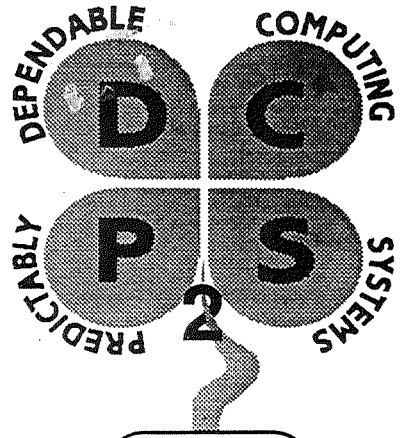
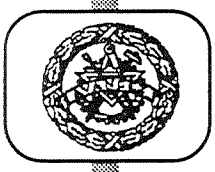
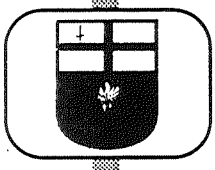


ESPRIT BASIC RESEARCH PROJECT 6362 — PDCS 2

Chalmers Tekniska Hogskola Göteborg



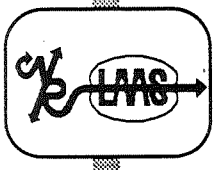
The City University London



CNR Pisa

Université Catholique de Louvain

LAAS-CNRS Toulouse

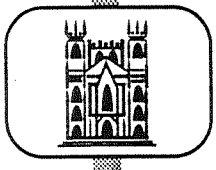


The University of Newcastle upon Tyne

Université Paris-Sud & CNRS

Technische Universität Wien

The University of York



PREDICTABLY DEPENDABLE COMPUTING SYSTEMS

First Year Report

September 1993

Considerations on current research issues in software safety

Lorenzo Strigini

IEI-CNR

Pisa, Italy

The current debate about software safety is the result of basic disagreements, among both researchers and practitioners, about the usefulness of proposed techniques and possibly the very goals to be pursued. The underlying issue is the problem of predicting the safety of a software or software-based system, or the effectiveness of a method for obtaining safety. An argument is made here in favour of quantitative, probabilistic evaluation of safety. Then, some technical problems and research results are reviewed. These include design methods for predictable timing behaviour, the roles of formal methods and of design diversity in achieving safety, and the limits to the levels of software safety that can be assessed or certified.

1 Introduction

Software safety - its achievement, demonstration, certification - is the subject of much debate from both the technical and the regulatory and legal points of view, as the number of operational or planned safety-critical, software-based systems increases. The debate is fueled by basic, unresolved doubts and disagreements, rather than just problems in implementing or prescribing proven methods, supported by a consensus in the scientific community.

A first issue is to what extent safety should be seen as a special concern, separate and different from, e.g., reliability or other aspects of *dependability* [Laprie 1989]. Safety is usually defined as the property of a system of not being subject to specific dangerous failures, such as those that may hurt human beings or damage property. If we define reliability as the property of (or as a measure of the probability of) the system behaving as

required, safety can be defined in exactly the same terms, with reference to a different set of requirements, which forbid a more restricted set of behaviours (those that are dangerous, instead of any incorrect behaviour). On the other hand, an argument is often made that safety is inherently different from reliability, in that it requires different methods for design and verification. This is the actual, important issue of dissension. In particular, it includes the issue of the applicability of reliability models and of quantitative evaluation.

Leveson [Leveson 1991] has strongly argued against reliance on quantitative methods in software safety assessment, as "asking for numbers" will cause engineers to be preoccupied with obtaining the right numbers, possibly by using the wrong hypotheses or methods, rather than with obtaining a really safe system. This problem is particularly felt in cases where the numbers available only support claims for rather modest reliability (or safety); this is the normal case for software. Another objection is that traditional, quantitative reliability-oriented methods lead to "risk allocation" between subsystems, rather than attention to system-level design, where most serious risks can be detected (in the interaction between subsystems rather than inside subsystems) and cured. A blind "risk allocation" attitude may even lead to ignoring cases where subsystems that are individually well-functioning (according to individual specifications which are not properly derived from system design) may interact in dangerous ways. Then, managerial emphasis on a safety attitude, and use of the existing safety engineering culture, seem to improve design safety; attempts to quantitative assessment seem only to deliver numbers that are either too low or untrustworthy.

In my opinion, what these arguments convincingly show is that one must be wary of simplistic analyses, and that nothing can take the place of careful, competent design in producing a good system. However, there are two reasons why quantitative assessment is unavoidable.

In the first place, safety requirements are often, by necessity, stated in quantitative terms: we want the probability of a very serious accident to be lower than some acceptable threshold, and knowing that the designers did their best to avoid such accidents does not tell us how well they succeeded in a particular case. Accepting a product on the basis that methods of "design for safety" were applied, with thorough checks of the design, but without quantitative evaluation, implies a strong trust that these methods always yield products that are safe to the desired level. No such trust seems to be warranted for non-trivial software products [Littlewood and Strigini 1993].

So, if a probabilistic safety assessment is required for a system that includes software in any critical function, quantitative assessment of the software's contribution to risk seems mandatory. Requiring an analysis that convincingly shows the attainment of some quantitative safety level (e.g., probability of mishaps) has the advantage of compelling the analyst to explicitly state the probabilistic model used, and hence the assumptions about which events and cause-effect chains can be neglected, the sensitivity of the model to variations in the parameter values, and such. Such a rigorous argument can be scrutinized

for errors and unwarranted assumptions; informal claims about the methods used in development cannot.

Independently of whether the safety of the delivered system has to undergo quantitative evaluation, quantitative methods are usually a good basis for design decisions. Engineering for safety (like engineering in general) is not just a matter of finding technical solutions that improve individual aspects of the system, but of choosing the right compromise, and selecting between alternative designs which include different compromises. A (software) example [Rushby 1992] is a safety-critical multiple-modular redundant control system (for instance, the flight control system of an inherently unstable aircraft): a very simple design has multiple copies of the software running in a strictly synchronous fashion on separate computers. This solution may be vulnerable to common-mode failures due to electromagnetic interference, which could be alleviated by running the software replicas asynchronously, and to design faults, which could be alleviated by diverse design of the replicas. Both the possible fixes (asynchronous execution and diversity) require a more complicated mechanism for synchronization and voting, hence a higher chance of subtle errors and more overhead (hence possibly more hardware and lower reliability). No one of these alternative designs is then obviously better than another one: any comparison will show pros and cons of switching from one to another. The only rational way to choose the safest among these alternative designs is to model the effects of the design choice on the resulting system safety, and compare them. This quantitative assessment may be difficult and possibly not provide clear support for a decision, but a designer who did not even attempt it would be compelled to trust intuition or luck to avoid choosing a demonstrably inferior design.

These issues of validation and of providing a rational basis for design decisions are central to the topics treated in the rest of this paper. Section 2 discusses real-time design, as an example of the problem of design trade-offs between predictable behaviour and efficiency in software systems, and their effects on dependability. Section 3 deals with the role of formal methods in achieving software safety. Section 4 considers design diversity for tolerating the effects of software (design) faults. Section 5 returns to the issue of quantitative evaluation and its limits for systems with very stringent safety requirements, and Section 6 contains some conclusions.

2 Predictable behaviour and real-time issues

In many computer systems, "real time" requirements are safety-critical: the computer system may cause an unsafe state both by delivering a result with a wrong value and by failing to deliver a correct one on time. For the purpose of this discussion, I shall call "real-time" systems (the term "hard real-time" is sometimes used) those where *guaranteed response times* are a more important concern than *throughput.* Whatever the *measure* of the response times required, obtaining a strong enough *guarantee* that the timing constraints are met is a major design concern. In practice, whether a given software product is treated as a

"real-time" project, or not, depends on the difficulty of achieving such guarantees and the cost or safety implications of its response times for the system in which it is to be used.

Real time requirements pose a number of problems:

- the execution and response times of programs must be taken into account in deciding whether the minimal requirements are satisfied. This implies that any mathematical language used for discussing system requirements and their satisfaction must include timing concepts; some results in this field will be cited later;

- in building real-time systems, much of what the computer science community has learned about ways to improve efficiency (e.g. by clever ways of managing concurrency: from dynamic scheduling to interrupts) must be considered under the guise of a danger to predictable timing behaviour.

It has long been known that so-called "general-purpose" operating systems (really, operating systems for time-sharing systems) are not a good basis for developing operating systems for hard-real-time applications. This is not simply because they may be slow in performing important basic operations, but because their response times are difficult to forecast. The time when a task is scheduled depends on when the asynchronous events that demand attention take place with respect to the inner working of the machine: which task is executing when an interrupt arrives, in which order messages are enqueued, and so on. The end result is that such operating systems, while allowing efficient use of the hardware on average, do not offer a sufficient guarantee of acceptable response times: their internal workings are just too difficult to analyze for their behaviour to be predictable. So, "hard" real-time systems are often built with only skeletal operating systems and with static scheduling, to decrease overheads and improve predictability. However, some authors claim [Jensen and Northcutt 1990] that a "static, deterministic mindset" must be substituted by sophisticated dynamic strategies, trying to optimize response time according to application-specific policies, to deal with complex, rapidly changing application environments.

So, a designer has two contrasting goals: ease of validation (that is, of forecasting in detail the behaviour of the software in operation), and efficiency in using the available resources to cope with the demands made on the software. Emphasis on the former goal leads [Xu and Parnas 1991] towards a "static" design approach, which tightly constrains the possible execution histories of a system to guarantee predictable behaviour. Emphasis on efficiency leads to a very dynamic approach, which builds into the system flexible mechanisms for scheduling execution at run-time, to optimize the response to each particular load condition encountered, using the computing power that is available (as a side note, dynamic management implies a certain overhead, but this may be offset by the better use of resources by the application tasks) . A cognate, useful distinction is that between "event-triggered" and "time-triggered" systems [Kopetz and Kim 1990]. The former are based on the idea of reacting to events as they take place in the external world, for instance by reacting to externally-generated interrupts; the latter, closer to "static" design approaches, on

periodically observing the state of the external world and starting actions based on these observed states (of course, event-driven behaviour can be emulated by a time-triggered system, and the reaction times are determined by the "sampling frequency" chosen). Most existing designs are intermediate solutions, but it is interesting to consider the effects of shifting emphasis between the two extremes.

An example of a "static" and "time-driven" approach to real-time design is the MARS architecture and design methodology [Kopetz et al. 1989]. There, the internal operation of the software is predictable with a high level of detail: tasks are statically scheduled, and run on computing nodes that are duplicated for fault tolerance. The nodes are connected through a (redundant) network managed by a TDMA protocol, with preallocated time slots for the various tasks. The whole system is strictly "time-driven" (a clock synchronization protocol ensures that all nodes see the same, global time base), rather than "event driven".

The static approach guarantees that no foreseen load of urgent events (including the failures of individual system components) may cause the system to miss a deadline. The term "foreseen" here implies that the timing of the events that the system must be able to cope with is stipulated in the requirements: the designer is given bounds on the rate of change in the external world, and then makes sure that the tasks are scheduled frequently enough to guarantee timely reactions. The simple structure of the system has many advantages: tasks can be designed expecting a fixed sequence of interactions with other tasks, and there is certainty that individually "correct" software modules will not fail just because of unforeseen snags in their interaction. This approach to design accepts a potential waste of resources in return for ease of convincingly validating the system. The system may miss a deadline if the environment "violates its side of the contract", by producing demands more often than stipulated. Finding the probability of this event is part of the analysis of the application environment: it is not some detail inside the implementation of the computer system, which will unexpectedly strike the application engineer. From the point of view of design and verification, time can be allocated to the different activities beforehand, with precise requirements on what each task must do and within which deadlines, and the response to any external event can be forecast by following its effects through the pre-scheduled interactions among tasks.

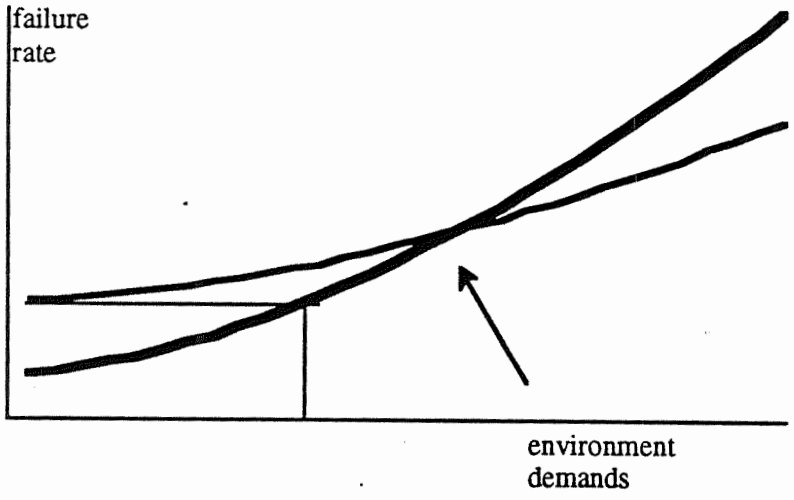
For applications with stringent safety requirements, the method of choice must now be very close to this extreme. It allows convincing demonstrations that a given application design satisfies given timing requirements, and that the simple run-time support mechanisms required will perform as intended. As technology provides cheaper computing power, static designs will become applicable to more demanding applications. However, two arguments apply in favour of more dynamic approaches. One is in terms of obtaining better utility than the static approach, by processing less time-critical tasks in the amount of spare time that the static approach would waste. An important line of research (e.g. in [Stankovic and Ramamritham 1989]) is to explore compromise solutions, e.g., using part of the resources for static, guaranteed-deadline scheduling and the rest for best-effort, run-time scheduling. The problems with following this argument include: determining which load level (if any)

is deterministically guaranteed to be tolerated; understanding (in probabilistic terms) the behaviour of the system under higher loads; validating the more complex run-time support employed; and the major problem of preventing errors in lower-criticality tasks from upsetting the working of the more critical tasks, and of validating the prevention measures.

A second argument for dynamic approaches applies where the static approach would be unfeasible in terms of expenditure, weight or power budgets, or such limitations. An application environments might submit the computer system to a variety of peak loads, characterized by different combinations of tasks, with varying demands on resources by the different tasks. Often-quoted examples are air defence (where the enemy can increase and vary the "environmental demands" almost at will), and autonomous robots and vehicles in complex environments. It may happen that the designer cannot be guaranteed bounds on demands such as to allow a static design within the given constraints, although the application still dictates timing requirements for individual tasks. Guaranteeing to each task its worst-case allotment of resources, even at times when they are not needed, would require more computational power than can be provided. In such conditions, attempting a static design would waste computing power in trying to preserve a delusion of predictability, as the environment could at any time violate the necessarily simplistic assumptions on which the static design is based. Then, the argument goes, it is better to use the available resources to try and cope with demands as they arise, aiming for efficient use of resources and graceful degradation (with preservation of the more critical functions) under overload conditions. This leads to employing a "dynamic" design approach. A shift in requirements has thus taken place: rather than requiring an extremely good guarantee of correct operation under a specified peak load, a component of "best effort" requirements is introduced, e.g. a certain level of guarantee at a "nominal peak load" plus evidence of good average behaviour and/or graceful degradation at higher loads. The arguments about the safety effects of the computer system are necessarily of a probabilistic kind, and the resulting safety guarantee much weaker than would be given by the (unfeasible) static design. The application owner may well prefer to employ a computer system with this level of dependability, rather than not employing any.

The important design problem is how to choose the right approach for a specific application. How does the dependability obtainable with more static and, respectively, more dynamic designs decrease as the environment becomes more complex or unpredictable? Rigorous answers are only possible if the question is stated in rigorous terms, within a given application problem and a set of design constraints, and with clearly defined measures for "complexity" and "dependability" in that application. However, generic *intuitive* expectations are sketched in the figure below. The abscissae represent an indicator of application "difficulty" - e.g. variability of load, or peak load, in terms of frequency of events that require a response; the vertical axis represents a system failure rate, or some other cost measure of interest. The thick curve represents a conservative "static" design approach; the thin one, a more "dynamic" approach. It should be stressed that this picture only describes intuitive expectations: evaluating real designs would only yield scattered points, not regular curves, and a different graph would be needed for each application

problem and set of design constraints. Two important aspects of these expectations are shown : 1) whatever the design method, the bound that can be guaranteed on failure rate gets worse as the environment becomes more demanding; 2) the two curves cross at some point. For moderately demanding environments (at the left of the cross-over point), the conservative approach allows one to reach lower failure rates; indeed, for required failure rates lower than a certain threshold (horizontal line in the graph), it is the only viable approach: a dynamic design is handicapped by the inherent higher difficulty of validating it. As the application becomes more demanding (more variable demands), a static system becomes more likely to fail to respond, so the failure rate gets worse. The dynamic approach also suffers from the more demanding environment, but it may cope better, through higher efficiency and graceful degradation. It has a better chance of responding properly, as it will reorganize its internal priorities to satisfy the most urgent demands. However, the lower bound on the failure rate that can be demonstrated will degrade as the environment becomes more demanding, for any given level of resources, regardless of which design deals best with that level of demand.



The practical questions of interest for a designer or evaluator are: given an application environment and resource constraints, what are the safety levels achievable with different designs? which of the feasible approaches will offer higher safety, and will this be enough? Rigorous analysis of this kind is necessary for good design decisions. Some experience with it would also improve our rules of thumb (for each given application environment) about the degree of safety that can be achieved in computer applications, and where the cross-over point is (if any), at which the conservative approach no longer pays off.

This allocation of risks between the implementation of the computer system and the evaluation of its operating conditions can be described in terms of *assumption coverage* [Powell 1992]. Restrictive assumptions about the operating conditions allow a simpler, hence more reliable design, but the "coverage" of the assumptions (their probability of being verified) must be considered in evaluating the resulting dependability of the system. This approach to design and verification may be more effective than trying to design for all

foreseeable (although unlikely) circumstances, but it is not universally applicable. [Powell 1992] applies the concept of assumption coverage to fault-tolerant system design, where the simplifying assumptions refer to the possible component failures. Here, with proper modelling of the designed system, it is often possible to estimate the cross-over point (probability of the "unlikely" failure modes) where a more complete (and complex) fault-tolerant design becomes better than a simpler one. At the other extreme of the spectrum, where very little modelling seems now possible, there is the case of the application of "expert systems" to demanding critical applications (e.g., controlling unmanned, autonomous vehicles, or managing complex emergencies [Abbott 1990]), where conventionally designed (and more predictable) systems are considered unable to provide the required, adaptable behaviour. Again, it may well be that the application is indeed too demanding for a conservative design to cope; however, the more flexible design cannot be expected to deliver the same level of reliability or safety as the conservative design does in applications where it were appropriate.

These issues become increasingly important as computer applications are planned which tax the more conservative design methods beyond their limits. An innovative design may satisfy its "functional" requirements, but without the level of safety which is customary for conservative designs (where they are applicable). Again, guidance for choosing the appropriate design, and for deciding whether a project's safety goals can be attained at all, can only be provided by a quantitative analysis of both the application environment and the computer system.

3 Formal methods for safety

In the computer science community, the term "formal methods" is generally used with a restricted meaning, designating the application of mathematical logic to the specification and verification of designs. This is a currently a very active research field; I shall make a few remarks about the use of such methods, and point out developments that seem to be of special interest for software safety.

Much of the on-going work on these methods aims at building systems which are correct with respect to their specifications. This, if feasible, could of course achieve safety, if safety requirements were completely and correctly represented in the specifications. Even though this is not necessarily the case, work towards the development of "provable" software has an important role, both in the pursuit of reliability and in that of safety.

Besides achieving safety by avoiding design errors (we might call this "safety through reliability"), "formal" software engineering methods have other safety-specific uses. Mathematical formalism can be used both for deductions (and computations), and as a help in clarifying and checking intuitive reasoning. In a stylized view of the software development process, which proceeds from the initial statement of requirements, by repeated transformations through specifications to programs and finally executable code, the deductive use of formal methods is mostly in the intermediate phases: for instance,

programs can be proven to be consistent with a top-level formal specification via a series of intermediate proofs of consistency between different levels of refinement. But an important role of formalism is also in stating requirements. A formal specification of requirements can be analyzed (is it consistent? what can be deduced from it? which behaviours does it forbid?) and its consequences checked against intuitive requirements. This is not a completely algorithmic process (like checking a proof of correctness), but it is a powerful aid for debugging requirements.

This is also true for safety requirements. Formal, top-level safety requirements for an embedded computer system can be compared against reality and intuitive requirements. It is also possible, in principle, to describe the overall safety requirements for a controlled system, and physical laws governing its behaviour, in the same formal language used for the software safety requirements. Then, the latter two sets of properties should be sufficient to prove the satisfaction of the overall safety requirements: although this is no proof that the system is safe (this is subject to the completeness with which relevant aspects of reality have been included in the formal descriptions), it can be a powerful tool for debugging the requirements themselves. It should be noticed that - conceptually - such a procedure is hardly novel: in all those fields of engineering where the relevant behaviour of systems can be described by continuous mathematics, such "proofs of correctness" are the norm. Analysing the design of a concrete structure, or that of a radio receiver, is precisely that, and suffers from the same pitfalls: if, e.g., the electronic design of the receiver is checked, but its thermal design is not, the receiver might fail catastrophically, even if its design was "formally proven".

Exercises in "formal" requirement capture can now be found in the literature. An example [Ravn and Rischel 1991] concerns a ship's auto pilot. Using a notation called *duration calculus* (a mathematical notation that can describe timing issues is a prerequisite for many practical applications), it is shown that the general requirements for the controlled system (e.g., the deviation of the ship from its course must be bounded), the engineering assumptions on the controlled system (e.g., the possible rate of drift of the ship), and the control strategy chosen for implementation can all be described in the same language, so that the control strategy may be shown to satisfy the requirements, under the given physical assumptions. The method can be extended to the requirements of components in the control systems [Hansen et al. 1991].

This approach helps in clarifying the reasoning that leads to accepting a specification for the software, making evident the simplifications made in formalizing the requirements for the controlled system. It may be questioned how practical this kind of notation and proof procedures would be in the case of more complex application problems, a question that only experience can answer. An interesting application, however, is possible for those systems where separate safety requirements can be stated that are much simpler than the complete requirements. It would be possible to reason in terms of safety requirements only, and show that a subset of the software specifications (perhaps insufficient for defining and verifying "correct" behaviour) satisfies the safety requirements on the

controlled system. This procedure could then be extended to stating safety requirements for software components, and proving that these, together, satisfy the safety requirements for the software as a whole. Module-level "safety" requirements could be used both in the verification phase and to derive run-time checks for monitoring the software execution. Applying formal reasoning to a restricted set of safety specifications may be simpler than applying it to the complete behavioural specifications, and less likely to repeat conceptual errors made in implementing the system to satisfy the latter.

An example of formalized safety requirements (using the formalism of "Timed History Logic") with explicit separation from the "mission requirements" is in [Saeed et al. 1991].

The use of formal notations when defining a top-level software specification, which must capture the relevant aspects of the "real" requirements, is subject to a practical, non-mathematical problem: how easy is it for the domain specialist to produce or check the formal specification? The effectiveness of both the debugging of requirements and the subsequent, formalizable development phases is affected by how likely errors are: how easy the chosen notations and methods are for the developers to use, how dependable the support tools are - if any. The tools can be improved, to reach current "best practice" level; as for the user-friendliness of the methods themselves, only experience will help. Indeed, different notations are likely to be developed for different application environments.

In the later steps of the software development process, formal methods are used for proving consistency between levels of refinements. The limit of their application is towards the "lower" end of the process: compilation and execution. Research is under way on these aspects, mostly in the direction of "safety through reliability", e.g. for formally provable hardware designs [Brock and Hunt 1992] and compilers. However, the production of compilers, processors and operating systems which are formally verified in all their important aspects is still a research problem.

Regarding the acceptance of these methods in industry, it is sometimes claimed that an emphasis on formalism in software development produces visible improvements not only in dependability but also in productivity, for software of good commercial quality. This would be a powerful incentive for the use of formal methods: this kind of advantage is comparatively easy for a company to verify in the short term, after it commits to the use of such methods. However, no strong evidence has been provided. An interesting experience is under way in the ESPRIT LaCoS (Large Scale Correct Systems Using Formal Methods) project. This project is applying the RAISE method [Dandanell 1991] for software development (including a specification language, tools for managing the refinement of the specification, and translators from detailed specifications to high-level language code) in the development of eight industrial software projects, some of them for critical applications (subway train control, monitoring and control for mines). A different flavour of "formalism", as pointed out by Rushby [Rushby 1992], requires machine-assisted proofs of correctness, usually restricted to a critical "hard core" in a system, like the voting algorithm of a modular-redundant system. In this latter form, the adoption of

formal methods for defining and proving limited "correctness" properties seems to imply a reduced investment (mostly tools and training of a few specialists, rather than vast changes in a development organization). Its advantages would be obvious in an organization used to experiencing serious design faults in its software; for an organization with a very good record, instead, these advantages, while obvious in terms of added confidence, would be very difficult to quantify, as only long-term statistics could show whether improved safety is produced. When it comes to very stringent safety requirements, evaluating the effect of the residual faults in "formally" developed software (e.g. because of flaws in the requirements, or mechanical errors in theorem-proving tools) is as difficult as for any other software, as the difficulty lies in the lack of statistical data. This will be the topic of Section 5.

4 The role of design diversity

Design faults are the main obstacle to producing highly reliable (or highly safe) computer systems. The problem is more widely recognized for software than for hardware, although, with highly complex hardware or very high dependability requirements, design faults are going to be an overwhelming concern there as well. No method is known for ensuring that non-trivial software products are fault-free when delivered. The only protection against residual design faults in operational software is redundancy with design diversity: adding to the "base" software, needed for accomplishing a task, additional components with the purpose of detecting and taking remedial action for its errors. Of course, diversity between the additional components and the base software is a necessary condition for fault tolerance.

With the term "design diversity" I am thus designating a wide spectrum of methods, which in software ("software fault tolerance") include multiple version programming, monitor programs, recovery blocks, back-up "degraded mode" programs, and so on [Anderson and Lee 1990]. Design diversity is an accepted principle for non-software, safety-critical systems. In software, it is currently applied in particular in the aerospace and transportation industries [Voges 1988]. There are known problems in its application: e.g., with multiple version software, difficulties with synchronization, "inexact voting" (as different variants of the software may produce slightly different results), "consistent comparison" [Brilliant et al. 1989] (as those slight differences may cause inconsistent branching in the diverse variants), recovery of long-term status information; with run-time acceptance tests, the problem of designing cost-effective tests. However, most such difficulties are understood in theory, and a designer can have a clear picture of the difficulties and possible solutions in any individual design problem [Strigini 1990].

Software design diversity can be used to achieve safety in a "reliability-oriented" fashion (masking failures to guarantee continuous, correct operation, where this is safety-critical) but also in a direct, "safety-oriented" fashion (detecting, and reacting to, hazardous states). For instance, the execution of multiple, diverse variants of a software system can be used to mask failures (by voting the results), or just to detect failures; a monitor program can be

designed to run sanity or safety checks on the behaviour of a main control program, without mimicking its behaviour. The "safety-oriented" use of software design diversity has the advantage of not requiring that the software added for redundancy (monitoring software) reproduce, even approximately, the same functions of the monitored software: this gives more confidence that the two do not suffer from common design faults (a remarkable example of this use is in [Theuretzbacher 1986]). For similar reasons (with the addition of the easier analyzability of non-software systems), it can be argued that "the best place to provide software-fault tolerance is outside the software" [Parnas 1989], by making the system where the software is embedded safe against software failures. This implies the use of simpler, non-software safety systems, or "inherently safe" system designs that do not require active control to satisfy safety constraints. However, "inherently safe" designs are sometimes precluded by the specifications, as e.g. in inherently unstable aircraft, and reliable non-software components for add-on safety mechanisms may not be available. There is therefore a range of applications where software diversity is a natural requirement, although it may not restore the level of safety of, for instance, a conservatively designed aircraft. At the same time, run-time sanity checks in the software (e.g. on the data crossing module interfaces, and on data structure consistency) are usually considered a standard, low-cost precaution. So, some degree of software design diversity has a role in any safety-critical system, and a major role in some of them.

The main problem with design diversity is that, although we know it to be the only remedy against residual design faults in an operational system, we do not usually know how effective it is. Therefore it is also difficult to establish sound rules for its application. Design diversity is best seen as a "safety factor": it boosts confidence, but it is difficult to know how much safety it buys. This problem, incidentally, is common to most safety factors; in fact, design diversity may have received more theoretical attention than other design-safety precautions, and some understanding of it has been gained. The practical limit to the use of these results is in the lack of statistical data to which they can be applied.

A method for the systematic application of software design diversity would ideally provide guidance for two phases of design: producing the redundant software components with the desired probabilistic behaviour, and composing them to obtain the desired system characteristics. Rational criteria for this second part are documented, e.g., in [Laprie et al. 1990]. A first criterion for the rational design of fault-tolerant systems is to specify the types and numbers of component failure to be tolerated, and then select a design that would tolerate them if the fault tolerance mechanisms were 100 % effective. The quoted paper offers a classification of redundant architectures meant for tolerating both hardware and software faults answering questions like: "if my requirement is that the system must operate safely despite up to one software failure and one hardware failure, which configurations can I use"? The next task is to estimate the probability that the redundant design will fail catastrophically (or that it will perform correctly), to check that the chosen configuration is appropriate. Models exist for this task, and they are useful at least in that they identify the factors influencing the effectiveness of design diversity. For instance, they confirm and

quantify the intuitive result that the probability of failure for a redundant component is very sensitive to correlated failures between diverse versions of the software.

A quantitative approach also offers theoretical guidance in the design of an *adjudication* (or generalized voter) function, to make the best use of the multiple results delivered by redundant variants of a component. Many alternative "voter" designs exist, based on plausible arguments, and differing in details like the majority required for "approving" a result, or the algorithm for excluding unreasonable results before a vote. Until recently, no rigorous comparison was provided of their respective merits. It turns out [Di Giandomenico and Strigini 1990] that stating the problem in simple probabilistic terms points the way to an optimal solution: an adjudicator which will provide the most reliable (or the safest, depending on what is sought) result, given the (probabilistic) characteristics of the redundant variants.

These probabilistic methods have their limit in the practical difficulty of obtaining the parameters of interest regarding the behaviour of the redundant components. For the very reliable systems where design diversity is now typically employed, the relevant probabilities are too small for practical measurement. In such cases, these methods give useful qualitative insight for designers, and they may be useful for proving that a system falls short of its requirements, but not that it satisfies them.

When it comes to guiding the design of the redundant software components towards obtaining the required probabilistic behaviour, no proven technique exists. We do not even know how to organize software development toward a goal like a stated reliability, except through massive overkill, much less towards more elusive goals like low correlation between failures of two components. The existing theoretical models of correlated failures in redundant software [Eckhardt and Lee 1985; Kersken and Saglietti 1992; Littlewood and Miller 1989] do describe the phenomenon at a macroscopic level, but do not help in the design phase. Attempts to define useful "diversity metrics" have not, so far, succeeded. Much more practical experience and statistics are needed for an improvement in this area. Past [Voges 1988] and current [Smith et al. 1991] development exercises have developed software for realistic critical applications to produce both development experience and limited statistics on the behaviour of these systems. The accumulation of such data is of fundamental importance for progress in this field; however, only massive amounts of data could support general design guidelines tied to safety levels for redundant software components.

Another way to improve the practice of design diversity would be to define for the software component developer what kind of diversity should be obtained, in terms which are intuitively right and which can be verified in a non-statistical way. Attempts have been made recently to formally [Privara 1991] define both functional equivalence and diversity among formal (algebraic) specifications. If this approach proved viable in practice, the refinement of high-level specifications could be oriented towards obtaining lower level specifications that can be proved to be "formally diverse". This is not, it must be stressed,

a guarantee of diverse failure behaviour of the implemented programs, but rather a way of clarifying what is meant by the requirement of "diversity": once formalized, its consequences can be rigorously scrutinized and compared with the expectations held on the corresponding intuitive requirements, and its satisfaction can be checked.

Summarizing, at the moment there is good understanding of *how* systems employing diverse redundancy can be built, of criteria for choosing their configuration, and of methods for modelling them. The main open problem is the collection of data for evaluating any individual system, and hence also for deciding in which systems and in which form to use these methods.

5 Assessment: the problem of ultra-high requirements

For the acceptance of a safety-critical system, a rigorous and convincing demonstration that the safety requirements have been met should be an obvious prerequisite. However, safety requirements are sometimes so stringent as to preclude any such demonstration, at least for software systems, as extremely low failure probabilities are required. A frequently quoted example is the requirement of a failure probability of less than 10^{-9} per hour for highly critical avionic (non-software) systems. Experience with software teaches us that the design faults that routinely escape design reviews and testing often produce failures with much higher (cumulative) probability. Statistical evidence seems therefore to be the only acceptable argument for claiming such high levels of dependability. This consideration applies to failures with respect to safety specifications just as to failures with respect to all other specifications. The only difference is that in properly designed systems (where "system" means both the software and other components) the former are less likely than the latter: this does not alleviate the problems discussed here.

Research in the statistical evaluation of software and hardware reliability is progressing and offering more practical methods. For instance, in the field of reliability growth modelling there are new methods [Brocklehurst et al. 1990] for improving the trustworthiness of predictions, and methods [Laprie et al. 1991] for modelling the reliability and availability growth of a system, using the corresponding knowledge about its components. However, evaluation based on testing is limited by the amount of testing we are willing or able to perform. To gain confidence in such low probability of failures as 10^{-9} per hour, a system must be observed for an inordinately long time. As shown in [Littlewood and Strigini 1993], for instance, under reasonable assumptions, observing failure-free operation of the software for any length of time would only, by itself, give a 50 % probability of it staying failure-free for an equal length of time in the future. Extrapolating the reliability growth observed during debugging (assuming enough failures are observed to provide some confidence in the procedure) does not offer better results. In fact, a practical demonstration that the requirement has been met can only be reached after the system has been in operation for a very long time without failures; a system could even reach the end of its operational life (without failures) before any confidence could be built either that the requirement was satisfied or that it was not. Of course, failures in operation could provide

convincing evidence that a requirement was not satisfied. This alone could justify the statement of probabilistic safety requirements, as a guarantee for a customer or user against a system provider. However, if the system providers do not have means for predicting the relevant probabilities, their willingness or not to market at such conditions would remain more an indication of self-confidence than of accomplishment.

Three ways out of this problem are usually taken:

1. establishing more modest requirements for the software, i.e., accepting higher risks from the software. This may mean that the whole system (plant, aircraft) including the software is allowed to be less safe, or that its design is changed, to keep the same levels of safety with less dependable software: the role of computers is changed, so that system safety depends less on the software. This is a solution, but not a general one, in that some systems are specified from the beginning in such a way that complex software must be present in a safety-critical role;

2. refusing to state the requirements in probabilistic terms, as done, for instance, in the DO178A guidelines [RTCA/EuroCAE 1985] regarding avionic software, which explicitly refuse to state probabilistic requirements on software dependability. This attitude has stirred controversy in the software community, as software may be an essential part of equipment for which a probabilistic requirement (like the above-mentioned 10^{-9} failure probability per hour) may be stated. In general, this solution only works when either system safety does not rely on software at all (an extreme version of solution 1), or the acceptance of the whole system is based on non-probabilistic arguments. Then, the design of the software can be considered satisfactory based on considerations of good design and verification practice - the only problem being that this is much less well defined for software than for most physical components. However, as pointed out in the Introduction, many important systems must undergo probabilistic risks assessment, so that this solution is not applicable;

3. if both the previous solutions are inapplicable, one must attempt to improve the predictions of the (probabilistic) failure behaviour of the software. Apart from spending more money on testing, this may include making predictions on a different basis than statistical testing alone. After analysing the software to show its "correctness", considering the development methods used, the quality of the development organization, and so on, a probability figure must be produced. This result can take two forms: a claim that the software is totally correct (with respect either to its complete specifications, or to safety specifications), or a claim that it looks good enough for the purpose (it has a very low probability of failure during operation). Currently, both kinds of claims seem difficult to support. Correctness can be demonstrated either through formal proofs, based on a formalization of the "real" requirements, or exhaustive, informal analysis, subject to the analyst's perception of completeness: both procedures involve an unknown, but probably too

high, probability of human error [Barwise 1989]. Statistical arguments have to depend on the existing scant data on the performance of development methods or organizations[3].

So, the mandatory practical solution is, in the short term, solution 1: defining the role of software so that the dependability required of it can be convincingly validated. Limited improvements in the claims that can be accepted for software can be expected from better methods for testing and for the analysis of statistical evidence, and will make solution 1 more acceptable, by raising the limits on the claims that can be made based on feasible amounts of testing. The other way forward is in combining all the available sources of data to improve on the degree of confidence allowed by each of them. The test results, the assessment of the development organization, the results of the design reviews and the proofs performed all contribute to support an argument for assurance. This is now done, informally, whenever disparate arguments are offered to support the case for the acceptance of a product. A more rigorous, mathematical method is needed, if the process is to be usable in practice with the possibility of rigorous checks. This is certainly one of the important challenges for research.

Last, there is a basic problem in all quantification of human error: evaluating the probabilities of errors in an activity A - say, producing software - is another activity, B, subject to error in its turn. There is no end to this series. However, if activity B is simpler than activity A, we may be more confident in its results. An argument based on simplicity is basic to all attempts to reason probabilistically about human error; it would be useful to give it a better basis, at least by collecting statistics on design faults in simple systems, and modelling faults in their analysis.

6 Conclusions

Most research in information technology may yield important results for safety-critical systems. However, this discussion has been centered on issues that are more directly related to safety concerns. The open issues discussed can be summarized as follows.

Designing software-based systems for ease of validation implies a conservative approach, which limits the ability to tackle complex application environments. An example is in the design of hard real-time systems. Failures may be made more likely both by a conservative, static design, which is occasionally unable to cope with a demanding environment, and by a design which attains responsiveness through a complex internal organization. Recognizing these different dangers is a first step towards mathematically analyzing them and thus making design decisions on a rational basis.

Regarding "formal" methods, a very active research field, I have noticed how some current results and on-going research have a potential for direct "safety-oriented" use, besides their often noticed role in proving the "correctness" of a piece of software with respect to a

formal specification. Well developed, tested (at least in laboratory examples) methods exist: the real questions concern their practical applicability and effectiveness.

Regarding design diversity for tolerating design faults, I have pointed out its important role in pursuing software reliability or safety, and the availability of analysis methods for guiding its application. Design diversity is systematically used in some safety-critical applications, and is in these areas a mature technique. The problem is how its effectiveness can be judged, in those applications, with very stringent safety requirements, where it is usually employed.

The discussion of "ultra-high" dependability introduces cases where current advanced techniques for dependable design and for evaluation do not provide a solution. Useful research results exist in the form of statistical methods to make the best of the data that it is feasible to collect. An open field is that of rigorous, systematic methods for basing decisions on disparate evidence about a design's reliability, to substitute for the intuitive, informal arguments that are normally used now whenever statistical evidence is insufficient.

This whole discussion has been dominated by a concern about the *assessment* of safety. I hope it has shown that a sound approach may already offer the means for making decisions, in some practical design or system acceptance problems, on a rational (usually probabilistic) basis. However, there are serious limits in our ability to evaluate the overall safety of software-based systems with stringent safety requirements.

More experience is needed, concerning both methods for assessing and for obtaining safety, and in the form of both statistical data and qualitative results: reports about ease of use of methods, anecdotes about which approach has proven most valuable for individual problems. It is useful to consider that there are two obstacles to the diffusion of methods which are intuitively useful. One is created by practical difficulties and doubts, regarding, e.g., the ease of use of formal notations, the support for diverse software redundancy in off-the-shelf operating systems, and such. On-going research and gradual experimentation by industry are likely to remove these problems: toolsets are being provided, personnel trained, and experience put to use for improving the methods. The other obstacle is the inability to quantify the return to be expected from investment in these methods. Any advantages in terms of productivity, avoidance of expensive design changes and such would be relatively easy to measure. However, manufacturers turn to methods like formal proof and design diversity mainly because of stringent safety requirements; success under this respect cannot be demonstrated in the short term. An improvement - some guidance for software providers, customers and regulators seeking very safe systems - can be expected from the application of presumably useful methods to critical software with *moderate* dependability requirements. The existing projects aimed at providing case studies and affordable, usable tools can provide immediate help for developers of such applications. In turn, extensive study of the results obtained in such applications can provide some knowledge to be used in the world of extreme safety requirements. This is a

challenging task, as it requires a long-term cooperation among the involved parties to share and exploit the data as they become available.

Acknowledgments

The author's work was supported in part by the Commission of the European Communities, in the framework of the ESPRIT Basic Research Projects 3092 and 6362 "Predictably Dependable Computing Systems".

This paper is derived from a presentation at the 9th CSR Conference on Software Safety, Luxemburg, 1992. The author is indebted to many conference participants for their comments to that presentation and to Robert Malcolm, Jack Stankovic, Pierre-Jacques Courtois, Sarah Brocklehurst, Hermann Kopetz and the anonymous reviewers for extensive comments on previous versions of the paper.

References

[Abbott 1990] R.J. Abbott, "Resourceful systems for Fault Tolerance, Reliability, and Safety," vol. 22, no. 1, pp.35-68., 1990.

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

[Barwise 1989] J. Barwise, "Mathematical proofs of computer system correctness," *Notices of the American Mathematical Society*, vol. 36, pp.844-851, 1989.

[Brilliant et al. 1989] S.S. Brilliant, J.C. Knight and N.G. Leveson, "The Consistent Comparison Problem in N-Version Software," *IEEE TSE*, vol. 15, no. 11, pp.1481-1485, 1989.

[Brock and Hunt 1992] B. Brock and W.A. Hunt Jr. "Report on the Formal Specification and Partial Verification of the VIPER Microprocessor," in *Proceedings COMPASS '91, Sixth Annual Conference on Computer Assurance - Systems Integrity, Software Safety and Process Security*, pp. 91-98, Gaithersburg, Maryland, 1992.

[Brocklehurst et al. 1990] S. Brocklehurst, P.Y. Chan, B. Littlewood and J. Snell, "Recalibrating software reliability models," *IEEE TSE*, vol. SE-16, pp.458-470, 1990.

[Dandanell 1991] B. Dandanell, "Rigorous Development Using RAISE," *ACM SIGSOFT SEN*, vol. 16, no. 5 (Proc. ACM SIGSOFT '91 Conference on Software for Critical Systems, New Orleans, Louisiana, Dec. 1991), pp.29-43, 1991.

[Di Giandomenico and Strigini 1990] F. Di Giandomenico and L. Strigini. "Adjudicators for Diverse-Redundant Components," in *Proc. 9th Symposium of Reliable Distributed Systems*, pp. 114-123, Huntsville, Alabama, 1990.

[Eckhardt and Lee 1985] D.E. Eckhardt and L.D. Lee, "A theoretical basis for the analysis of multiversion software subject to coincident errors," *IEEE TSE*, vol. SE-11, pp.1511-1517, 1985.

[Hansen et al. 1991] L.M. Hansen, A.P. Ravn and H. Rischel. "Specifying and Verifying Requirements of Real-Time Systems," in *ACM SIGSOFT '91 Conference on Software for Critical Systems*, pp. 44-54, New Orleans, Louisiana, ACM Press, 1991.

[Jensen and Northcutt 1990] E.D. Jensen and J.D. Northcutt. "Alpha: A Non-Proprietary Operating System for Mission-Critical Real-Time Distributed Systems," in *1990 IEEE Workshop on Experimental Distributed Systems*, Huntsville, Alabama, 1990.

[Kersken and Saglietti 1992] M. Kersken and F. Saglietti. *Software Fault Tolerance: Achievement and Assessment Strategies*, Springer-Verlag, 1992.

[Kopetz et al. 1989] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft and R. Zainlinger, "Distributed Fault-Tolerant Real-Time Systems: The MARS approach," *IEEE Micro*, vol. 1, pp.25-40, 1989.

[Kopetz and Kim 1990] H. Kopetz and K. Kim. "Temporal Uncertainties in Interactions among Real-Time Objects," in *Proc. 9th Symposium on Reliable Distributed Systems*, pp. 165-174, Huntsville, Alabama, USA, 1990.

[Laprie 1989] J.C. Laprie. "Dependability: a Unifying Concept for Reliable Computing and Fault Tolerance," in *Dependability and Resilient Computing Systems, ed. T. Anderson*, pp. 1-28, Blackwell Scientific Publications, 1989.

[Laprie et al. 1990] J.C. Laprie, J. Arlat, C. Beounes and K. Kanoun, "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures," *IEEE Computer (Special Issue on Fault Tolerant Systems)*, vol. 23, no. 7, pp.39-51, 1990.

[Laprie et al. 1991] J.C. Laprie, K. Kanoun, C. Beounes and M. Kaâniche, "The KAT -- Knowledge-Action-Transformation -- Approach to the Modeling and Evaluation of Reliability and Availability Growth," *IEEE TSE*, vol. SE-17, pp. 370-382, 1991.

[Leveson 1991] N.G. Leveson, "Software Safety in Embedded Computer Systems," *CACM*, vol. 34, no. 2, pp.34-46, 1991.

[Littlewood and Miller 1989] B. Littlewood and D.R. Miller, "Conceptual modelling of coincident failures in multiversion software," *IEEE TSE*, vol. SE-15, pp.1596-1614, 1989.

[Littlewood and Strigini 1993] B. Littlewood and L. Strigini, "Validation of Ultra-High Dependability for Software-based Systems," *CACM*, vol. 36, no. 11, to appear, 1993.

[Parnas 1989] D.L. Parnas. "Can we tolerate software errors?," in *Proc. 11th IFIP World Congress*, p. 502, San Francisco, California, U.S.A., 1989.

[Powell 1992] D. Powell. "Failure Mode Assumptions and Assumption Coverage," in *Proc. 22nd International Symposium on Fault-Tolerant Computing*, pp. 386-395, Boston, Massachusetts, USA, 1992.

[Privara 1991] I. Privara. *Diverse Program Specification*, ESPRIT PDCS Technical Report 60, 1991.

[Ravn and Rischel 1991] A.P. Ravn and H. Rischel. "Requirements capture for embedded real-time systems," in *IMACS-IFAC Symposium MCTS 91*, Villeneuve d'Ascq, France, 1991.

[RTCA/EuroCAE 1985] RTCA/EuroCAE. *Software Considerations in Airborne Systems and Equipment Certification*, Doc DO178A/EUROCAE ED-12A, Radio Technical Commission for Aeronautics and European Organization for Civil Aviation Electronics, 1985.

[Rushby 1992] J. Rushby. "Formal methods for safety-critical systems," presentation at *9th CSR Conference on Software Safety*, Luxembourg, 1992.

[Saeed et al. 1991] A. Saeed, R. de Lemos and T. Anderson. "The Role of Formal Methods in the Requirements Analysis of Safety-Critical Systems: A train set example," in *Proc. 21st. IEEE Int. Symposium on Fault-Tolerant Computing*, pp. 478-485, Montreal, 1991.

[Smith et al. 1991] I.C. Smith, D.N. Wall and J.A. Baldwin. "DARTS - an experiment into cost of and diversity in safety critical computer systems," in *IFAC/IFIP/EWICS/SRE Safety of Computer Control Systems (SAFECOMP '91)*, pp. 35-39, Trondheim, Norway, 1991.

[Stankovic and Ramamritham 1989] J. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real--Time Operating Systems," *ACM Operating Systems Review*, vol. 23, no. 3, pp.54-71, 1989.

[Strigini 1990] L. Strigini. *Software Fault Tolerance*, ESPRIT PDCS Technical Report No 23, 1990.

[Theuretzbacher 1986] N. Theuretzbacher. "Using AI-Methods To Improve Software Safety," in *Proc. 5th IFAC Workshop SAFECOMP'86*, pp. 99-105, Sarlat, France, 1986.

[Voges 1988] U. Voges (Ed.). *Software diversity in computerized control systems*, Wien - New York, Springer-Verlag, 1988.

[Xu and Parnas 1991] J. Xu and D.L. Parnas, "On Satisfying Timing Constraints in Hard Real-Time Systems," *ACM SIGSOFT SEN*, vol. 16, no. 5 (Proc. ACM SIGSOFT '91 Conference on Software for Critical Systems, New Orleans, Louisiana, Dec. 1991), pp.132-145, 1991.