no intrinsic way of reducing the potentially infinite set of requirements to

> finite terms. Yet for practical reasons we do need some way of picking a finite set from the infinite set of possible ones. In the case of requirements, no sensible way of picking this finite set presents itself. From a purely descriptive standpoint we have no way of knowing which of the infinitely many relations between form and context to include, and which ones to leave out. But if we think of the requirements from a negative point of view, as misfits, there is a simple way of picking a finite set. This is because it is through misfit that the problem originally brings itself to our attention. We take just those relations between form and context which obtrude most strongly, which demand attention most clearly, which seem most likely to go wrong. We cannot do better than this. . . .
>
> In the case of a real design problem, even our conviction that there is such a thing as fit to be achieved is curiously flimsy and insubstantial. We are searching for some kind of harmony between two intangibles; a form which we have not yet designed, and a context we cannot properly describe. The only reason we have for thinking that there must be some kind of fit to be achieved between them is that we can detect incongruities, or negative instances of it.

Such detection of course involves judgement – both while the design is being created, and for any resulting real (i.e. fallible) system, while this system is deployed. Hence my view is that system specifications are, at least conceptually, just a valuable adjunct to an authoritative judgement system. (An analogy I would make is to the notion of a contract – normally one would expect this to be written and signed – but in some situations and environments a handshake will be equally acceptable and indeed binding in law.) But in practice, as stated in [11], "it is difficult to see how an engineering process can be used to create a system where there is no initial notion of specifying the required properties of the to-be-created artefact". One can hardly rely entirely on some scheme of having all the designers continuously interrogating a judge throughout the design process! However, with regard to dependability concept definitions, I still prefer to avoid involving reference to a specification.

For example, Cliff's specification-oriented approach led him to the following definition of the notion of an error: "An error is present when the state of a system is inconsistent with that required by the specification". In contrast, the definition I am used to is: "An error is that part of the system state that may cause a subsequent failure: a failure occurs when an error reaches the service interface and alters the service." No doubt this definition could be clarified through formalisation, but I feel this should be done using judgement rather than specification as a starting point.

This is because, in my view, there will often be considerable subjective judgement involved in identifying errors, particularly errors due to design faults in complex software. Once a fault has been activated, all subsequent state transitions up to the occurrence of a failure are to be regarded as errors. However, identifying the fault occurrence involves deciding which instruction, or set of instructions, is incorrect or missing, i.e. how the faulty program compares with "the" correct program. But there could be several, equally sensible ways of correcting a program, and hence identifications of the location of "the" fault in the code, the moment when it was

activated, and its subsequent errors – and it is not evident that a specification could or should be so detailed as to provide means of adjudicating between these different equally-correct programs.

A rather different situation, also illustrating the distinction between these two definitions of error, is that of a system failure which comes to be regarded as due to some earlier failure of another system, either one it is deployed with, or one it was created by. The state of the given system prior to its failure will be erroneous by the definition that I favour. This is the case even if is decided that the given system was correctly processing the faulty inputs it received, or is correctly interpreting the faulty design that it incorporates, (so that its state is in all probability consistent with its specification, should one have been documented).

Nevertheless, despite such differences of approach, I very much welcome Cliff's contribution to the development and fomalisation of dependability concepts, and hope to contribute to extending it to deal with further basic dependability concepts. One such is that of "dependence" – which, perhaps surprisingly, has not been dealt with in the standard accounts of dependability concepts and terminology, so is the main point I deal with next.

2.2 Dependence, Trust and Confidence

It is commonplace to say that the dependability of a system should suffice for the dependence being placed on that system. What we term the "dependence of system A on system B" is thus a measure of the impact of B's undependability on A's dependability. Clearly, this can vary from total dependence (any failure of B will cause A to fail) to complete independence (B cannot cause A to fail). In other words, dependence can be defined as a measure of the difference between the dependability that A would have, were B to be totally dependable, with that which A has in the presence of the actual (presumably less than fully dependable) B.

If there is reason to believe that B's dependability is insufficient for A's required dependability, the former must be enhanced, and/or A's dependence reduced, and/or additional means of fault tolerance introduced "between" A and B, e.g. in the form of a "wrapper".

The concept of dependence leads on to those of "trust" and "confidence", two terms that are much in current vogue in the EU IST Programme. (Another word in common use in some circles is "trustworthiness" – which I in fact regard as being synonomous with "dependability.") In my view, trust can very conveniently be defined as "accepted dependence" – i.e. the dependence (say of A on B) allied to a judgement that this level of dependence is acceptable. Such a judgement (made by or on behalf of A) about B is possibly explicit, and even laid down in a contract between A and B, but might be only implicit, even unthinking. Indeed it might even be unwilling – in that A has no alternative option but to put its trust in B.

Thus to the extent that A trusts B, it need not assume responsibility for, i.e. provide means of tolerating, B's failures. (The question of whether it is capable of doing this is another matter.). Indeed, turning things around, the extent to which A fails to provide means of tolerating B's faults is a measure of A's (perhaps unthinking or unwilling) trust in B.

Thus the notion of trust is applicable to technical or socio-technical systems, as well as to humans. A distinction between trust and confidence is that the former leads to the

act of becoming dependent, the latter is inapplicable to technical systems, since it concerns how some human, or group of humans, might feel about this act. A system which provides evidence which can be used to attempt to justify A's trust in B, i.e. to provide confidence regarding A's dependence on B, can itself of course fail. One type of failure of such a confidence-building system (which might be system A itself), produces an underestimate of A's dependence on B, which could lead to a decision to avoid using B, even though B is adequately dependable. What is normally a more serious type of failure of a confidence-building system puts A at unacceptable risk of failing due to a failure of B, i.e. of a "trusted" system turning out to be "untrustworthy".

2.3 Concepts and Terminology

This continued interest that I and a number of people involved in dependability research take in concepts and definitions perhaps seems rather pedantic, though I believe it is fully justified. One reason of course is the subtleties involved, and the need to clarify them. Another is the fact that a number of what are essentially dependability concepts are being re-invented (sometimes rather incompetently), or at least re-named, in numerous research communities, which variously categorise their area of interest as safety, survivability, trustworthiness, security, critical infrastructure protection, information survivability, or whatever.

The issue of whether the different research communities use a common set of terms is much less important than their failure to recognise that they are concerning themselves with (different facets of) the same concept. One consequence is that they are not getting as much advantage from each other's insights and advances as they might. However, regardless of the terminology employed, I believe it is very important to have, and to use, some term for the general concept, i.e. that which is associated with a *fully general notion of failure* as opposed to one which is restricted in some way to particular types, causes or consequences of failure. (I also believe it is essential to have separate terms for the three essentially different concepts named here "*fault*", "*error*" and "*failure*" – since otherwise one cannot deal properly with the complexities (and realities) of failure-prone components, being assembled together in possibly incorrect ways, so resulting in failure-prone systems.) Only when this is done will, I believe, the researchers take an adequately general approach to the problems that they are attempting to tackle. And if I manage to put over only one point in this lecture – this is the one I hope it will be.

In fact, time and time again it seems to me that muddled thinking about dependability-related notions has been a barrier to progress – most recently I have been alerted to this in the work of the "intrusion detection" research community. This community concerns itself with a major aspect of the problem of protecting computer networks and networked computers from hackers. As some of the researchers involved have admitted, the community has got itself into very confused and confusing debates as it tries to expand its horizons beyond the problem of merely detecting the fact that some hacker is, or has recently been, intruding into a system. I will return to this topic later.

3 On Fault Classification

As I have indicated, the faults that might affect a computer-based system are many and varied. A detailed classification is provided by Laprie [14], the first part of which is summarised in Figure 2.

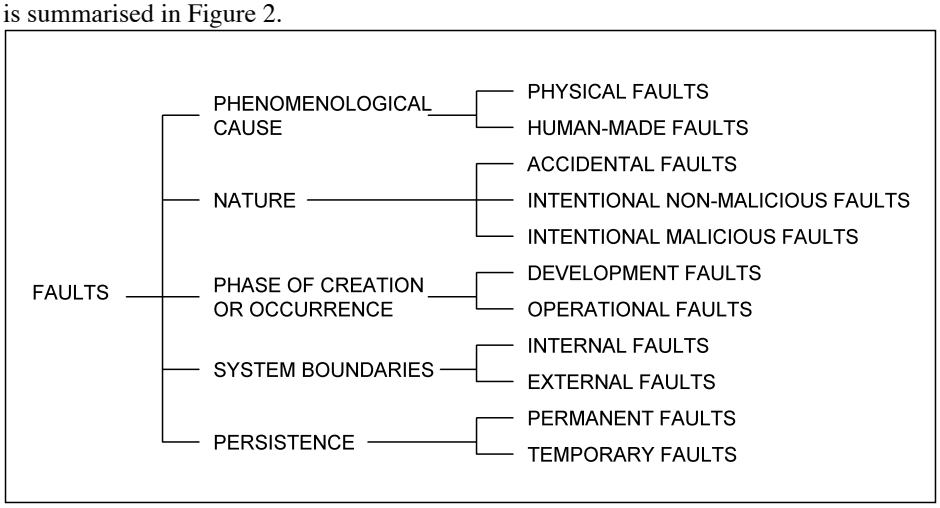


Fig. 2. Fault classification

The actual application of this classification itself involves judgement. For example, is a hardware component that occasionally fails as a result of electronic interference suffering temporary operational physical faults, or should one regard it as having been provided with inadequate shielding, i.e. regard the situation as being due to a permanent human-made development fault? Nevertheless, such classifications can provide useful guidance, in particular with regard to planning means of fault prevention, tolerance, diagnosis, and removal, based on the designer's assumptions about likely faults. Actual automated fault tolerance strategies can however make only limited use of such classifications, depending on the extent to which faults can be quickly and accurately recognised as belonging to a class for which some specialised fault tolerance measure is particularly appropriate. This is often not feasible – for example, it is often difficult to be sure whether the immediate cause of an error is an operational hardware fault or a residual software design fault, so if both possibilities are to be allowed for a very general fault tolerance strategy must be employed.

Logical, or qualitative, classification is however just a starting point. All work on system dependability is, or at any rate should be, guided by considerations of relative probabilities. There is little point, for example, in providing a parity bit with each byte being transmitted over a network if the most common form of fault causes long bursts of errors. Similarly, there is little point employing expensive and time-consuming code verification tools if in a particular application domain virtually all the significant failures arise from inadequate system specifications. The success with which one can design a system that will achieve some required level fault of tolerance therefore

depends on the quality of the statistics that are available, or of the statistical assumptions made by the designers, concerning the faults that will occur.

In principle, and often in practice, one can have relatively accurate statistics concerning operational hardware faults, detailed enough to provide very useful guidance as to what specific fault tolerance measures are needed where, and what ones are not worth their cost. When it comes to residual design faults, such statistics as are available are too imprecise to be of much use, so fault tolerance provisions have to be very general. Thus one of the motivations behind the original recovery block scheme [9] was to provide means of error recovery that would work (almost) no matter what fault existed where in the suspect program.

```
ensure   acceptance  test
by       primary alternate
else by alternate 2
              .
              .
else by alternate n
else     error
```

Fig. 3. The Recovery Block Structure

With the program structuring scheme that we developed this was the case so long as the underlying recovery and control mechanism was not corrupted. However, the degree to which the error recovery mechanism could be used to provide successful fault tolerance, i.e., enable the program to continue and produce satisfactory results, of course depended on the adequacy of the programmer-supplied error detection measures (such as acceptance tests) and last-ditch alternate blocks. Our first demonstration recovery block system involved a fault-tolerant application program containing a complete acceptance test and final alternate block, running on a simulated machine which completely confined programs within their allotted resources. We then provided visitors with means of making arbitrary changes to the code of any or all of the alternate blocks (other than the final one) in the running application program – the challenge to them being to find some means of preventing the program from producing correct results. Within a short period of time this demonstration system had been honed to the point where no visitors were able to subvert it. This demonstration was a very compelling one.

The demonstration in fact indicated that when the concern is with the possibility of malicious faults the only sensible thing is to assume that the situation is statistically as bad as could be imagined – that faults occur at locations, in circumstances, and with a frequency, that are essentially “pessimal” from a designer’s viewpoint. (The term “pessimal” is in fact not in the dictionary, though its meaning, and the need for such a word, are I claim both self-evident.) I will return to the problems of tolerating malicious faults later.

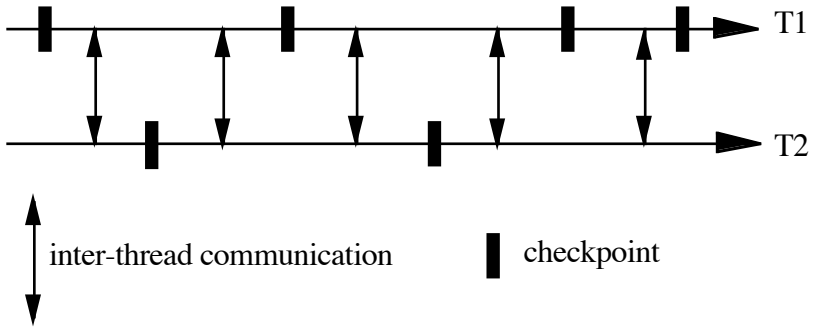
4 On Fault Assumptions

The problems of preventing faults in systems from leading to system failures vary greatly in difficulty depending on the (it is hoped justified) assumptions that the designers make about the nature as well as the frequency of the faults, and the effectiveness of the fault tolerance mechanisms that are employed. For example, one might choose to assume that operational hardware faults can be cost-effectively masked (i.e. hidden) by the use of hardware replication and voting, and that any residual software design faults can be adequately masked by the use of design diversity, i.e. using N-version programming. In such circumstances error recovery is not needed. In many realistic situations, however, if the likelihood of a failure is to be kept within acceptable bounds, error recovery facilities will have to be provided, in addition to whatever fault prevention and fault masking techniques are used.

In a decentralised system, i.e. one whose activity can be usefully modelled by a set of partly independent threads of control, the problems of error recovery will vary greatly depending on what design assumptions can be justified. For example, if the designer concerns him/herself simply with a distributed database system and disallows (i.e. ignores) the possibility of undetected invalid inputs or outputs, the errors that have to be recovered from will essentially all be ones that are wholly within the computer system. In this situation backward error recovery (i.e. recovery to an earlier, it is hoped error-free, state) will suffice, and be readily implementable, such is the nature of computer storage. If such a system is serving the needs of a set of essentially independent users, competing against each other to access and perhaps update the database, then the now extensive literature on database transaction processing and protocols can provide a fertile source of well-engineered, and mathematically well-founded, solutions to such problems [8].

However, the multiple activities in a decentralised system will often not simply be competing against each other for access to some shared internal resource, but rather will on occasion at least be attempting to co-operate with each other, in small or large groups, in pursuit of some common goal. This will make the provision of backward error recovery more complicated than is the case in basic transaction-oriented systems. And the problem of avoiding the “domino effect”[20], in which a single fault can lead to a whole sequence of rollbacks, will be much harder if one cannot disallow (i.e. ignore) the possibility of undetected invalid communications between activities.

When a system of interacting threads employs backward recovery, each thread will be continually establishing and discarding checkpoints, and may also on occasion need to restore its state to one given in a previously established checkpoint. But if interactions are not controlled, and appropriately co-ordinated with checkpoint management, then the rollback of one thread can result in a cascade of rollbacks that could push all the threads back to their beginnings.



Fig, 4. The domino effect

However, the domino effect would not occur if it could safely be assumed that data was fully validated before it was output, i.e. was transmitted from one thread to another. (Similarly, the effect would be avoided if a thread could validate its inputs fully.) Such an assumption is in effect made in simple transaction-based systems, in which outputs are allowed to occur only after a transaction has been "committed". Moreover, in such systems the notion of commitment is regarded as absolute, so that once the commitment has been made, there is no going back, i.e. there is no provision for the possibility that an output was invalid. The notion of nested transactions can be used to limit the amount of activity that has to be abandoned when backward recovery (of small inner transactions) is invoked. However, this notion typically still assumes that there are absolute "outermost" transactions, and that outputs to the world outside the database system, e.g. to the users, that take place after such outermost transactions end must be presumed to be valid.

The conversation scheme [4] provides a means of co-ordinating the recovery provisions of interacting threads so as to avoid the domino effect, without making assumptions regarding output or input validation. Figure 5 shows an example where three threads communicate within a conversation and the threads T1 and T2 communicate within a nested conversation. (Not all of these threads need represent activity inside a computer – some might represent activity in its environment.) Communication can only take place between threads that are participating in a conversation together, so while T1 and T2 are in their inner conversation they cannot damage or be damaged by T3.

The operation of a conversation is as follows: (i) on entry to a conversation a thread establishes a checkpoint; (ii) if an error is detected by any thread then all the participating threads must restore their checkpoints; (iii) after restoration all threads then attempt to make further progress; and (iv) all threads leave the conversation together, only if all pass any acceptance tests that are provided. (If this is not possible, the conversation fails – a situation that causes the enclosing conversation to invoke backward error recovery at its level.)

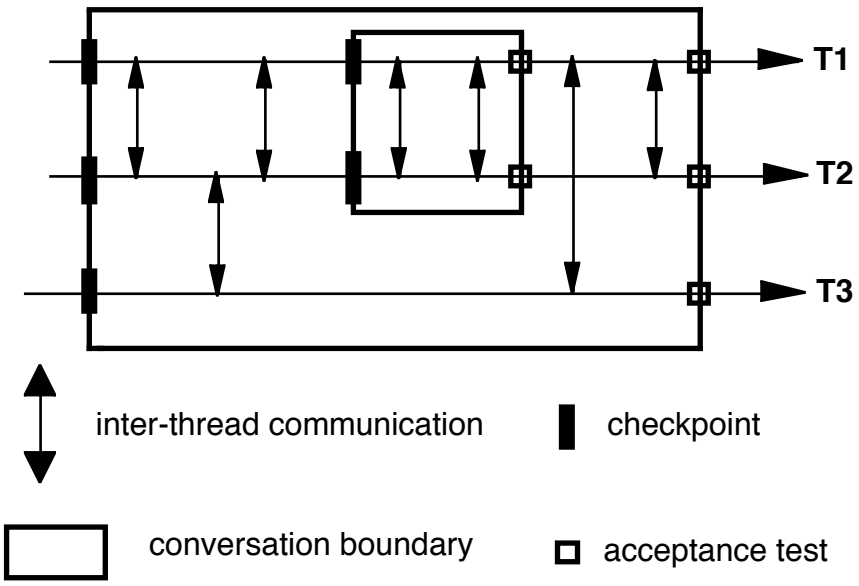


Fig. 5. Nested conversations

Both transactions and conversations are examples of atomic actions [16], in that viewed from the outside, they appear to perform their activity as a single indivisible action. (In practice transaction-support systems also implement other properties, such as "durability", i.e. a guarantee that the results produced by completed transactions will not be lost as a result of a computer hardware fault.) And both rely on backward error recovery.

However, systems are usually not made up just of computers – rather they will also involve other entities (e.g. devices and humans) which in many cases will not be able to simply forget some of their recent activity, and so simply go straight back to an exact earlier state when told that an error has been detected. Thus forward error recovery (the typical programming mechanism for which is exception handling), rather than backward recovery will have to be used. Each of these complications individually makes the task of error recovery more difficult, and together they make it much more challenging. This in fact is the topic that I and colleagues have concentrated on these last few years.

Our Co-ordinated Atomic (CA) Action scheme [23] was arrived at as a result of work on extending the conversation concept so as to allow for the use of forward error recovery, and to allow for both co-operative and competitive concurrency. CA actions can be regarded as providing a discipline, both for programming computers and for controlling their use within an organisation. This discipline is based on nested multi-threaded transactions [5] together with very general exception handling provisions. Within the computer(s), CA actions augment any fault tolerance that is provided by the underlying transaction system by providing means for dealing with (i) unmasked

hardware and software faults that have been reported to the application level to deal with, and/or (ii) application-level failure situations that have to be responded to.

Summarising, the concurrent execution threads participating in a given CA action enter and leave the action synchronously. (This synchronisation might be either actual or logical.) Within the CA action, operations on objects can be performed co-operatively by *roles* executing in parallel. If an error is detected inside a CA action, appropriate forward and/or backward recovery measures must be invoked co-operatively, by all the roles, in order to reach some mutually consistent conclusion. To support backward error recovery, a CA action must provide a recovery line that co-ordinates the recovery points of the objects and threads participating in the action so as to avoid the domino effect. To support forward error recovery, a CA action must provide an effective means of co-ordinating the use of exception handlers. An *acceptance test* can and ideally should be provided in order to determine whether the outcome of the CA action is successful. Error recovery for participating threads of a CA action generally requires the use of explicit error co-ordination mechanisms, i.e. exception handling or backward error recovery within the CA action; objects that are external to the CA action and so can be shared with other actions and threads must provide their own error co-ordination mechanisms and behave atomically with respect to other CA actions and threads.

Figure 6 shows an example in which two concurrent threads enter a CA action in order to play the corresponding roles. Within the CA action the two concurrent roles communicate with each other and manipulate the external objects co-operatively in pursuit of some common goal – portrayed in the Figure by the arrow from Role 1 to Role 2. However, during the execution of the CA action, an exception e is raised by Role 2. Role 1 is then informed of the exception and both roles transfer control to their respective exception handlers H1 and H2 for this particular exception, which then attempt to perform forward error recovery. (When multiple exceptions are raised within an action, a resolution algorithm based on an exception resolution graph [4, 27] is used to identify the appropriate “covering” exception, and hence the set of exception handlers to be used in this situation.) The effects of erroneous operations on external objects are repaired, if possible, by putting the objects into new correct states so that the CA action is able to exit with an acceptable outcome. The two threads leave the CA action synchronously at the end of the action.

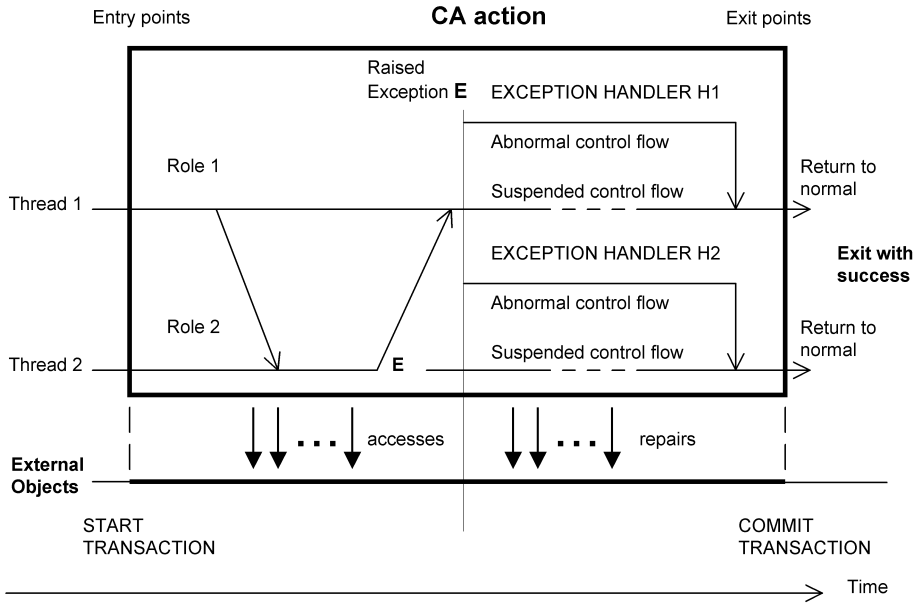


Fig. 6. Example of a CA Action

In general, the desired effect of performing a CA action is specified by an acceptance test. The effect only becomes visible if the test is passed. The acceptance test allows both a normal outcome and one or more exceptional (or degraded) outcomes, with each exceptional outcome signalling a specified exception to the surrounding environment (typically a larger CA action).

We have in recent years, with colleagues in several EU-funded research projects, investigated the advantages and limitations of this approach to structuring systems so as to facilitate the design and validation of sophisticated error recovery, through a series of detailed case studies. Publications describing these include [21, 24-26, 28].

However, my purpose in describing this particular line of development in fault tolerance was not so much to argue the merits of CA actions, but rather to illustrate the crucial role that a designer's choice of fault assumptions should make in directing the subsequent design activity. (For example, the vast majority of research, and practice, in the distributed systems world assumes that a computer fails by crashing – i.e. is a "fail-silent" device, despite the existence of evidence that this is by no means always the case.) The crucial nature of this choice applies not only when one is considering the fault assumptions underlying the design of a fault-tolerant computer, but also the merits of a particular system design, implementation and validation process. (Which if any aspects of this process can justifiably be assumed to be faultless – the specification, the compiler, the formal validation?) Yet all too often, inadequate attention is paid to identifying and justifying a set of fault assumptions – this indeed is one of the major messages I want to put across in this talk.

5 On Structure

Another of the messages that I want to convey is the particular importance of the role that system structuring plays in achieving dependability, especially where such dependability has to be achieved in the face of complex system requirements, and the complex realities of a fault-ridden world. I have had a keen personal interest for many years in the topic of system structuring, initially motivated by work at IBM on methodologies and tools for aiding the design of a large multiprocessing system [29] and then at Newcastle on dependability. The earliest work at Newcastle, on recovery blocks, was in fact all about structuring. Recovery blocks offer a means of introducing lots of extra redundant code into an application (acceptance tests and alternate blocks) without greatly adding to the overall system complexity. Unless this were the case, the scheme would of course be self-defeating.

The recovery block structure, with its underlying recovery cache for automating the provision of checkpoints, avoids causing a complexity explosion by allowing the programming of the different alternate blocks to be performed independently, both of each other, and of the problems of recovering from each other's errors. Thus, as always, structuring is being used as a means of dividing and conquering complexity. However, it is worth distinguishing between different sorts of complexity, and its counterpart, simplicity.

Tony Hoare once said: "The price of reliability is utter simplicity – and this is a price that major software manufacturers find too high to afford!" This is true, but so is Einstein's remark that: "Everything should be made as simple as possible, but not simpler"[3]

As I've discussed above, one can gain much simplicity by making assumptions as to the nature of the faults that will not occur (whether in system specification, design or operation). But this will be a spurious simplicity if the assumptions are false. Good system structuring allows one to deal with the added complexity that result from more realistic fault assumptions.

What is meant here by good structuring is not just the conventional characteristics, such as coupling and cohesion, that are used to determine the impact of structuring on performance, but also a characteristic which might be termed "strength". A strongly-structured system is one in which the structuring exists in the actual system, (as opposed to being used just in descriptions of, or the design for, a system) and helps to limit the impact of any faults – the analogy being to water-tight bulkheads in a ship.

For example, one of the standard (hardware) fault tolerance techniques is Triple Modular Redundancy (TMR) – figure 7 is a typical illustration, found in many textbooks, of part of an overall TMR system, involving a triplicated component and voter.

[3] As quoted in Reader's Digest (British edition), Vol 111, No 666, October 1977, p. 164. The German original is normally given as "Alles sollte so einfach wie möglich gemacht werden, aber nicht einfacher".

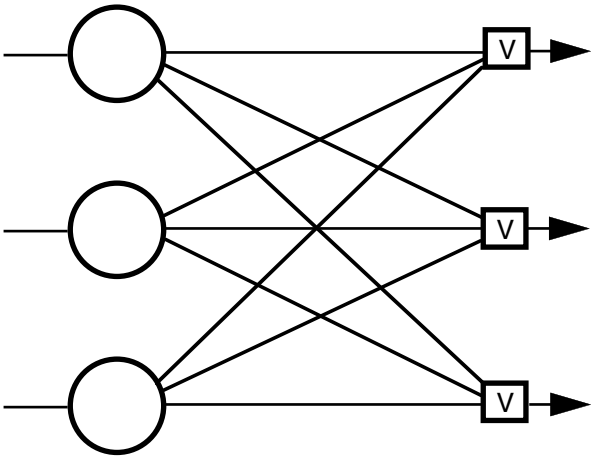


Fig. 7. Triple Modular Redundancy

One of the principal assumptions underlying TMR is that its structuring is strong. A majority vote that is obtained by collusion, whether accidental or deliberate, between two of the triplicated components or voters, is worthless. Thus it is essential that there to be good reason to assume that there is and can be no communication between these components or between these votes – that they are properly insulated from each other, that no-one has accidentally left a screw-driver across them, and – more subtly – that they are indeed wired together in the form shown in the Figure.

Taking a software example, it would be possible to use mere programming conventions to implement a recovery block-like approach. But the scheme is only truly effective if there are some effective means of enforcing the required separation between alternates, for example, so that one can have adequate reason to assume that a residual design fault in one alternate cannot impact any of the other alternates.

6 On Diversity

Rather than continue to develop this theme of the importance of system structuring, let me move on the topics of redundancy and diversity – which are much more specific to fault tolerance.

All fault tolerance involves the use of redundancy – of representation and/or activity – whose consistency can be checked. I have concentrated on what one might term "built-in" fault tolerance – but the system design process can also benefit from redundancy and consistency checking. For example, a system specification is likely to be improved (to the benefit of the dependability of the resulting system) if the specification is scrutinised by knowledgeable system designers who have the task of creating a system to match the specification. This opportunity is of course lost if this process is automated. And indeed, a recent experiment by Ross Anderson has

convincingly shown the advantages of massive (human) redundancy, in developing security specifications [2].

Redundancy takes several different forms. Repeated operation of a single device, or the parallel operation of multiple copies of such a device, can provide means of tolerating some kinds of fault, but not in general design faults. What is needed for this is design (including specification) diversity. Unfortunately, despite its importance, the concept of design diversity is not at all well understood. Like complexity, it is hard to define, leave alone to measure effectively.

The effectiveness of redundancy depends on the extent to which the diversely-designed representations or activities are dependent upon each other with respect to types of fault that are to be tolerated. If they are completely independent – something that is often assumed despite being rarely if ever true – then the probability of coincident faults can be very low indeed. Extremely high dependability could be achieved in such circumstances, for example via the use of majority voting, assuming that the initial non-redundant versions are already reasonably dependable.

Such independence arguments were used as the basis of highly over-optimistic early estimates of the efficacy of N-version programming. In fact it turns out that there is a strong theoretical basis for the non-independence of faults in "independently-designed" software. As explained for example in [15], the demands placed upon systems by their environment can vary in 'difficulty', and this variation induces dependence upon the failure processes of different 'diverse' versions. Nevertheless, redundancy, including software design diversity, can provide considerable added dependability – though it is problematic to predict just how much, given the difficulty of assessing the degree of dependence, indeed the degree of diversity.

Just as deliberate use of diverse designs, or of diversity in the design process, can have significant benefits, accidental lack of diversity can have considerable dangers. This is well known in the world of biology, for example – but the phenomenon is also highly relevant in a computer world in which particular (ad hoc) standard platforms and protocols are becoming increasingly dominant. One might have thought that uniformity would lead to a reduction in complexity that would be very beneficial with respect to dependability – unfortunately life is not so simple. A nine hour outage, on 15 January 1990, of the long-distance phone system in the USA [18] was largely due to the fact that all the switches were of a single common design; and the impact of computer viruses is much greater now that so many people are using basically the same hardware and software.

So much for diversity. The topics that I have discussed so far, namely (i) fault concepts, classification and assumptions, (ii) system structuring, and (iii) redundancy and diversity, are to my mind the perennial central topics underlying the problem of achieving dependability from complex systems. I hope that I have succeeded in bringing out the fact these are a set of, so-to-speak, everlasting dependability research topics – ones that have been studied for years and yet still need much more study. However, let me now, against this background, devote the final part of my talk to a brief summary of one particular research issue in dependability that is I believe particularly topical.

7 Malicious Faults

Recently, some thirty years after we started thinking about design fault tolerance, I had a great sense of deja vu. This wass because I and my colleagues (this time not just in Newcastle, but also from several other research groups across Europe) were undertaking a EU-funded research project, to see whether we could extend the scope of fault tolerance technology (and in our case, ideas such as CA Actions) to cover a type of fault that hitherto has largely been regarded as one to be prevented and/or removed, rather than coped with automatically. This is the intentional malicious fault, arising from the nefarious activities of hackers and – much worse – corrupt insiders (including people who have systems administration roles).

Such protection is needed because the likely reality is that most large systems will have to be used even though it is known that they contain vulnerabilities. Some of these vulnerabilities might even have been identified already, but for some reason must be allowed to remain in the system; other vulnerabilities will be awaiting discovery – probably first by system hackers. Thus means for tolerating malicious faults are needed, not just for reporting detected intrusions to the management, if continuous service is needed from the system.

Over a decade ago I and a colleague, John Dobson, first started to think about this sort of problem, though we did not develop the idea extensively at the time. The main result of our work was a paper [7] whose title "Building reliable secure systems from unreliable insecure components" (a deliberate allusion to von Neumann) both neatly summed up our approach, and provoked one of the referees of the conference at which it was presented, the IEEE Oakland "Privacy and Security" conference, to describe it as "highly controversial" – though now I think the idea, or at least the aim, is more accepted.

Indeed, the recently-completed collaborative research project to which I alluded above, namely MAFTIA (standing for "Malicious- and Accidental- Fault-tolerant Internet Applications"), brought together teams working on encryption, intrusion detection, asynchronous distributed algorithms, rigorous evaluation and, of course, fault tolerance. The project's major innovation was a comprehensive approach for tolerating both accidental faults and malicious attacks in large-scale distributed systems, including attacks by external hackers and by corrupt insiders. However, this is a whole story in itself–full details can be found from the project's web-site, at:

http://www.newcastle.research.ec.org/maftia/

8 Concluding Remark

Much of this lecture has been aimed at trying to explain what I believe to be some of the most important issues of long term and continuing importance in dependability. But my fundamental aims in this lecture, implied by its title, have been first to argue how important it is to accept the reality of human fallibility and frailty, both in the design and the use of computer systems, and second to indicate various constructive approaches to trying to cope with this uncomfortable reality. If such an acceptance were more prevalent in the computer science community, it would I believe go some way toward improving our standing, and that of our subject, among the general public.

9 Acknowledgements

In this talk I have attempted to cover a wide variety of topics related to system dependability, and have been drawing not just on my own work but also on that of many colleagues, both past and present. Alluding to another famous quotation, let me say that though I do not claim to be able to see further than other people, I have stood on many people's shoulders. It is therefore a pleasure to acknowledge the great debt I owe to many colleagues at Newcastle and, especially in recent years, to colleagues in the ESPRIT PDCS and DeVa Projects, the IST MAFTIA and DSoS Projects, and the EPSRC Interdisciplinary Research Collaboration on the Dependability of Computer-Based Systems (DIRC).

References

1. Alexander, C. Notes on the Synthesis of Form. Harvard University Press, Cambridge, Mass., USA, 1964.
2. Anderson, R., How to Cheat at the Lottery (or, Massively Parallel Requirements Engineering). in Proc. Computer Security Applications Conference, (Phoenix, AZ, 1999).
3. Avizienis, A., Laprie, J.C. and Randell, B., Fundamental Concepts of Dependability. in Third IEEE Information Survivability Workshop, (Cambridge, Mass., 2000), Software Engineering Institute, Carnegie-Mellon University, Pittsburg, 7-12.
4. Campbell, R.H. and Randell, B. Error Recovery in Asynchronous Systems. IEEE Trans. Software Engineering, **SE-12** (8). 811-826.
5. Caughey, S.J., Little, M.C. and Shrivastava, S.K., Checked Transactions in an Asynchronous Message Passing Environment. in 1st IEEE International Symposium on Object-Oriented Real-time Distributed Computing, (Kyoto, 1998), 222-229.
6. Davies, C.T. Data processing spheres of control. IBM Systems Journal, **17** (2). 179-198.
7. Dobson, J.E. and Randell, B., Building Reliable Secure Systems out of Unreliable Insecure Components. in Proc. Conf. on Security and Privacy, (Oakland, 1986), IEEE.
8. Gray, J. and Reuter, A. Transaction Processing: Concepts and techniques. Morgan Kaufmann, 1993.
9. Horning, J.J., Lauer, H.C., Melliar-Smith, P.M. and Randell, B., A Program Structure for Error Detection and Recovery. in Proc. Conf. on Operating Systems, Theoretical and Practical Aspects (Lecture Notes in Computer Science, vol. 16), (IRIA, 1974), Springer Verlag, 171-187.
10. Horning, J.J. and Randell, B. Process Structuring. ACM Computing Surveys, **5** (1). 5-30.

11. Jones, C.B. A Formal Basis for some Dependability Notions. in Aichernig, B.K. and Maibaum, T. eds. Formal Methods at the Crossroads: from Panacea to Foundational Support, Springer-Verlag, 2003.
12. Laprie, J.C. (ed.), Dependability: Basic concepts and associated terminology. Springer-Verlag, 1991.
13. Laprie, J.C. (ed.), Dependability: Basic concepts and terminology — in English, French, German, Italian and Japanese. Springer-Verlag, Vienna, Austria, 1992.
14. Laprie, J.C., Dependable Computing: Concepts, Limits, Challenges. in 25th IEEE International Symposium on Fault-Tolerant Computing - Special Issue, (Pasadena, California, USA, 1995), IEEE, 42-54.
15. Littlewood, B. and Miller, D.R. Conceptual Modelling of Coincident Failures in Multi-Version Software. IEEE Trans. Software Engineering, **15** (12). 1596-1614.
16. Lomet, D.B. Process Structuring, Synchronization, and Recovery Using Atomic Actions. ACM SIGPLAN Notices, 12 (3). 128-137.
17. Naur, P. and Randell, B. (eds.). Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968. Scientific Affairs Division, NATO, Brussels, 1969.
18. Neumann, P. Computer Related Risks. Addison-Wesley, New York, 1995.
19. Randell, B. Facing up to Faults (Turing Memorial Lecture). Computer Journal, **43** (2). 95-106.
20. Randell, B. System Structure for Software Fault Tolerance. IEEE Trans. on Software Engineering, **SE-1** (2). 220-232.
21. Romanovsky, A., Xu, J. and Randell, B., Exception Handling in Object-Oriented Real-Time Distributed Systems. in Proc. 1st IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC'98), (Kyoto, Japan, 1998), 32-42.
22. von Neumann, J. Probabilistic Logic and the Synthesis of Reliable Organisms from Unreliable Components. in Shannon, C.E. and McCarthy, J. eds. Automata Studies, Princeton University Press, Princeton, NJ, 1956, 43-98.
23. Xu, J., Randell, B., Romanovsky, A., Stroud, R.J. and Wu, Z., Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. in Proc. 25th Int. Symp. Fault-Tolerant Computing (FTCS-25), (Los Angeles, 1995), IEEE Computer Society Press.
24. Xu, J., Randell, B., Romanovsky, A., Stroud, R.J., Zorzo, A., Canver, E. and Henke, F.v., Developing Control Software for Production Cell II: Failure Analysis and System Design Using CA Actions. in FTCS-29, (Madison, USA, 1999), IEEE CS Press.
25. Xu, J., Randell, B., Romanovsky, A., Stroud, R.J., Zorzo, A.F., Canver, E. and Henke, F.v., Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions. in Proc. 29th Int. Symp. Fault-Tolerant Computing (FTCS-29), (Madison, 1999), IEEE Computer Society Press.
26. Xu, J., Randell, B., Romanovsky, A., Stroud, R.J., Zorzo, A.F., Canver, E. and Henke, F.v. Rigorous development of an Embedded Fault-Tolerant System Based on Coordinated Atomic Actions. IEEE Trans. on Computers (Special Issue on Fault Tolerance), **51** (2). 164-179.
27. Xu, J., Romanovsky, A. and Randell, B., Co-ordinated Exception Handling in Distributed Object Systems: from Model to System Implementation. in Proc. 18th

IEEE International Conference on Distributed Computing Systems, (Amsterdam, Netherlands, 1998), 12-21.
28. Zorzo, A.F., Romanovsky, A., J. Xu, B.R., Stroud, R.J. and Welch, I.S. Using Co-ordinated Atomic Actions to Design Complex Safety-Critical Systems: The Production Cell Case Study. Software — Practice & Experience, **29** (8). 677-697.
29. Zurcher, F.W. and Randell, B., Iterative Multi-Level modelling: A methodology for computer system design. in Proc. IFIP Congress 68, (Edinburgh, 1968), D138-D142.