Fault Tolerance Against Design Faults

Lorenzo Strigini

Centre for Software Reliability, City University
Northampton Square, London EC1V OHB, U.K.
E-mail: strigini@csr.city.ac.uk

Abstract

This chapter surveys techniques for tolerating the effects of design defects in computer systems, paying special attention to software. Design faults are a major cause of failure in modern computer systems, and their relative importance is growing as techniques for tolerating physical faults gain wider acceptance. Although design faults could in principle be eliminated, in practice they are inevitable in many categories of systems, and designers need to apply fault tolerance for mitigating their effects. Limited degrees of fault tolerance in software – "defensive programming" – are common, but systematic application of fault tolerance for design faults is still rare and mostly limited to highly critical systems. However, the increasing dependence of system designers on off-the-shelf components often makes fault tolerance a necessary, feasible and probably cost-effective solution for achieving modest dependability improvements at affordable cost. This chapter introduces techniques and principles, outlines similarities and differences with fault tolerance against physical faults, provides a structured description of the space of design solutions, and discusses some design issues and trade-offs.

1. Introduction

"Systematic failures" - failures that are not due to random physical decay but to defects in the design or manufacturing of products, and therefore happen systematically when certain circumstances occur in the use of the product - are a common problem in engineering. Among their causes are *design faults*: avoidable defects due to human error or lack of foresight in developing the system. The advent of software gave new prominence to design faults, since software products are uniquely defined as being a purely intellectual construction, immune from physical damage or decay, and yet they were seen to fail often, to the point of being the main cause of failure of many systems. For computer systems, design faults – in software, less often in hardware, and in organizations and procedures of use – are reported to account for a large fraction of failures, including high consequence ones, in anecdotal evidence [1], and published statistics (e.g. in [2], between 25% and 65% of failures in different categories of systems). The importance of physical faults in determining unreliability of computer systems is decreasing compared to that of design faults and human error in operation and maintenance. (If some human errors were attributed to faults in the design of user interfaces and procedures, the relative weight of design faults would increase further.) This trend may be due in part to some categories of software becoming less reliable, but also to an increasing ability to achieve high reliability against physical faults of hardware, in part via fault tolerance. Techniques for avoiding or removing design faults have improved over the years, but the problem of design faults is still serious in at least two respects: i) there are safety- and mission-critical systems that require better dependability than current methods can verify, or possibly even achieve [3]; and ii) for a large part of the industry that produces hardware and software components for purchase "off the shelf", dependability has low priority [4], so that users, or

This is a chapter written for the book "Dependable Computing Systems: Paradigms, Performance Issues, and Applications", edited by Hassan B. Diab and Albert Y. Zomaya, to be published by Wiley (ISBN: 0-471-67422-2).

system integrators, often find that these components (especially software components) do not deliver the dependability levels they require, even when these are comparatively modest. Hence the appeal of fault tolerance techniques, which use redundancy to eliminate, reduce or make less harmful the disruption caused by design faults in the service provided by a system. The principle of fault tolerance against software faults was first advocated, and possible structures proposed, in the '70s [5, 6].

This paper - like most research so far - puts special emphasis on fault tolerance for *software* design faults, although many of the principles involved apply just as well to tolerating, e.g., design faults in computer hardware. These are also widespread [7] and probably growing in importance with the increasing scale of integration and complexity of hardware systems, but their documented contribution to system unreliability is generally far less important than that of software bugs.

From one viewpoint, software fault tolerance is just "defensive programming", which has always been part of software practice, possibly with some added emphasis on structuring principles to guide its application and contain its complexity. From a different point of view, any emphasis on providing fault tolerance for design faults is, in this author's experience, a radical change from the common attitudes of many practitioners and researchers alike. Many see fault tolerance to design faults as a low-quality solution, compared to the more desirable goal of fault-free software. According to this view, fault tolerance may be necessary in hardware because physical faults are inevitable; but in software the only reason for failures is design error, something that should be tracked and eliminated. So, emphasis on techniques for "getting things right" has sometimes led to neglecting the need for systems to survive even if design errors slip through the process and lead to design faults in the final product. This negative attitude is – luckily – receding. One reason is the recognition that most system designers and users simply cannot obtain software that is free of design faults. As more system designers use "off-the-shelf" software - or whole computer systems - as components, which they cannot change, they have to confront the fact that these components may fail, and yet their systems must be dependable enough despite these failures. A parallel can be drawn with the evolution of attitudes towards operator error, and the increasing recognition that designers of automation must take into account the natural error-proneness of operators [8]. This change is noticeable even in areas, like security, where the preference for seeking "perfect" solutions used to be most pervasive [9, 10]. Methods for designing correctly, for removing design faults during the evolution of systems, and for tolerating the failures they cause are all necessary.

There is no sharp boundary between fault-tolerant designs that apply against design faults and against physical faults, as the consequences of design faults are often similar to those of physical faults (especially transient faults). For instance, a programmer error or compiler error may create a design fault that in certain conditions of execution (created by chance or by malicious attack) will cause a wrong jump in the program control flow or corrupt the value of a variable: consequences that could also be caused by a physical cause, e.g. electromagnetic interference. What is different is the kind of precautions needed so that faults do not cause the fault tolerance mechanisms themselves to fail when their intervention is required: hence an emphasis on "diversity" of design within a redundant system, when the redundancy is meant to tolerate design faults.

This chapter focuses on what is specific to tolerating design faults. Even so, the topic cannot be surveyed in depth in the space available. The references at the end of the chapter point to several survey and reference works which provide more extensive coverage and further bibliography [11, 12, 13].

The rest of the chapter is organized as follows. Section 2 will propose some examples of fault-tolerant design as they apply to design faults, and discuss the principles applied, in particular the role of *diversity* and the trade-offs between different dependability goals. Section 3 deals with assessing the dependability levels that these techniques can or do achieve, and their practical desirability in various contexts. Section 4 goes into more detailed aspects of design, discussing the design choices available to a designer of a software fault-tolerant system, the constraints imposed by the characteristics of the applications, and the design problems that they require to be solved. Section 5 contains a summary of established results and open issues.

2 Examples and principles

2.1 Fault-tolerant components

If creating or obtaining components without design faults (fault avoidance) is infeasible, a designer must assume that errors will happen, and build additional "checks and balances" – redundancy – so that errors in the operation of a system are detected in time and their effects corrected as far as possible. Ideally, the aim is to correct an erroneous internal state of the system before it propagates to cause an externally observable failure[1]. For simple examples of such redundancy (more complex design issues will be discussed in Section 4), let us consider the example in Fig. 1: within a system S, we look at a component C1, which receives inputs from other components and produces outputs which are used as inputs by other components – in this instance just one other component, C2. These "components" may be of any size and nature: complete digital computers with the software running on them, hardware logic modules, software subsystems, individual processes, or individual instantiations of procedures, objects or smaller program modules, including sections of execution of sequential processes. C1 is represented in Fig. 1 as a black box, which performs a certain computation (specified here as computing an output as a function f_{C1} of the component's input and internal state).

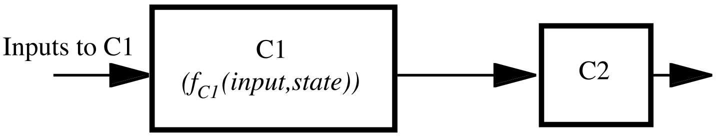

Fig. 1: Reference scenario for examples of fault-tolerant design

A system designer who feared that C1 could perform incorrectly too often could substitute for it the "C1 (self-checked)" component in Fig. 2. "C1" is still meant to perform the task specified for it. The "checker" block applies tests to its output to detect at least some of the possible erroneous results. The "switch" block forwards the result to its intended user (C2) if no error has been detected, otherwise it sends an error message (in some designs) or nothing (in others, where for instance C2 has the ability to recognize the absence of a message and deal with the problem). This scheme can be applied at all levels of scale, from in-line run-time checks in the code of every program unit, familiar to most programmers, to encapsulating complete, complex products.

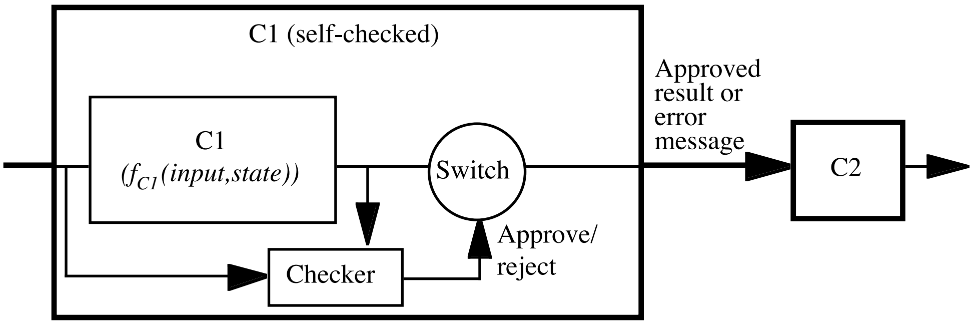

Fig. 2: System with self-checking design of C1

[1] Defining a few terms at this point is appropriate to avoid ambiguity. In this chapter, a *system* is made up of *components*, which can be seen as "systems" (i.e., further subdivided in components) in their own right (in which case they are often referred to as *subsystems*). A system is meant to provide services (to other systems, which may include human users), and *fails* when its behavior departs from what was intended. When it is useful to identify a cause of failures, this is called a *fault*. The terms "design fault", "design defect", "bug" are used as synonyms. Faults may lead to erroneous states, or *error*s, inside a system, which do not become system failures until they manifest themselves as deviations of the system's externally visible behavior from the intended service. Thus, fault tolerance is meant to prevent errors inside a system, or failures of its components, from becoming system failures. *Dependability* is an umbrella term for qualities like reliability, availability, safety, which can be characterized by various attributes and quantitative measures. These terms are drawn from the IFIP Working Group 10.4's work [14], and from [15].

With this design scheme, the system designer substitutes the original C1 with a slightly more complex "self-checked component" which, upon most invocations, produces the same output as C1 would, but which produces uncontrolled erroneous outputs less often. When the checker detects an error, the output of "C1 (self-checked)" will not be the proper value of the function f_{C1}. But it will be a behavior that does not masquerade as correct behavior, and the rest of the system can then be designed to react to this special behavior in a more desirable way than if C1 simply delivered an incorrect output without warning. For instance, batch programs may be designed to terminate with an error message for a human to take charge of the problem; safety-critical processes may be designed to switch to a safe state; etc.

Of course, the designer will need to address many problems. For instance, can one invent suitable checks for detecting most possible errors of C1? Might the "checker" require too much time for its operation? How easy is it to devise an adequate reaction by the rest of the system to an error message?

Another scheme, "multiple-version software" (or "N-version software") is represented in Fig. 3. Here, in an architecture similar to "multiple modular redundancy", there are several (three in the figure) parallel "channels", each one computing function f_{C1}, but, instead of using three copies of C1, they use two additional "versions" or "variants" of C1. These are other components that have to satisfy the same specification as C1, but are developed separately from it and from each other. All receive copies of the input, and all perform the computation. Normally, all three will produce identical, correct results[2]. Even if one of them fails, but the other two are correct, the output of the majority voter will be the correct value of f_{C1}. If, among the situations in which at least one version fails, those in which *only* one fails are more frequent than those in which two or more fail, then, again, this fault-tolerant component will be one that produces uncontrolled erroneous outputs less often than C1 alone, and, besides, produces exactly the specified output more often than C1 alone. To secure this improvement, one attempts to procure the multiple versions in ways that should make it unlikely that they contain similar bugs.

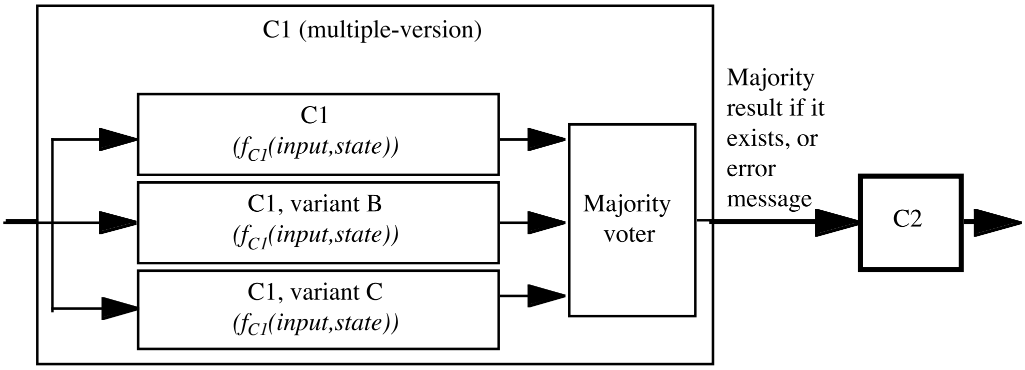

Fig. 3. System with multiple-version fault-tolerant design of component C1

Many more design solution are possible, but these two allow some early, general, observations:

- both examples illustrate a "component-structured" approach to fault tolerance: the designers applies some form of protective redundancy to the whole component, by adding fault-tolerant mechanisms that act at the interfaces between these components. This creates a *fault containment unit*, i.e., the redundancy is there to prevent the consequences of faults in the component from propagating outside it. The other components that would use this unit's results see either a correct result or a "clean" failure, like absence of output or an explicit error message. Making C1 fault-tolerant does not change its role in the system, i.e., those properties of its behavior that are assumed in the design of the other components (except perhaps in details of its reactions to errors), and thus does not change the way a designer would go about verifying that system S satisfies its specifications (at least in the absence of faults) or selecting or designing the other components.

[2] Multiple version software can also be used if the specification of C1 is such that results of correct versions need not be identical. Section 4 discusses the implications in more detail. The specification of the "voter" component can be generalized to that of an "*adjudicator*" that can cope with consistent though different results [16].

Each interaction is therefore a *decision point*, where error detection, masking or recovery may be initiated[3];

- these schemes may be applied to any component, and may, in principle, be repeated in a nested fashion. If the system S of which Figures 1-3 represent a detail is a component of a larger system, its failures are still internal errors for the larger system, and can still be confined inside a fault-tolerant variant of S;
- in both examples, it is possible for the fault tolerance at times to fail, i.e., to allow a failure of a functional component to become a failure of the whole fault-tolerant component. In the self-checked component, component C1 may sometimes fail with an erroneous result of a kind that the checker will not recognize; in the multiple-version component, it may happen that two or all three variants produce consistent, wrong outputs on the same invocation, so that the voter will produce an incorrect majority result. So, a fault-tolerant component fails on certain forms of *common failure* between its redundant components. To decide whether a fault-tolerant system or component is dependable enough, one needs to estimate the probabilities of these common failures. These probabilities can be improved (reduced) either by making C1 less likely to fail, or by increasing the effectiveness of the fault-tolerant mechanisms: by providing more effective checkers, in Fig. 2, or by reducing the similarity among the situations in which the variants fail, in Fig. 3. What improvements are feasible and cost-effective will depend on the details of each design, and especially of the function that the component is specified to perform; e.g., whether, in Fig. 2, checkers can be re-designed to improve their effectiveness (coverage) against the specific (but usually unknown to the designers) failure modes of C1;
- achieving fault tolerance has thus two aspects: building a structure that includes the necessary redundancy, and ensuring that this redundancy is sufficiently effective. These two aspects are discussed next.

2.2 Fault-tolerant design: redundancy and failure diversity

Designing fault tolerance for design faults will require the same steps as the design of fault tolerance against any other type of fault for any other purpose [15, 19, 20]:

- identifying dependability requirements and threats to their satisfaction: probable fault types and where they would cause erroneous states or signals;
- deciding which errors would cause failures with high enough probability and/or severity of effects that fault tolerance should be applied against them
- identifying fault-tolerant mechanisms that appear feasible and practical for coping with those errors. A fault tolerance strategy will usually combine some of these steps:
  - error detection
  - error confinement, preventing propagation of the erroneous information, and/or error masking, guaranteeing correct service despite the error
  - damage assessment and diagnosis of the origin of the error, to guide the following steps
  - reconfiguration, to exclude erroneous components from further computation
  - recovery, to correct an erroneous system state. This may be achieved directly, exploiting redundancy in the data, so as to allow the subsequent computation to proceed without propagating the effects of the error (forward recovery); or by undoing all the computation performed after the error, in order to redo it without repeating the error (backward error recovery);
  - fault treatment: correcting the design faults identified thanks to the errors detected,
  - reintegration of components that were excluded for recovery or fault treatment
- verifying the effectiveness of fault tolerance, usually in two stages:
  - deterministic checks that the fault-tolerant mechanisms, if completely effective, would block all the error types and combinations that they are intended to defend against;
  - probabilistic assessment, i.e., checking that, based on the available evidence about the probabilities of the various error types and the effectiveness of the mechanisms, the probabilities of the various kinds of system failures have been reduced to acceptable levels. This is by far the more difficult step.

[3] This does not exclude the possibility of error detection taking place inside the component as well. In some proposed designs for applications with long-lived processes, the decision points can be between segments of execution of a process [17]. With real-time embedded systems based on repetitive tasks, the decision points can be between successive executions of the same task[18].

The design process will usually iterate these steps to seek a solution with satisfactory effectiveness and cost.

In most of the steps listed, the designer is only concerned with the properties indicated under "deterministic checks". The design goal will be to ensure that mechanisms are in place wherever they are thought necessary, and that the way they are added does not add excessive complexity to the system's design. We can call this "design for redundancy", as opposed to "design for failure diversity", which is instead concerned with the effectiveness of these mechanisms: how likely they are actually to prevent system failures, i.e., not to fail when they are needed. It should be noted that designers of fault-tolerant systems use "*design diversity"* between redundant components inside a fault-tolerant component (for instance, they complement component C1 in the examples above with additional, different versions of C1, or with a checker that does not depend on repeating C1's computation) to achieve a high degree of "*failure diversity*": a low (ideally 0) probability of these redundant components failing at the same time[4], and secondarily, if they do fail at the same time, a high probability of their failures being detectable (e.g., if two or three versions in a 3-version component fail, their erroneous outputs should differ so that the voter will detect the lack of a majority). The distinction between "design diversity" and "failure diversity" – between means and goals – is important, and the next section will examine it in more detail.

2.3 The role of diversity

It is often said that *diversity* between the design of redundant components (as opposed to simple redundancy, provided by replicating identically designed components) is a necessary condition for tolerating design faults: "diversity" or "dissimilarity" is often used as a synonym for "software fault tolerance". Indeed, in a simple fault-tolerant system as in Fig. 3, if all the versions were identical, one would expect that a demand[5] that makes one of them fail would make them all fail in identical ways. It is to avoid these *common mode failures* that the variants are required to be diverse, so that they will not be too likely to contain identical bugs that cause identical errors on the same inputs.

In practice, some degree of software fault tolerance is often obtained without design diversity. This usually depends on a form of "data diversity". For instance, if replicated embedded computer systems as in Fig. 3 read data from replicated sensors, their readings will usually differ slightly: even if one replica of the software fails, another, identical one, receiving slightly different inputs, may perform correctly. It is also often possible to repeat a desired computation without repeating exactly the same demand. It is common experience that if a text processor crashes the user can often restart it and successfully re-enter the same command that previously caused the crash. By not repeating the identical sequence of previous commands, the user prevents the reoccurrence of the same internal state, so that the specific command that caused the failure before does not cause a failure again. Ammann and Knight experimented [21] with small perturbations on numerical inputs to programs, and confirmed that, after a randomly selected input caused a failure, submitting a slightly changed input would be less likely – sometimes drastically so – to cause failure; their paper proposes several other applications of the same principle. A similar phenomenon explains the success of "software rejuvenation" [22]: improving software reliability by frequent restarts, which reset the software's internal state. Gray [23] observed that Tandem systems, with two computers running identical copies of software tolerated many software failures. Thanks to the loose synchronization between the replicated computations, complex failure-causing conditions of the combined application and operating system states would often affect only one replica. An analysis of Tandem system operation found that 82% of the detected software-caused component failures were tolerated [24].

The empirical evidence cited, however, still shows that a sizeable fraction of the design-caused failures would affect all the redundant copies of a computation, if performed by identical code.

To seek diversity between program versions for software fault tolerance, researchers have recommended various measures [25]: strict separation between the developers of the different versions,

[4] I.e., at the same invocation of the fault-tolerant component, or more generally, at times near enough to each other to allow a failure of the fault-tolerant component.

[5] The word "demand" will be used to designate the values of all the variables that affect the behavior of a component: typically, for software, its inputs, its memory variables and program counter, and the states of any other components that affect its behavior.

avoiding the use of common libraries or compilers (which may induce similar bugs into the two designs) and also "forced" diversity, i.e., imposing the use of different language and tools, as well as different design solutions, to avoid possible causes of common mistakes. The diversity may extend where possible (as in certain process control or safety applications) to having software-based versions in parallel with others that use hard-wired logic or analog hardware implementations; and/or to feeding the diverse versions inputs from sensors measuring different aspects of the physical process controlled (this is sometimes called "*functional* diversity").

Though these all appear reasonable recommendations, their effectiveness is difficult to demonstrate and compare: they aim to influence the likelihood of *similar mistakes*, hoping thus to decrease the probability of such *faults* in the design of the multiple versions that might cause *common failures*. These are three distinct causal mechanisms that are imperfectly understood and need further investigation, combining the study of human error with that of the effects of faults on failures [26, 27 , 28]. A potential difficulty is that the goal of forcing diversity may conflict with that of obtaining high reliability from the individual versions. It may be that the design and development methods most suitable for producing a reliable component lead to similar implementations, where the most likely bugs are similar and likely to cause similar errors in all versions. This leads to a dilemma: it seems undesirable to compromise the quality of the individual versions, but, on the other hand, a redundant system made from more diverse, though individually less reliable versions may exhibit fewer failures than one made from more reliable, but more similar ones. It is easy to describe the possible trade-off in mathematical, probabilistic terms [29], but the available empirical knowledge is insufficient for guiding choices among specific alternative combinations of development methods.

2.4 Possible dependability goals

Design fault tolerance approaches can be used for all kinds of dependability objectives.

From the viewpoint of design for redundancy, the same spectrum of requirements can be defined, with respect to any given type of component failure, as with all fault-tolerant design. At one end of the spectrum, the "multiple version" solution in Fig. 3 aims to completely *mask* faults: no system failure is caused, so long as the numbers and combinations of component failures do not exceed some specified threshold (one version failure, in Fig. 3). It is also common to have requirements for graceful degradation: when the system cannot provide the intended service, it should still provide some kind of reduced service. The extreme case of requirement for degraded service is that the fault-tolerant component at least avoid those types of failure behaviors that would be too difficult for the designer to control through actions in the rest of the system, leading to solutions like the "self-checking component" of Fig. 2. Such guarantees of "clean" component failures can greatly simplify the task of designing a fault-tolerant system [30 , 31].

From the viewpoint of how system failures affect people and organizations, dependability requirements are often grouped into three (overlapping) groups: guaranteeing continuity of service (in the form of avoiding failures – *reliability*, or limiting their overall duration – *availability*[6]), avoiding those failures that are considered especially dangerous (*safety*); and protecting against failures that are due to, or would open the way to, malicious activities against the system (*security*). Fault tolerance can be applied towards all three kinds of goals[7].

[6] Towards the goal of improving availability, fault tolerance may be used directly to reduce the probability of system failure, or instead to reduce the duration of down-time subsequent to a failure. For instance, a designer can increase the availability of a server by providing additional copies (or diverse versions) of the server as "stand-by" servers, to take over in case the "primary" server fails. If the switch-over to a stand-by server is immediate, users perceive no failure, i.e., they enjoy improved reliability and availability of the server; if the switch-over is not immediate, but still faster than waiting for the primary server to be readied for resuming service, users still experience improved availability. For the sake of completeness the case should be mentioned in which the additional servers are always active and share the load of the primary, i.e., they improve *performability* (the statistics of the service's performance).

[7] Some authors reserve the name "fault tolerance" for designs that aim at continuity of service, excluding those that provide safety at the expense of availability, like the one in Fig. 5. But, as the examples that we will describe show, identical design principles at the component level can serve either goal at the system level. It is sound practice to keep the concerns of safety distinct from those of

Two additional examples will illustrate the interplay of these properties at the component and system levels.

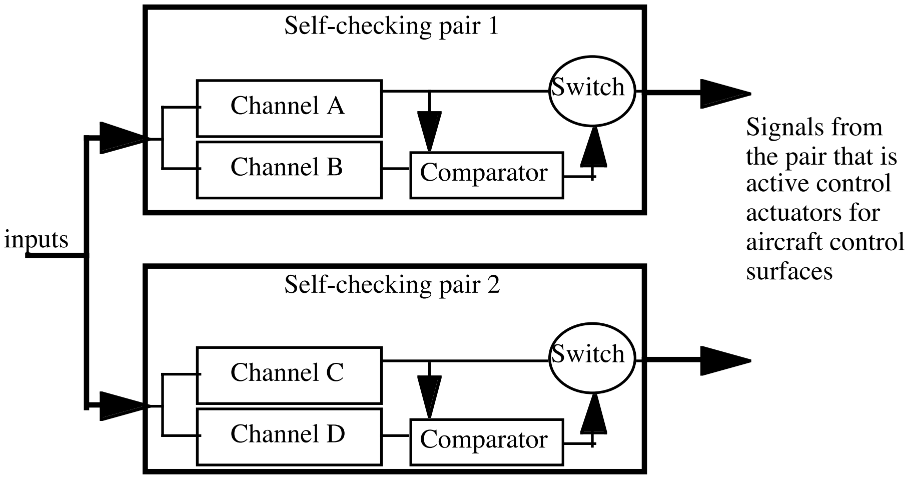

Fig. 4 A configuration of two self-checking pairs, common in aviation applications. When used with diversity, it can tolerate (hardware- or software-caused) failures of any one of the four parallel channels.

Fig. 4 represents a scheme similar to that applied to provide flight-critical functions in the Airbus family of "fly-by-wire" aircraft [32], following a common fault-tolerant configuration. but also often used in non-diverse applications. It uses four parallel channels (built on separate hardware, with separate power sources and all other usual precautions for tolerating physical faults), with different versions of hardware and software. Instead of using voting as in Fig. 3, the channels are arranged in two self-checking pairs (an "N self-checking programming" design [33]). In each pair, one channel plus a comparator act as the "checker" of Fig. 2, preventing the pair from outputting erroneous commands. So, any one failure of a channel is tolerated by a pair going silent, and so the other pair takes over in controlling the aircraft. The individual pair satisfies a safety-like requirement (avoiding potentially dangerous outputs), to satisfy a reliability and availability requirement on the whole control system (continuous provision of correct control outputs), to satisfy a safety requirement of the surrounding controlled system (keeping the aircraft safely airborne).

A standard architecture for safety is similar to Fig. 2, with C1 representing the active control system for a physical process and the checker ("protection system" or "interlock") verifying that C1's commands do not violate safety requirements for the controlled process, and usually able to order the controlled system to a safe state. Focusing the checks on safety properties, rather than on correct execution of the intended control algorithm, is a way of both improving the reliability achievable for the checker (by simplifying its specification) and making it more diverse from the control system, thus decreasing the probability of common failures of the two causing accidents. In a railway application [34], for instance, the checker is implemented as a rule-based system designed directly from the pertinent safety regulations for railway operation.

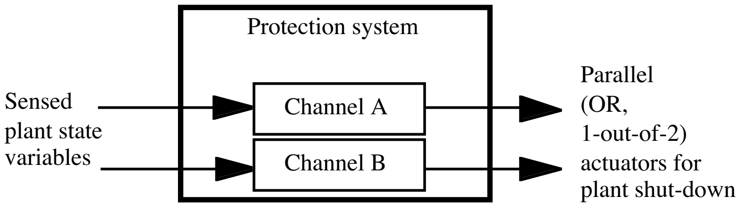
Protection system
Sensed plant state variables
Channel A
Channel B
Parallel (OR, 1-out-of-2) actuators for plant shut-down

Fig. 5 A safety-oriented architecture using diversity.

availability in the management process, but trying to separate designs according to the same criteria would hide useful common principles.

The protection systems themselves may require fault tolerance against design faults. Fig. 5 represents a protection system for a potentially dangerous plant (e.g., a chemical plant or nuclear reactor). The plant has a separate control system, meant to keep it operating in a near-to-optimal regime, and to keep it within a safe "envelope" of operation. The protection system's function is to detect any departure from the safe envelope – a potentially hazardous situation – and issue commands to shut down the plant. There are two diverse channels (again protected against physical faults by various precautions), each one controlling actuators that can shut down the plant, irrespective of what the other channel orders. "Functional diversity" may be applied, monitoring different physical variables (e.g., the temperature and pressure of the reactor) which would give warning of hazards. In some designs, one of the channels is software-based, the other one being a simpler, hardware design. The software-based channel can implement more refined algorithms for handling plant shut-down with lower stress on the protected plant, and so it can be configured to intervene before the hardware channel does, to reduce the cost and risk of shut-down events. The simplicity of the hardware channel, in turn, provides assurance for its role as the last line of defense.

This "1-out-of-2" configuration[8] for the shut-down function gives lower *probability of failure on demand* (*pfd*, i.e., failure to shut down the plant when necessary) than one would obtain by adding a third version, channel C, to create a voted, 2-out-of-3 configuration: on any demand on which the 1-out-of-2 system fails (i.e., both channels A and B fail), so would the 2-out-of-3 system; plus the presence of C may cause the 2-out-of-3 system to fail on additional demands, on which only one among A and B fails, but C also does. The 1-out-of-2 configuration's strong guarantee against unsafe failures (i.e., high reliability of the safety shut-down service) comes at the cost of higher probability of safer, but still expensive, failures: any time one of the two channels erroneously decides that there is a hazard, the plant will shut down. This example illustrates the common situations in which safety and availability concerns create opposite requirements on design, requiring a reasoned trade-off. Similar trade-offs exist at the component level. For instance, in the self-checking configuration of Fig. 2, a designer may consider that software often produces correct results despite internal errors: a stringent check used to command a clean failure of the whole application, or a thorough recovery before operation can continue, while reducing the probability of unpredictable, potentially expensive failure modes, will also reduce its probability of delivering a correct service, or its performance in doing so.

Similar conflicts between different dependability requirements can arise when the purpose of fault tolerance is security. For instance, a service can be provided on the World-Wide Web via multiple servers, which use different operating systems and application software [35]. This redundancy can be used to enhance continuity of service, as an attack is less likely to disable all the servers at the same time than to disable a single server. However, concern for avoiding catastrophic damage to the servers' content might recommend shutting down the service when discrepancies between the servers indicate a *possible* successful attack on one of them, thus using the redundancy to improve data integrity at the expense of availability.

3. Potential and actual benefits

3.1 Potential benefits

The purpose of fault tolerance is to reduce the frequency of failures (or of specific kinds of failures), and/or to mitigate the severity of their effects. Designs like those in Figures 2 through 5, if working as intended, will achieve this. The main concern is how often they will avoid or mitigate failures that would otherwise happen: whether fault tolerance will produce enough of a reliability gain to be worth its cost. Studying the effectiveness of fault tolerance serves multiple purposes:

- deciding on adopting a technique: how effective is a certain form of fault tolerance going to be? In particular:
  - is it cost effective when compared to other methods for achieving the same dependability improvement? [9]

[8] In the sense that correct operation of one out of the two channels is sufficient for correct operation of the shut-down function.

[9] For most software engineering methods that promise to increase already high quality, costs are easier to predict than effectiveness. Software fault tolerance is no exception. Costs include development and

  - is it effective as an *extra* precaution, added to the other available methods?
- acceptance of a product by a customer, or licensing of a critical system by a safety regulator: how can one judge the dependability of a specific system built with design fault tolerance, without waiting for the (possibly enormous [3]) length of operational experience needed to assess it by purely statistical methods?
- deciding, once it is accepted that some form of fault tolerance is necessary, which form to apply: which form of redundant design, which measures to promote diversity.

Answers to these questions tend to address separately the two aspects of redundancy in the architecture, and failure diversity. The reliability that can be achieved by a given component-structured fault-tolerant architecture can be assessed via probabilistic models of kinds that are standard in computer reliability engineering [19], e.g. Markov chains. Examples for software fault tolerance are e.g. in [33 , 39] and references therein. The solutions of the models will depend on the values of their parameters, including probabilities of failure of the individual components, and, crucially, the probabilities of common failures between the redundant parts of fault-tolerant components[10]. The main difficulties lie in this latter part.

3.2 Models and empirical evidence

The effectiveness of any fault-tolerant architecture depends crucially on the probability of common failure between its redundant parts. Substantial research effort has been spent on studying this factor in the case of N-version software (summaries are in references [40, 11, 41]). Empirical studies ranged between the two extremes of case studies attempting to approximate realistic development processes, producing few program versions, and statistically controlled experiments made affordable by using student programmers. All experiments showed software fault tolerance to produce some reliability advantage.

An early question was whether "independently developed" software versions could be trusted to fail independently. This would be good news from two viewpoints: to assess the probabilities of joint failures of two or more components, one could just use the products of their individual probabilities of failure, without any special effort; and the reliability gains would be massive, because these probabilities are small. Checking this conjecture was the goal of a famous experiment run by Knight and Leveson [42], in which 27 versions were developed and then run on one million test cases. On average, it turned out that a 2-out-of-3 voted configuration would be about 20 times less likely to fail, per demand, than an individual version, but this was far short of what failure independence would have produced. This experiment established that to claim failure independence in any specific multiple-version system, one would need to find evidence of it for that system, rather than relying on a supposed general law[11].

Further insight was provided by probabilistic models developed by Eckhardt and Lee (*EL model*), and Littlewood and Miller (*LM model*) (and by Hughes for physical failures). Surveys and references are in [13, 43]. These authors considered that the outcome of a development process is uncertain, and studied

run-time costs of redundant components, offset in part by savings on common parts of the development and on verification phases. Development costs for fault-tolerant software are discussed e.g. in [36 , 33 , 37 , 38].

[10] These will appear in the model directly as probabilities of common failure (per unit of time or per demand on the component), or as "coverage factors": an error coverage factor is defined as the conditional probability of a fault-tolerant mechanism (e.g., the checker in Fig. 2) reacting appropriately to an error (i.e., for the checker, detecting the error), given that the error occurred. For the case of error detection by comparison of results, and correction by voting, it matters not only whether two versions fail on the same invocation, but also whether they produce results that the comparator/voter considers equivalent.

[11] This result is still often cited as proving that N-version software is not effective. In fact, diversity improved reliability in this experiment, albeit less than if failures of diverse versions were independent. The only negative result concerning the efficacy of diversity is that one cannot depend on it having *spectacular* effects on dependability. In the practice of safety engineering, positive correlation between failures of redundant components is often assumed, unless there are very cogent arguments for excluding it.

the average effect that should be expected from diverse, independent developments of software versions. These models indicate that if the two development processes are similar, though rigorously independent, the average *pfd* of two-version systems will be greater than the product of the average *pfd*s of the versions, as observed in the Knight and Leveson experiment. An intuitive explanation is that for the builders of diverse versions of a program some demands will presumably be "more difficult" than others, in the sense of being more likely to be demands on which the delivered programs will fail. So, even if diverse versions are built "independently", their failures are more likely to happen on certain demands than on others, which in turn leads to this "average positive correlation". Using versions produced by the same process is actually the worst case: the average reliability of two-version systems can only improve if one manages a project so as to "force" diversity (to make easier for the developers of one version those demands that are more difficult for the developers of the other), provided that this does not reduce the average reliability of each version. The LM model shows that achieving even zero average *pfd* for a two-version system, despite comparatively large *pfd* for the individual versions, cannot be excluded. However, the model cannot practically be used to forecast such a result in a specific case. This is a limitation of the EL and LM models: while they give useful insight into the relationship between design diversity and the reliability of fault-tolerant systems, in particular avoiding fallacies in arguments for failure independence that may otherwise look intuitively correct, direct practical applications to assessing a specific system, or directing the choice of architectures and development methods, are problematic. The reason is that these models rely on an idealized, extremely detailed probabilistic description of the failure processes, whose parameters cannot be directly evaluated in practice. However, my colleagues at City University and I have found the same general modeling approach to be useful for many questions about the effectiveness of diversity in various contexts, including some applications to assessing specific systems[12].

Although diversity proved beneficial in all experiments, and the models give some understanding of the parameters that determine its efficacy, one would want evidence of it being beneficial in actual industrial use. Here, although users appear to have been generally satisfied with the results [40], data are hard to obtain. In particular, the more widely publicized applications are for "ultra-high reliability" applications, in which failures of even one version would be rare and make statistical assessment difficult. It should be noticed that, even if these systems were actually free of design faults, the decision to use fault tolerance as a precaution could still be right: the role of fault tolerance is to protect against faults that *may* end up being present in the delivered system, although unnoticed, and despite the effort spent in avoiding them.

3.3 Other potential advantages and concerns

Fault tolerance against design faults also has some secondary advantages. It usually gives improved on-line error detection and thus supports testing and debugging. The increased confidence this gives may reduce the delay between the completion of a system and the time it can be confidently put into operation, with possibly substantial economic gains.

Fault-tolerant designs also bring some additional concerns: added complexity, which may bring additional design faults, either in additional components (e.g., voters) or in the system design connecting them. This makes "component-structured" approaches attractive, to give designers clear, and separate, views of a system's composition from the viewpoints of "functional" behavior and of fault tolerance. It also usually requires added hardware, and thus higher likelihood of physical faults: the design must provide enough fault tolerance against these to offset their increased probability.

3.4 Adoption of fault tolerance methods against design faults

So long as its cost is moderate, fault tolerance can be seen as a necessary part of reasonable engineering practice. So, some degree of defensive programming is considered a part of all good software practice.

Conveniently, fault tolerance mechanisms can be used to measure their own usefulness, by logging their interventions and the system failures thus prevented. These logs also help diagnosis and repair of faults, and feedback on the development and procurement processes. This feedback would also allow a manager of the software development process to find a reasonable balance between expenditure in

[12] Summaries of recent research are currently maintained on the Web at http://www.csr.city.ac.uk/diversity.

avoiding faults and in fault tolerance, or a system developer to estimate roughly the degree of fault tolerance needed in newly developed systems.

Decisions become far more difficult for highly dependable systems for critical applications. Some safety-critical applications use multiple-version (custom-built) systems or similar high-cost forms of fault tolerance. The cost of developing multiple versions is justified by the high cost of failure and the consequent very low probabilities required. But there may be no statistical knowledge for judging the advantages achieved, because the concern is with very rare, and thus seldom observed, failures, which yet are important because they may lead to serious accidents. Examples include civil aviation: Airbus has used diversity in software and hardware [32], and Boeing in hardware [44]. Other applications include various railway signaling and control systems [40 , 45, 46, 34, 47]. Fault tolerance is applied on top of stringent and expensive precautions for fault avoidance. For neither set of precautions is there a quantified assessment of effectiveness, e.g., in terms of the mean number of dangerous failures avoided per year. Fault tolerance is an added "safety margin" in the operation of already very dependable systems. This is similar to "safety margins" in other areas of engineering. A baseline design is demonstrated to be satisfactory under certain assumptions; the safety margin is a defense against all kinds of unanticipated departures of the real world from the assumptions. It is impossible to quantify this protection with any precision, but it is believed necessary to err, in case, on the side of excessive prudence. These industrial sectors adopted software fault tolerance when first introducing software in safety-critical roles, and facing the need to preserve the good safety record which had previously been achieved with non-software technologies that were thought to be better understood.

Another viewpoint is that fault tolerance may make it possible to obtain reliability levels that would be infeasible by other means, regardless of the accepted cost. For high-quality development practices, there may be an upper bound on the reliability obtainable without fault tolerance, simply because people are fallible, and increasing the resources spent on a project beyond some threshold would only increase overhead and human communication problems, without improving the results. This bound may change with breakthroughs in software engineering, but if it exists, fault tolerance will be a more effective way of using extra effort for system dependability.

There is another situation in which designers cannot improve the dependability of their components as needed, and thus fault tolerance becomes necessary. It is when components are obtained "off the shelf", as is increasingly common. This is arguably the correct way of building most engineered systems, but in the computing sector system developers often have to deal with components that are built to insufficient standards, and lack proper documentation of their dependability. The problem has existed for a longer time for hardware, as system developers could not afford, e.g., to develop their own high-integrity processors, but cannot trust industrial standard processors to be sufficiently free from design faults. So, for instance, both Airbus and Boeing use diverse processors in fault-tolerant, flight-critical computers on their aircraft. As the practice of using off-the-shelf software components even in critical systems becomes more common, so does concern about their dependability. For a system integrator, it will usually be impossible or impractical to modify off-the-shelf components, or to verify them extensively. On the other hand, fault tolerance can be applied without requiring access to the internals of the off-the-shelf item by:

- procuring additional off-the-shelf items with similar functionality, to build an N-version or similar architecture. The difficulty of doing so will vary between applications, being easiest when a "1-out-of-N" structure (as in the example of Fig. 5) is possible: e.g., for complete communication or alarm systems, with a human users "adjudicating" among the alternative results available, or for servers operating in distributed systems with good "fail-stop" properties. While N-version software for custom-built systems is expensive, multiple off-the-shelf software versions will usually be cheaper than even a single custom-made version. Preliminary investigations about both the feasibility and the reliability gain of such designs have produced encouraging results [48, 49];
- or alternatively, by producing "checker" components that, although custom-made, may be simple and cheap enough to produce.

A form of fault tolerance that has become common with off-the-shelf components is that of "wrappers": small components that, for instance, intercept calls to functions that the system designer does not intend to be used, or to functions known to be defective [50 , 51][13], or that act as checkers

[13] These deliver true "fault tolerance", in that they isolate the fault directly, rather than reacting to the errors that the fault causes.

[52]. The concern about the vulnerabilities of off-the-shelf systems is also central to interest in fault tolerance for security (*cf* e.g. references in [53] and [54]).

4. Design Solutions

Software fault tolerance has been applied and studied for a long time. Many proposed design schemes can be found, e.g., in the proceedings of the IEEE conferences FTCS, ISSRE, and DSN. Additional schemes, not documented in the open literature, are used in industry. Useful reference books are [15 , 11 , 12]. Rather than enumerating design variations, this chapter outlines the range of design approaches from which specific designs can be synthesized.

Many mechanisms for tolerating the effects of physical faults will also help to some extent against design faults, but it is important that the possibility of design faults be actually taken into account by system designers. An example documenting this need is the accident during the first launch of the Ariane V rocket [55], in which an exception handler reacted to a simple data error by shutting down operation of a critical processor. This behavior was programmed on the assumption of physical faults only, and would allow an identical back-up processor to continue providing the critical functions. However, this particular data error was due to a system design fault (the designers had kept in Ariane V some software functions designed for Ariane IV that were no longer appropriate). Both the primary and the back-up processors thus became unavailable for their flight-critical function, leading to inappropriate control signals to the rocket and its subsequent destruction. To tolerate design faults, a designer may use the redundant resources that are available for tolerating physical faults, but must take into account types of failure behavior that physical faults would be unlikely to produce.

It is convenient to organize this discussion in three parts: the role of standard, general-purpose fault-tolerant mechanisms; algorithm- and application-specific techniques; and the component-structured schemes that have been devised especially for tolerating component design faults.

4.1 The role of generic fault tolerance techniques

Most fault tolerance techniques for physical faults will also help to tolerate some design faults. For instance, many computer platforms have built-in features for error detection. Hardware will detect attempts to divide a number by 0, or to violate a process's address space; hardware or software checks may check type violations or array overflow at run time. These mechanisms give fine-grained checks: applied frequently during the computation, and on almost every data item. But they will only detect attempts to perform actions that are "illegal" for the abstractions they protect. For instance, an array overflow check will detect a program's attempt to violate the rules for operating on arrays, but cannot detect an attempt to write, within the bounds of the array, a word that has the wrong value according to the program's specification. So, they cannot be expected to give high coverage for design-caused errors. Programs can also be instrumented with "executable assertions" or "run-time checks" on necessary properties of correct executions, which, being based on the specification of the specific programs, can be more effective.

In addition to detecting errors, fault tolerance requires recovery or error masking, and so on. Here again, structuring concepts developed for other forms of fault tolerance are relevant:

- support for exception handling and rules for exception propagation in language implementations;
- primitives for checkpointing and recovery;
- atomic transactions, provided as standard features in components such as database servers.

4.2 Algorithm- and application-specific techniques

Many algorithms lend themselves to specialized forms of fault tolerance. Compared to more generic methods, these may deliver better coverage, lower run-time cost(requirements of CPU time and other resources) or other advantages. On the other hand, their specialized nature means that each new problem requires a specialized, effective and computationally affordable error detection or correction algorithm; for some problems, none may be found. Some algorithms (e.g., for iterative approximation, or heuristic search, or process control) have "natural" fault tolerance properties, in that corruption of part of their data will not prevent the algorithm from converging to the correct solution, possibly with some extra delay. Everything else being equal, a designer would prefer an algorithm that satisfies the specification and has this property, i.e., some degree of fault tolerance at no extra cost, over one that lacks it. The literature on intentionally designed robust algorithms includes a few (imprecisely delimited) categories:

- "self-stabilizing [56, 57] algorithms, guaranteed to reach a state defined as acceptable, starting from any state: such an algorithm can tolerate corruption of its internal data, at the possible cost of extra delays. The literature provides many examples of such algorithms with proofs of their self-stabilizing property, as well as impossibility results, i.e., characteristics that make it impossible to transform an algorithm to achieve self-stabilization;
- "robust data structures", extensively used, e.g., in disk storage. A complex data structure is protected against corruption of its physical implementation. For instance, a linked list may be complemented with additional links, so that the loss of the information describing one link can be detected and, possibly, the list can be reconnected. The amount of redundancy provided determines the maximum numbers of data errors for which detection, or correction, can be guaranteed. Robust data structures are a special case of error detecting and/or correcting *codes*;
- "algorithm-based fault tolerance" techniques (see e.g. [58]), akin to arithmetic codes. For instance, the contents of matrices can be protected by adding redundant lines and columns, such that the augmented matrix, unless corrupted by faults, will satisfy some invariant properties even after processing through standard operations of matrix arithmetic;
- the "complex checkers" studied by Blum and co-authors [59, 60]. These rely on a form of "data diversity". Given a component that has to compute a mathematical function f of its input, such that a known invariant property I links its values on different values of its arguments, a "complex checker" will, when the component is invoked with an argument x, produce additional invocations on arbitrarily chosen argument values x_1, x_2, ... and verify that the invariant $I(f(x), f(x_1), f(x_2), ...)$ holds.

4.3 Component-structured fault tolerance

In contrast with these techniques, most research on software fault tolerance has dealt with "component-structured" fault tolerance, introduced here in section 2, in which diversity is explicitly sought between redundant variants of a component, and/or between a component and checkers of its results. This approach brings some advantages of modularity, in that system design deals with the interfaces of components, not their internal details. System designers can proceed in top-down fashion from the system-level requirements of the system being designed (say, a car, a therapy machine), to choose the levels of decomposition at which they wish to apply fault tolerance, and which configuration gives the required trade-off between, e.g., performance, reliability, safety, cost. Adapting programs so that they can act, e.g., as channels in a multiple-version component may require no special provision at all to the program's source code (this is especially desirable when designing with off-the-shelf components), or limited changes [17] that do not increase much the burden on component programmers. System designers may also choose the technologies for different components in ways that appear to reduce common vulnerabilities, as in the examples cited in discussing Fig. 5; some of the components may be implemented as pure hardware with hard-wired or analog functions, or they may be human operators. Last, these techniques allow "end-to-end" error detection. By situating error detection at the interfaces, the designer can check all possible paths for error propagation between components, and, if desired, avoid the cost of many error detection checks inside the components (at the possible cost of higher error latency).

A specific fault-tolerant design implies a combination of choices with respect to a set of binary or multiple-choice design decisions: it corresponds to a point in a multidimensional "decision space". Many of these decisions imply trade-offs or design difficulties to be solved. Such a list of essential design decisions would include:

- numbers of redundant (diverse or identical) instances of components, and additional components (like checkers). These decisions are often constrained by other design aspects, especially development and run-time cost, and other fault tolerance requirements, and in turn affect the cost and feasibility of most other decisions;
- type of components: e.g., for software, long-lived processes *vs* repetitive short-lived tasks, with or without a permanent state separate from their short-lived program variables, etc;
- error detection methods: run-time checks *vs* comparison among results of diverse versions, or combinations of the two;
- error correction methods for error masking and/or state recovery, which can use backward recovery and retry, or various forms of forward recovery;
- size of fault containment units, for each different method applied;

- allocation of redundant executions to hardware modules, and combinations of redundancy of execution in space (multiple processors) and in time (multiple executions on each processor);
- scheduling of redundant executions: unconditional *vs* upon detection of errors. The latter reduces the cost of error-free computations, at the cost of increased delays for error recovery.

A brief discussion follows of some of the salient issues for designers.

Error detection and confinement

The choice of methods can be guided by rough estimates of development and run-time cost, which are comparatively easy to obtain, and of coverage, which has to be roughly assessed on the basis of the specifications of the components to be made fault-tolerant: e.g., run-time checks will appear desirable if a theoretically perfect check exists that also allows a simple (i.e., cheap and probably reliable) implementation with acceptable run-time cost, while multiple-version comparison will be preferred if no reasonably thorough, simple and cheap check seems possible but apparently very diverse algorithms are available for computing the same function (which removes a likely cause of common failures, although it does not guarantee their absence). In the intermediate cases between these two extreme situations, decisions will depend in part on educated guesses about the component failure modes that can be expected and the effectiveness against them of the alternative mechanisms.

Another issue is the allocation of the error detection and confinement mechanisms between components. In Figures 2-3, the fault-tolerant mechanisms are bundled in the augmented C1 component to prevent the generation of erroneous outputs. Actually, these mechanisms can be shared with, or even completely moved to, the recipients of those outputs (here, C2). This may be helpful for guaranteeing error confinement even if fault tolerance in C1 is made ineffective, e.g., by hardware failure. In terms of project management, it is helpful in that it allows checks in C1 and C2 to be based on the different information available to their designers, and to the modules themselves at run time. For instance, checks in C1 can use data in the internal state of C1, invisible to C2, while checks in C2 can use predicates that must be true about the combination of inputs that C2 receives from C1 and from other components.

Discrepancies between correct computations

In many applications, it is possible for diverse variants to produce results that are different, even if all correct. This may be due, for instance, to different rounding errors in floating-point computations. As an effect of this, two versions that nominally obey the same specification may take opposite decisions when, e.g., comparing a numerical variable against a threshold [61]. Such discrepancies are also possible in replication without diversity, if, for instance, copies of the same software read sensor inputs asynchronously, or if their behavior depends on that of other software in a distributed system [62]. These discrepancies affect various aspects of design. They may cause comparison between version outputs to signal an error, despite all versions being correct. So, voting and comparison (adjudication) algorithms must take into account these allowable discrepancies between correct results. They may also cause the versions to diverge over time so that a voter can no longer find a majority output. In a control system, they may cause dangerous discontinuity in control signals if, for instance, control is switched to a back-up version.

A solution is to require the versions to explicitly reach a consensus at each critical point identified in the specification. This, however, may require specifying in great detail the internal structure and algorithms of the versions, and thus decreasing the degree of diversity achievable. An alternative is to allow such discrepancies to happen. If they are rare enough, they can be treated as if they were actual component failures, using the same mechanisms that deal with the latter.

Error correction and state recovery

It is useful to distinguish between the correction of errors in the *outputs* of components (masking) and in the *internal states* of components (recovery). The latter may involve more data and time than the former and thus require different methods. In any case, erroneous values can be corrected by: backward recovery followed by re-execution of the failed computation (by the same software, or by a different version of the software written to the same specification, as in *recovery blocks* [63, 64]); voting or adjudication (generalized voting, taking into account discrepancies between correct results, the nature of the data to be decided upon, and possibly extra evidence about the dependability of the individual versions: see [16]) among the results from multiple versions; or by kinds of (application-specific) forward recovery other than voting.

Some techniques for the recovery of the internal state can be applied to broad categories of systems and components:

- backward recovery, which may rely on checkpointing (libraries of checkpointing and rollback functions exist for popular programming environments), or “atomic action” capabilities designed into the components and allowing specified operations to be rolled back completely and without side effects (for instance the “atomic transactions” provided by database management systems);
- forward recovery via components “re-learning” the correct state of the computation from an uncorrupted global state (data base, sensor readings), or from voted results of other versions, possibly after termination and restart of the erroneous component;
- forward recovery assisted by one or more other versions of the same component that have correct states. To accomplish this, each version needs to offer an interface for transmitting a copy of its internal state, on demand, or accept such a copy to effect its own recovery. Since the representations of the internal states (e.g., the variables defined in their code) will differ between the diverse versions, the specifications need to prescribe a set of abstract state variables whose values are sufficient for any version to reinitialize its internal state, and a standard format for their transmission between versions. Each version needs to include translation functions between this transmission format and the internal representation of this information [65].

Distribution and scheduling of executions

The components of a fault-tolerant software component (variants, checkers, adjudicators) can be executed in parallel or sequentially, on the same or on separate computers. Separate computers give protection against hardware faults, and the possibility of parallel execution for higher throughput, shorter average response time, and faster correction of version errors. If there are multiple variants, their execution may be unconditional or conditional on error detection. For instance, given three variants of a (stateless) component, an alternative solution to that of Fig. 3 is to run two at every invocation of the component, and only call the third one if the first two produce conflicting results. Resources are saved during error-free execution, in exchange for extra delay when an error is detected. More flexible execution schemes are possible, with various trade-offs between these costs and reliability [66, 67].

Granularity of components or fault containment units

The partitioning of a system into error containment units is constrained by many factors: some components cannot be partitioned because procured “off the shelf”, or because there is no natural way of dividing them into subcomponents with clean interfaces, or because such subdivision would badly reduce performance. Even so, there are trade-offs open to the designer. Choosing smaller units usually improves (decreases) error latency, for a given error coverage provided by the detection mechanisms, and thus reduces error propagation, and the cost of recovery: error-free computations cost more because more checks are run and – usually – more synchronization overhead is incurred; errors cost less. The more frequent checks also help in diagnosing the bug responsible for the detected errors. From the viewpoint of failure diversity, smaller units imply the testing of more intermediate results. This reduces the potential for dissimilarity among diverse variants, which is the main guarantee both of effective error detection through comparison, and of proper recovery (either through re-execution or through voting). On the other hand, reducing diversity reduces the problem of discrepancies between correct outputs from different versions.

Combinations of design decisions

Many combinations of these design decisions are described in the literature [12]. For instance, in the “recovery block” scheme, error detection is via checking; correction of the outputs and state recovery are via rollback and retry by a different variant. Additional variants are only executed on demand, sequentially. Proposed variations include the parallel, unconditional execution of variants in a recovery block-like fashion: error detection via acceptance tests and time-outs (avoiding the problem of inexact voting), but with alternate results available immediately upon detection of an error.

In safety-critical applications, whole control or safety systems are often structured in simple architectures like those in Fig. 4 and Fig. 5. Different degrees of redundancy may be applied to different “levels” in an architecture: for instance, the Boeing 777 “fly-by-wire” computer uses diversity at the hardware level but a single high-level application program; while the Elektra railway system uses diversity at the application level, and non-diverse redundancy at the hardware and operating system level. If the system is inherently complex, it may require more than a uniform application of one scheme to the whole system to be both effective and cost-effective. We should expect that different parts of the system naturally lend themselves to the use of different techniques, including different amounts of redundancy and diversity. When multiple variants are used, the degree of redundancy and diversity may differ for different functions, according to their criticality and to the difficulty of guaranteeing good dependability through fault avoidance. Platform architectures have been developed to support such flexible configurations, e.g. [68, 69].

An example of a hybrid algorithm-specific scheme is the use of *certification trails* [70]. Within an application algorithm, a set of intermediate results is identified that can be checked with coverage close to 100%. Two versions of software are then used, but only one normally needs to compute these intermediate results (called the "certification trail"); if this version then fails, the other version can check whether the "certification trail" is correct, and if it is, it only needs to perform the remaining part of the algorithm.

5. Summary

Fault tolerance against design faults is a necessary part of dependable design. It encompasses many techniques for detecting and correcting errors, and for structuring the redundancy thus added to a system. The structuring concepts are the same that govern fault tolerance against physical faults, and many architectural solutions for redundant design have been published or can be devised from these basic principles, as a function of the dependability requirements and threats. However, the probability of common design-caused failures between redundant components is affected by more complex factors than affect physical failures: the required "diversity" is a less understood set of phenomena than simple isolation from common causes of physical failure. Common sense informed by simple mathematical models, together with practical constraints, will inform choices about how to pursue diversity. However, there is ample room for improving our knowledge about how to drive development so that design faults, if present, are unlikely to cause common failures in redundant components, and thus system failure. This requires more empirical research, supported by improved ties with research on human error.

All empirical evidence is that software fault tolerance, in experiments or in practical use in which failures of the non-fault-tolerant components could be observed, has provided dependability advantages. The open questions concern how the cost-effectiveness of these techniques compares with that of possible alternatives. Predicting the effect of these techniques on a specific system before it is built, as well as on the completed system before it is used, is difficult, as it is for most other techniques against design-caused undependability. However, N-version programming has been studied extensively and a basic mathematical understanding reached.

"Component-structured" software fault tolerance, especially with multiple versions of software, has long been practiced only by developers of highly safety-critical systems. With the trend to use "off-the-shelf" components for many applications, all kinds of software fault tolerance become more attractive for system integrators, and schemes requiring multiple versions of the software may become cost-effective even for systems of limited criticality. However, the many lower cost solutions based on exploiting standard error detection methods, loosely coupled replication without design diversity, and software "rejuvenation" are worth considering as well. An essential factor for dependability is a "fault-tolerant design attitude": the knowledge that design faults are almost certainly present in the components we buy, and therefore the system integrators must make sure that they do not cause high-consequence system failures. Developing appropriate system design solutions within the constraints imposed by using off-the-shelf components, as well as collecting empirical measurements of the efficacy of various fault-tolerant solutions and producing ways for informing the choice of architectures and of redundant components, are important areas for current practical research.

Acknowledgments

This work was supported in part by the U.K. Engineering and Physical Sciences Research Council (EPSRC) through projects DOTS (Diversity with Off-The-Shelf components, grant GR/N23912/01 and DIRC (Interdisciplinary Research Collaboration on the Dependability of computer-based systems, grant GR/N13999/01). The author wishes to thank his colleagues Peter Popov, Peter Mellor and Ilir Gashi for their comments on previous versions of this paper.

References

[1] RISKS, "Forum On Risks To The Public In Computers And Related Systems".

[2] J.-C. Laprie, "Dependability of Computer Systems: from Concepts to Limits,", in Proc. IFIP International Workshop on Dependable Computing and its Applications, Johannesburg, 1998, pp. 108-126.

[3] B. Littlewood and L. Strigini, "Validation of Ultra-High Dependability for Software-based Systems", Communications of the ACM, 36, 1993, pp. 69-80.

[4] B. Littlewood and L. Strigini, "Software Reliability and Dependability: a Roadmap", in A. Finkelstein (Ed.) "The Future of Software Engineering - State of the Art Reports given at the 22nd Int. Conference on Software Engineering, Limerick, June 2000", ACM Press, 2000, pp. 177-188.

[5] L. Chen and A. Avizienis, "On the Implementation of N-Version Programming for Software Fault Tolerance during Program Execution", in Proc. 1st International Computer Software and Applications Conference, COMPSAC 77, New York, 1977, pp. 149-155.

[6] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith and B. Randell, "A Program Structure for Error Detection and Recovery", in Proc. Operating Systems, International Symposium, Rocquenfourt, 1974, pp. 172-187.

[7] A. Avizienis and H. Yutao, "Microprocessor entomology: a taxonomy of design faults in COTS microprocessors", in Proc. 7-th IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-7), San Jose, California, USA, 1999, pp. 3-23.

[8] D. A. Norman, "Commentary: Human error and the design of computer systems", Communications of the ACM, 33, 1990, pp. 4-7.

[9] C. Meadows and J. McLean, "Security and Dependability: Then and Now", in Proc. Workshops on Computer Security, Dependability, and Assurance: From Needs to Solutions, York, England and Washington DC, USA, 1998, pp. 166-170.

[10] Y. Deswarte, L. Blain and J.-C. Fabre, "Intrusion tolerance in distributed systems", in Proc. IEEE Symposium on Research in Security and Privacy, Oakland, CA, USA, 1991, pp. 110-121.

[11] M. R. Lyu (Ed.), "Software Fault Tolerance", Wiley, 1995.

[12] L. Pullum, "Software Fault Tolerance Techniques and Implementation", Artech House, 2001.

[13] B. Littlewood, P. Popov and L. Strigini, "Modelling software design diversity - a review", ACM Computing Surveys, 33, 2001, pp. 177-208.

[14] J.-C. Laprie (Ed.), "Dependability: Basic Concepts and Associated Terminology", Springer-Verlag, 1991.

[15] P. A. Lee and T. Anderson, "Fault Tolerance: Principles and Practice", Wien - New York, Springer-Verlag, 1990.

[16] B. Parhami, "A Paradigm for Adjudication and Data Fusion in Dependable Systems", this volume.

[17] A. Avizienis, P. Gunningberg, J. P. J. Kelly, L. Strigini, P. J. Traverse, K. S. Tso and U. Voges, "The UCLA DEDIX System: A Distributed Testbed for Multiple-Version Software", in Proc. 15th IEEE International Symposium on Fault-Tolerant Computing (FTCS-15), Ann Arbor, Michigan, USA, 1985, pp. 126-134.

[18] T. Anderson and J. C. Knight, "A Framework for Software Fault Tolerance in Real-Time Systems", IEEE Transactions on Software Engineering, SE-9, 1983, pp. 355-364.

[19] D. P. Siewiorek and R. S. Schwartz, "Reliable Computer Systems Design and Evaluation", Bedford, MA, Digital Press, 1998.

[20] D. K. Pradhan (Ed.), "Fault-Tolerant Computing: Theory and Techniques", Prentice-Hall, 1986.

[21] P. E. Ammann and J. C. Knight, "Data Diversity: An Approach to Software Fault Tolerance", IEEE Transactions on Computers, C-37, 1988, pp. 418-425.

[22] Y. Huang, C. Kintala, N. Kolettis and N. D. Fulton, "Software Rejuvenation: Analysis, Module and Applications", in Proc. 25th International Symposium on Fault-Tolerant Computing (FTCS-25), Pasadena, California, U.S.A., 1995, pp. 381-390.

[23] J. Gray, "Why do computers stop and what can be done about it?", in Proc. 5th Symposium on Reliability in Distributed Software and Database Systems (SRDSDS-5), Los Angeles, CA, USA, 1986, pp. 3-12.

[24] I. Lee and R. K. Iyer, "Faults, Symptoms and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System", in Proc. 23rd Int. Conference on Fault-Tolerant Computing (FTCS-23), Toulouse, France, 1993, pp. 20-29.

[25] A. Avizienis, "The Methodology of N-Version Programming", in M. Lyu (Ed.) "Software Fault Tolerance", John Wiley & Sons, 1995, pp. 23-46.
[26] B. Littlewood and L. Strigini, "A discussion of practices for enhancing diversity in software designs", Centre for Software Reliability, City University DISPO project technical report LS-DI-TR-04, 2000, www.csr.city.ac.uk/diversity.
[27] P. Popov, L. Strigini and A. Romanovsky, "Choosing effective methods for design diversity - how to progress from intuition to science", in Proc. SAFECOMP '99, 18th International Conference on Computer Safety, Reliability and Security, Toulouse, France, 1999, pp. 272-285.
[28] F. Saglietti, "A Classification of Software Diversity Degrees Induced by an Analysis of Fault Types to be Tolerated", in Proc. 5th International GI/ITG/GMA Conference on Fault-Tolerant Computing Systems. Tests, Diagnosis, Fault Treatment, Nuernberg, Germany, 1991, pp. 383-95.
[29] B. Littlewood and D. R. Miller, "Conceptual Modelling of Coincident Failures in Multi-Version Software", IEEE Transactions on Software Engineering, SE-15, 1989, pp. 1596-1614.
[30] R. D. Schlichting and F. B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems", ACM Transactions on Computing Systems, 1, 1983, pp. 222-238.
[31] D. Powell, "Failure Mode Assumptions and Assumption Coverage", in Proc. 22nd International Symposium on Fault-Tolerant Computing (FTCS-22), Boston, Massachusetts, USA, 1992, pp. 386-395.
[32] D. Briere and P. Traverse, "Airbus A320/A330/A340 Electrical Flight Controls - A Family Of Fault-Tolerant Systems", in Proc. 23rd International Symposium on Fault-Tolerant Computing (FTCS-23), Toulouse, France, 1993, pp. 616-623.
[33] J.-C. Laprie, J. Arlat, C. Beounes and K. Kanoun, "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures", IEEE Computer, 23, 1990, pp. 39-51.
[34] H. Kantz and C. Koza, "The ELEKTRA Railway Signalling-System: Field Experience with an Actively Replicated System with Diversity", in Proc. 25th IEEE Annual International Symposium on Fault -Tolerant Computing (FTCS-25), Pasadena, California, 1995, pp. 453-458.
[35] J. Reynolds, J. Just, E. Lawson, L. Clough, R. Maglich and K. Levitt, "The Design and Implementation of an Intrusion Tolerant System", in Proc. DSN 2002, International Conference on Dependable Systems and Networks, Washington, D.C., USA, 2002, pp. 285-292.
[36] G. E. Migneault, "The Cost of Software Fault Tolerance", NASA Langley Research Center Technical Memorandum TM-84546, September, 1982,
[37] U. Voges, "Software diversity", Reliability Engineering and System Safety, 43, 1994, pp. 103-110.
[38] K. Kanoun, "Real-World Design Diversity: A Case Study on Cost", IEEE Software, 18, 2001, pp. 29-33.
[39] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico and L. Strigini, "A Contribution to the Evaluation of the Reliability of Iterative-Execution Software", Software Testing, Verification and Reliability, 9, 1999, pp. 145-166.
[40] U. Voges (Ed.), "Software diversity in computerized control systems", Wien, Springer-Verlag, 1988.
[41] M. A. Vouk and D. F. McAllister, "Fault-tolerant software reliability engineering", in M. R. Lyu (Ed.) "Handbook of Software Reliability Engineering", IEEE Computer Society Press and McGraw-Hill, 1995, pp. 567-614.
[42] J. C. Knight and N. G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming", IEEE Transactions on Software Engineering, SE-12, 1986, pp. 96-109.
[43] B. Littlewood, "The impact of diversity upon common mode failures", Reliability Engineering and System Safety, 51, 1996, pp. 101-113.
[44] Y. C. B. Yeh, "Design Considerations in Boeing 777 Fly-By-Wire Computers", in Proc. 3rd IEEE High-Assurance Systems Engineering Symposium (HASE), Washington, DC, USA, 1998, pp. 64-73.
[45] G. Mongardi, "Dependable Computing for Railway Control Systems", in Proc. 3rd IFIP Int. Working Conference on Dependable Computing for Critical Applications (DCCA-3), Mondello, Italy, 1993, pp. 255-277.
[46] D. B. Turner, R. D. Burns and H. Hecht, "Designing micro-based systems for fail-safe travel", IEEE Spectrum, 24, 1987, pp. 58-63.
[47] J. F. Lindeberg, "The Swedish State Railways' Experience with n-version Programmed Systems", in F. Redmill and T. Anderson (Ed.) "Directions in Safety-Critical Systems", Springer-Verlag, 1993, pp. 36-42.
[48] I. Gashi, P. Popov, L. Strigini, "Fault diversity among off-the-shelf SQL database servers", in Proc. 2004 International Conference on Dependable Systems and Networks (DSN 2004), Florence, Italy, 2004, pp. 389-398..

[49] I. Gashi, P. Popov, V. Stankovic and L. Strigini, "On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers", in R. de Lemos, C. Gacek and A. Romanovsky (Ed.) "Architecting Dependable Systems", Springer-Verlag, 2004, pp. 196-220.
[50] J.-C. Fabre, F. Salles, M. R. Moreno and J. Arlat, "Assessment of COTS microkernels by fault injection", in Proc. Seventh IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-7), San Jose, California, USA, 1999, pp. 25-44.
[51] C. Fetzer and Z. Xiao, "HEALERS: A Toolkit for Enhancing the Robustness and Security of Existing Applications", in Proc. DSN 2003, International Conference on Dependable Systems and Networks, San Francisco, U.S.A., 2003, pp. 317-322.
[52] P. Popov, L. Strigini, S. Riddle and A. Romanovsky, "Protective Wrapping of OTS Components", in Proc. 4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction, Toronto, 2001,
[53] B. Littlewood and L. Strigini, "Redundancy and diversity in security", in Proc. ESORICS 2004, 9th European Symposium on Research in Computer Security, Sophia Antipolis, France, 2004, in print.
[54] T. Fraser, L. Badger and M. Feldman, "Hardening COTS Software with Generic Software Wrappers", in Proc. 1999 IEEE Symposium on Security and Privacy, Oakland, CA , USA, 1999, pp. 2-16.
[55] J. L. Lions, "Report by the Inquiry Board on the Ariane 5 Flight 501 Failure", ESA/CNES 19 July, 1996, http://www.esa.int/tidc/Press/Press96/ariane5rep.html.
[56] M. Schneider, "Self-stabilization", ACM Computing Surveys, 25, 1993, pp. 45-67.
[57] S. Dolev, "Self-Stabilization", MIT Press, 2000.
[58] "Special issue on Algorithm-Based Fault Tolerance and Result-Checking", IEEE Transactions on Computers, C-45, 1996,
[59] H. Wasserman and M. Blum, "Software reliability via run-time result-checking", Journal of the ACM, 44, 1997, pp. 826-849.
[60] M. Blum and S. Kannan, "Designing programs that check their work", Journal of the ACM, 42, 1995, pp. 269-291.
[61] S. S. Brilliant, J. C. Knight and N. G. Leveson, "The Consistent Comparison Problem in N-Version Software", IEEE Transactions on Software Engineering, SE-15, 1989, pp. 1481-1485.
[62] S. Poledna, "Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism", Kluwer Academic Publishers, 1996.
[63] B. Randell, "System Structure for Software Fault Tolerance", in Proc. International Conference on Reliable Software, Los Angeles, California, April 1975, (in ACM SIGPLAN Notices, Vol. 10, No. 6, June 1975), 1975, pp. 437-449.
[64] B. Randell and J. Xu, "The Evolution of the Recovery Block Concept", in M. R. Lyu (Ed.) "Software Fault Tolerance", Wiley, 1995, pp. 1-21.
[65] K. S. Tso and A. Avizienis, "Community Error Recovery in N-Version Software: A Design Study with Experimentation", in Proc. 17th International Symposium on Fault-Tolerant Computing (FTCS-17), Pittsburgh, Pennsylvania, USA, 1987, pp. 127-133.
[66] K. H. Kim and H. O. Welch, "Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications", IEEE Transactions on Computers, 38, 1989, pp. 626-636.
[67] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico and J. Xu, "An adaptive approach to achieving hardware and software fault tolerance in a distributed computing environment", Journal of System Architecture, 47, 2002, pp. 763-781.
[68] J. H. Lala and L. S. Alger, "Hardware and Software Fault Tolerance: a Unified Architectural Approach", in Proc. 18th International Symposium on Fault-Tolerant Computing, Tokyo, 1988, pp. 240-245.
[69] D. Powell, J. Arlat, L. Beus-Dukic, A. Wellings, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn and C. Rabejac, "GUARDS: a generic upgradable architecture for real-time dependable systems", IEEE Transactions on Parallel and Distributed Systems, 10, 1999, pp. 580-599.
[70] D. S. Wilson, G. F. Sullivan and G. M. Masson, "Certification of Computational Results", IEEE Transactions on Computers, 44, 1995, pp. 833-847.