

System Dependability

Brian Randell

Dept. of Computing Science
University of Newcastle upon Tyne

Abstract. The paper starts with a brief account of how and why, at about the time of the birth of what is now INRIA, the author and his colleagues became interested in the subject now known as system dependability. The main body of the paper summarizes the work over the last three years of the ESPRIT Basic Research project on Predictably Dependable Computing Systems (PDCS). This is a long term collaborative research activity, centred on the problems (i) of producing quantitative methods for measuring and predicting the dependability of complex software/hardware systems, (ii) of incorporating such methods into the design process, and (iii) of developing appropriate architectures and components as bases for designing predictably dependable systems. A further section of the paper then describes, in somewhat more detail, one of the current activities within PDCS. This is work being carried out by the author in collaboration with an INRIA colleague, Dr. Jean-Charles Fabre, on a unified approach to providing both reliability and security termed Object-Oriented Fragmented Data Processing (OOFDP).

1. Introduction

Twenty-five years ago my personal research interests were focussed on system design methodology in the area of complex multiprocessing operating systems. I was then at the IBM T.J. Watson Research Center, where I and my colleagues were aiming to aid system designers by providing means of predicting the likely impact of their various design decisions on the ultimate performance of the overall final system. The approach we developed was based on expressing the system design as a simulation program. Such a program would evolve and grow vastly more detailed as the design process progressed and hence its careful structuring would be of paramount importance. It was these twin issues of system structuring and design

decision-making, though both motivated mainly by performance considerations, that were uppermost in my mind when I received an invitation to the first NATO Software Engineering Conference, held at Garmisch in 1968.

The impact of this conference, particularly on many of the attendees, was immense. For example, both Edsger Dijkstra and I have since gone on record as to how the discussions at this conference on the "software crisis", and the potential for software-induced catastrophes, strongly influenced our thinking and our subsequent research activities. In Dijkstra's case the discussions led him into an immensely fruitful long term study of the problems of producing high quality programs. In my own case, following my move to Newcastle soon after the Garmisch Conference, they led me to consider the then very novel, and still somewhat controversial, idea, of software fault tolerance. Suffice it to say that our respective choices of research problems suitably reflect our respective skills at program design and verification.

It will thus soon be twenty-five years since my research interests, though continuing to involve a strong concern with issues of system structuring, switched rather abruptly to an aspect of the topic that (thanks incidentally to a French colleague, of whom more anon) I now know as *system dependability*. However this is not intended as a historical talk, summarizing developments in system dependability over the entire twenty-five year period. Rather, I will concentrate largely on recent work on the subject undertaken in connection with the ESPRIT Basic Research Project entitled PDCS (Predictably Dependable Computing Systems) that I have the honour of leading.

2. Computing Systems

The title of our project, with its emphasis on "computing systems", implies that it is not concerned solely with either hardware or software. Indeed for many years I - not alone of course - have held the view that the conceptual differences between hardware and software design were small, and indeed diminishing as VLSI technology makes possible the implementation of ever more complex logical designs on a single chip.

I thus always react negatively to discussions which imply that, for example, software design practices are developing more slowly than hardware design practices, that software is especially prone to design faults, or even that

software designers are less professional than hardware designers, when the differences really concern relative complexity. For obvious practical reasons, the more complex algorithms involved in providing a system's required functionality will normally be implemented in software rather than hardware. And, though there is no well accepted measure of the complexity of a logical design, even by such inadequate means as "lines of code", "number of transistors, etc.", it is clear that the complexity of the software in most large scale computing systems is several orders of magnitude greater than that of the hardware.

Such differences in complexity, far more than cultural or technical differences, are the root cause of the differing practices that are followed by the hardware and software design communities. And it was in fact a debate about these differing practices at a meeting a few years ago of the IFIP Working Group 10.4 on System Dependability and Fault Tolerance which in part led me to formulate, with some colleagues, a programme of long term research which has since become the PDCS project.

This particular IFIP Working Group meeting was the occasion of a number of very interesting presentations on the role of predictive evaluation in hardware design. These presentations brought home to me the extent to which the computer industry's hardware designers routinely made, and had their further decision making guided by, predictions as to the likely ultimate reliability, availability and performance of their systems.

A number of different techniques were involved, from simple rules of thumb, through stochastic models, to complex simulations. However, it became clear during the discussions following the presentations that these techniques were being increasingly challenged by the ever-growing complexity of the logical designs that the hardware designers were being called upon to produce, and that none took account of the possibility of residual design faults.

Nevertheless, within these limitations, the skilled deployment of these techniques enabled the hardware design process to be much more of an engineering process than was, or indeed is, the case with most if not all software engineering. (Parenthetically, let me mention that my involvement in the 1968 NATO Software Engineering conference - whose title was an expression of an aspiration, rather than a reality - made me extremely cynical regarding the vast mass of software activity that was suddenly re-labelled

Software Engineering in the early 1970s. I have retained much of this cynicism to this day.)

Given this situation regarding the predictive evaluation of hardware designs, it was evident to me and my colleagues that it would be a very worthwhile although extremely challenging task to try to extend the scope of logical design evaluation techniques and tools (i) to deal with the much greater complexity levels that were characteristic of software sub-systems, and hence of complex computing systems as a whole, and (ii) if possible to allow for the all-too-real likelihood that residual design faults would exist at such complexity levels, despite all attempts to prevent this. Moreover, we felt that such techniques and tools were best investigated within an overall design and system framework - just as in reality, if they were to be useful, they would need to be deployed as an integral part of an overall system engineering design process.

3. System Dependability

Turning back to much earlier work for a moment, when I and colleagues at Newcastle became interested in the possibility of achieving useful degrees of design fault tolerance, we found that one of the problems facing us was the inadequacy for our purposes of the concepts and terminology that hardware reliability engineers were then using.

In the 1970s hardware engineers took various particular types of fault (stuck-at-zero, stuck-at-one, etc.) which might occur within a system as the starting point for their definitions of terms such as system reliability and system availability. But given not just the absence of any useful categorization of design faults, but also the realization that in many cases the actual identification of some particular aspect of a complex system design as being *the* fault might well be quite subjective, we felt in need of a more general set of concepts and definitions. And of course we wanted these definitions to be properly recursive, so that we could adequately discuss problems that might occur either within or between system components at any level of a system.

The alternative approach that we developed took as its starting point the notion of *failure*, whether of a system or a system component, to provide its intended services. Depending on circumstances, the failures of interest could concern differing aspects of the services - e.g. the average real-time response

achieved, the likelihood of producing the required results, the ability to avoid causing failures which could be catastrophic to the system's environment, the degree to which deliberate security intrusions can be prevented, etc. The ensuing generality of our definitions of terms thus led us to start using the term "reliability" in a much broader sense than was perhaps desirable, or acceptable to others; it was our French colleague, Dr. Jean-Claude Laprie of LAAS-CNRS, who came to our linguistic and conceptual rescue by proposing the use of the term "dependability" instead.

Dependability is defined as the trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers [Carter, 1982 #719; Laprie, 1992 #697]. Dependability thus includes as special cases such properties as reliability, integrity, privacy, safety, security, etc., and provides a very convenient means of subsuming these various concerns within a single conceptual framework. It also provides means of addressing the problem that what a user usually needs from a system is an appropriate balance of several such properties.

The problem of producing a fully adequate set of system dependability concepts and terminology, appropriate for all the ways in which "things could go wrong", and all approaches to trying to prevent or cope with this situation, turned out to be much more difficult and complicated than I imagined at the outset. Indeed it is a problem that has engaged the efforts of various groups of colleagues, mainly under Dr. Jean-Claude Laprie's leadership, in the IFIP Working Group, and the PDCS project, and has recently culminated (I hesitate to say terminated), in the publication of a five-language book. I will start by summarizing some basic definitions from this book [Laprie, 1992 #697]:

A system **failure** occurs when the delivered service no longer complies with the **specification**, the latter being an agreed description of the system's expected function and/or service. An **error** is that part of the system state which is liable to lead to subsequent failure: an error affecting the service is an indication that a failure occurs or had occurred. The adjudged or hypothesized cause of an error is a **fault**.

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

. . . ∅ failure ∅ fault ∅ error ∅ failure ∅ fault ∅ . . .

Faults and their sources are extremely diverse; they can be classified in various different ways. One can distinguish **accidental faults**, which appear or are created fortuitously, from **intentional faults**, which are created deliberately, presumably malevolently. One can, sometimes somewhat arbitrarily, classify faults as being **physical faults**, which are due to adverse physical phenomena, and **human-made faults**, which result from human imperfections. With respect to a system boundary, one can distinguish **internal faults**, which are those parts of the state of a system which when invoked by the computation activity will produce an error, from **external faults**, which result from interference or interaction with its physical or human environment. Depending on when a fault was created, it can be termed a **design fault**, i.e. one that arises during the development of the system or of its operating and maintenance procedures, or an **operational fault**, which appears during actual use of the system. And depending on the nature of the conditions which govern the presence of the fault, it can be classed as a **permanent** or a **temporary fault**.

The methods involved in developing a dependable computing system can be classed into:

- **fault prevention:** how to prevent fault occurrence or introduction;
- **fault tolerance;** how to provide a service complying with the specification in spite of faults;
- **fault removal;** how to reduce the presence (number, seriousness) of faults;
- **fault forecasting:** how to estimate the present number, the future incidence, and the consequence of faults."

These four means for dependability all feature in the work of PDCS, though we put comparatively little stress on the first, fault prevention. This is not to imply that we view this topic as being of less importance than the other three; rather it is that virtually all research related to system design and construction is of relevance to fault prevention, even when not undertaken with this as an explicit aim. However the point I would stress is that combined utilization of

all four means is normally needed if a high level of dependability is required - preferably at each step in the design and implementation process.

4. The PDCS Project

The above comments make it clear that dependability is very much a *systems issue*, since virtually all aspects of a computing system, and of the means by which it was specified, designed and constructed, can affect the system's overall dependability. Users gain little satisfaction from being assured that particular system components are functioning faultlessly or that particular parts of the design process have been carried out absolutely correctly if the overall *system* does not provide a level of dependability commensurate with the level of dependence that they have to place on it.

Moreover, realistic dependability prediction and assessment normally have to involve careful calculation of probabilities and risks, rather than naive belief in certainties. Thus they should, if at all possible, be based on stochastic models and well-established statistics, rather than simplistic assumptions, whether about the functioning of system hardware components, or the care with which system design processes have been carried out.

These were the views which Jean-Claude Laprie and I shared when we initiated the planning of what became the PDCS project, a project whose membership is drawn largely from the European contingent on IFIP WG 10.4. However, it also includes various other leading European researchers with the complementary skills and interests that we felt were needed to make a suitably broad attack on the overall problem of achieving predictable dependability for complex computing systems.

The project started in 1989, and had a first phase which we now term PDCS1 that ran for three years. The institutions and principal investigators that were involved in PDCS1 were:

Centre for Software Reliability, The City University, London, UK (Bev Littlewood)

IEI del CNR, Pisa, Italy (Lorenzo Strigini)

Institut für Algorithm und Kognitiv Systeme, Universität Karlsruhe, Karlsruhe, Federal Republic of Germany (Tom Beth)

LAAS-CNRS, Toulouse, France (Jean-Claude Laprie)

Computing Laboratory, The University of Newcastle upon Tyne,
Newcastle upon Tyne, UK (Brian Randell)

LRI, Université Paris-Sud, Paris, France (Marie-Claude Gaudel)

Institut für Technische Informatik, Technische Universität Wien,
Vienna, Austria (Herman Kopetz)

Department of Computer Science, The University of York, York, UK
(John McDermid)

The second phase, PDCS2, has just started, with almost the same set of institutions involved, though not all as full partners, and with one additional full partner, namely Chalmers Technical University, Göteborg.

In view of what I have already said, it should be clear that the single most important characteristic of our work is the stress we have, and are continuing to, put on the necessity of taking a "systems engineering" approach, aimed at achieving well-coordinated progress towards:

- (i) developing effective techniques for establishing realistic dependability requirements, and so producing dependability specifications against which the system design process and its resultant products can be assessed;
- (ii) producing quantitative methods for measuring and predicting the dependability of complex software/hardware systems, allowing for the possible presence of design and deliberate faults as well as operational faults;
- (iii) incorporating such methods more fully into the design process, and all its means of attempting to prevent, remove and tolerate faults, so as to make the process much more controlled and capable of allowing design decisions (and decisions concerning the actual deployment of the system) to be based on meaningful analyses of risks and quantified likely benefits; and, ultimately
- (iv) producing an effective design support environment populated both with the tools necessary to facilitate practical use of such techniques and

methods, and with ready-to-use families of system components with known dependability characteristics.

An important characteristic of the goal we thus set ourselves was, and remains, to facilitate increased use of quantitative assessments of system dependability. Clearly there will be limits to the extent that many process and product characteristics can be meaningfully and accurately quantified. Nevertheless, we feel that a degree of concentration on techniques which will help lay the foundations of increased use of quantitative methods is fully justified. Indeed, as I have implied already, our view is that increased effective use of quantitative methods is a prerequisite to turning the activity of constructing large computer-based systems into a true engineering discipline.

The PDCS group's ultimate long term objective, a design support environment which is well-populated with tools and ready-made components, and which fully supports the notion of predictably dependable design of large distributed real-time computing systems, is of course extremely ambitious, and not something one would expect to achieve within the time span of a single project, but it nevertheless provides a good long-term focus for our work.

At the outset of the PDCS1 project we concluded a detailed review of the state-of-the-art with the comments: "although much research has been, and is continuing to be, undertaken on the multi-faceted problem of system dependability of complex hardware/software systems, comparatively little progress has been made on drawing all the relevant research threads together. For example, there are to date few links between work on fault prevention and fault forecasting - links which might greatly facilitate the tasks of (i) deciding what levels of effort should be expended, in a major system development project, on what types and methods of fault avoidance and fault tolerance, and (ii) predicting the probable outcome. Similarly, there are inadequate links between research activities aimed at different facets of dependability, such as reliability and security. "

Many of the topics which the PDCS Project has worked on, and in many cases is still actively pursuing, are therefore particularly "integrative" in nature. These include:

- the very notion of design environments for fault-tolerant systems which incorporate dependability evaluation tools in order to assist design choices
- a fault tolerance approach to object-oriented system security using the fragmentation/redundancy/scattering technique
- the derivation of optimal test strategies mixing deterministic and statistical testing
- the validation (wrt both fault removal and fault forecasting) of fault-tolerant systems via fault injection
- the assessment of the effect of different system environments on the reliability of a software system" via two approaches: discrete-time reliability growth models and explanatory variables
- an attempt to develop a cost-based model of operational security similar to the time-based model for reliability that is universal for reliability
- work on extending the current limitations to the reliability evaluation of safety-critical software, via a combination of information relating to the product, its production process, and past experience on similar products.

Clearly, there is insufficient time for me to provide adequate descriptions of all of these and our various other activities (work on which has resulted in, or contributed to, over a hundred publications in refereed journals and international conferences over the last three years).

Instead, after summarizing the project's work under the four headings: Fault Prevention, Fault Tolerance, Fault Removal, and Fault Forecasting, I will concentrate on just one topic. I have three reasons for the particular choice I have made. First, it is an example of Anglo-French cooperation, second the French contribution is led by an INRIA Researcher based at LAAS, Dr. Jean-Charles Fabre, and finally it is the research topic within PDCS in which I am currently most involved. It concerns the provision of reliability *and* security via a single unified mechanism, and takes advantage of object-oriented system structuring - something that, for all its current trendiness, I have remained sympathetic to ever since the original Simula Common Base

Language report was published, twenty-five years ago. Clearly this is a vintage year for Silver Jubilees!

5. An Overview of Work to Date in PDCS

This overview is based on sections of a report on PDCS1 - like this report it is structured according to the four principal means involved in achieving dependable systems: fault prevention, fault tolerance, fault removal and fault forecasting.

5.1. Fault Prevention

Any computer system that meets its requirements is dependable only to the extent that those requirements represent the user's needs; yet *requirements analysis and specification* is still one of the more neglected areas of computer science. Most system failures can be attributed to subtle design faults introduced because of a mismatch of assumptions and requirements relevant to various different system aspects. Work in the ESPRIT ORDIT project has produced a methodological framework for understanding the process of elicitation, modelling and re-presentation back to the user of organizational requirements (i.e. requirements which affect the design of the whole surrounding socio-technical system and not just the design of the computer system). In PDCS1 we have used this framework to look at some issues of determining security policies [Dobson, 1990 #495] and as a basis of the meaning of the terms "safety" and "security" [Burns, 1992 #609]. We have also examined the process by which non-functional requirements are transformed into functional requirements.

Work on *real time object models and notations* has been oriented towards providing a framework for specifying, analyzing and implementing real-time systems. Work in PDCS1 has focused on foundational issues, establishing classes of timeliness requirements and ways of classifying architectures. The classification has centred on ideas of timing grids which enable architectures to be characterised, and bounds to be placed on certain temporal characteristics, e.g. carrying out 'synchronised' actions at physically different nodes in a distributed system. Foundational work was also undertaken in producing (formally based) object models for defining and reasoning about architectures, although this has not yet reached a stage suitable for publication.

Timeliness analysis is a central issue to PDCS. In real-time systems, the intended result must be produced within a specified time interval after the occurrence of a stimulus. Considerable work in a number of areas has been performed recently by PDCS [Vrchoticky, 1991 #641] and a number of other projects [Shaw, 1989 #638; Kligerman, 1986 #639; Puschner, 1989 #614; Vrchoticky, 1991 #641]. For example, we have analyzed the diverse effects which determine the maximum execution time of a simple application, the control of a rolling ball, in the very restricted execution environment given by the MARS architecture.

5.2. Fault Tolerance

Advances in our understanding of *fault tolerance strategies, architectures and notations* have been made in PDCS1, including: (i) classifying existing architectures for combined hardware and software fault tolerance [Laprie, 1990 #420]; (ii) generalizing the notion of adjudication in redundant systems [Di Giandomenico, 1990 #419]; (iii) proposing design rules for the composition of subsystems employing different fault tolerance strategies [Strigini, 1991 #491]. In PDCS1 two description languages have been defined and investigated [Babaoglu, 1991 #546; Bondavalli, 1990 #421], based on a data-flow model, suitable both for describing fault-tolerant schemes in application software and as intermediate languages to be supported by a fault-tolerant interpretation layer. A design notation for a wide class of fault-tolerant software structures is proposed in [Liu, 1991 #547], and an approach to fault-tolerant design based on system-level diagnosis theory has been introduced [Xu, 1991 #526].

The application of formal methods to program development has already made it possible in certain cases to refine a program specification into the text of a program which is guaranteed to satisfy the specification. It is now possible to study the use of such techniques for fault-tolerant programs. In PDCS1, a method has been explored for obtaining diversity in redundant variants of a software component, while preserving consistency with a formal specification [Gaschignard, 1990 #523].

The production of a *design support environment* populated by a number of ready-to-use tools will be fundamental to the design of predictably dependable computing systems in practice. In PDCS1 several significant advances have been made with regard to such tools, in particular by the

definition and implementation of SoRel from LAAS, a tool for quantitative dependability evaluation and by the development at Vienna of a tool set for the MARS Design System (MARDS), which is a prototype design environment for real-time systems.

Reaching agreement is an essential problem in distributed fault-tolerant systems. Examples are agreement on data (reliable broadcast)[Babaoglu, 1985 #14; Cristian, 1985 #53], time (clock synchronization), and component state (membership) [Cristian, 1988 #640; Kopetz, 1991 #487] in spite of different kinds of failures in the components and in the communication system. In PDCS1 we have worked on the problem of distributed clock synchronization and the distributed membership problem, and have also investigated the problems of a monitoring service for improving the availability of distributed systems and the security aspects.

Several solutions have been proposed to the problem of *tolerance of accidental and intentional operational faults*: classical solutions based on data replication and ciphering, information dispersal for files [Rabin, 1989 #496], and alternate routings for communications [Koga, 1982 #412]. A novel solution is Fragmentation-Redundancy-Scattering (FRS) [Fray, 1986 #382], the principle of which is to split sensitive information into fragments, to add redundancy to them, and finally to scatter them over a distributed system. Any isolated fragment does not by itself provide any significant information. Availability and integrity are obtained through redundancy of fragments, and confidentiality is ensured due to the absence of logical relation between the fragments. This is the basis of the work on object-oriented fragmented data processing [Randell, 1991 #568] described in more detail below.

5.3. Fault Removal

Regarding work on *testing in the value domain*, software test criteria proposed in the literature as guides for determining input test cases relate to the structure or to the function of the software, defining respectively structural and functional testing. Using these criteria, the methods for generating the test cases can be either deterministic or probabilistic. In the first case test data are predetermined by selection in accordance with chosen criteria. In the second case test inputs are generated according to a probability distribution on the input domain; both the distribution and the number of

input data items being determined according to the chosen criteria. Work in PDCS1 concentrated upon examination of the fault revealing power of software statistical testing [Thévenod-Fosse, 1989 #447]. This involves the notion of statistical test quality with respect to one or several criteria, and a new ordering relation to compare the stringency of different criteria with regard to random test data. We defined an original testing method, *structural statistical testing*, and the partial ordering of fifteen current structural criteria is shown in [Thévenod-Fosse, 1990 #438]. Experiments were conducted on industrial software (four real programs) from the nuclear field [Thévenod-Fosse, 1991 #443]. These indicated that the power of structural statistical testing — i.e. of random test data generated according to an appropriate distribution defined so that the program structure is properly scanned — to reveal faults is higher than that of deterministic test data, or of random data generated from a uniform distribution over the input domain. The efficiency of the mixed testing strategy combining deterministic and random test inputs [Thévenod-Fosse, 1989 #447] was confirmed by these experiments. Another experiment was conducted [Jassim, 1990 #552] on a software product, in which statistical testing was found to be superior to deterministic testing. Surprisingly, there was some evidence that this superiority of statistical testing was more pronounced for the more elusive (smaller rate) faults than for others.

Concerning *testing in the time domain*, within PDCS1 we have developed the notions of event-triggered (ET) and time-triggered (TT) systems [Kopetz, 1990 #450]. These characterize two different system architectures where ET systems observe (internal and external) events and react to them by initiating appropriate actions, whereas TT systems initiate all their actions only at predefined points in time. TT systems are especially attractive for implementing real-time applications, because they allow design of the temporal behaviour of the system. In ET systems, many decisions affecting the temporal behaviour must be taken at run-time (e.g. scheduling) or are left to "chance" (e.g. ordering of messages). Our preliminary analysis of the problems of testing real-time systems has shown that TT systems offer a number of specific advantages which considerably ease the testing of such systems, especially with respect to repeatability and controllability of test runs. Based on these system properties, we have developed a test methodology for the MARS architecture [Schütz, 1990 #483]. This test methodology includes the definition of several test phases, each with a

distinct and well-defined purpose, the design of test beds to facilitate test execution, and the proposal of a number of test tools.

As dependability of fault-tolerant systems is conditioned to a large extent by the efficiency of the fault tolerance algorithms and mechanisms, testing of fault tolerance is to be considered as an integral part of the validation of fault-tolerant systems. Fault injection constitutes an invaluable means towards this end by providing inputs for exercising fault tolerance. Up to now — except for the work related to the validation of fault tolerant protocols recently reported in [Echtle, 1991 #455] — most fault injection studies have focused on the fault forecasting objective. Previous experiments carried out on actual fault tolerant systems [Arlat, 1990 #456] have shown that the data gathered during experiments aimed at fault forecasting can be used in practice in a feedback loop so as to influence the design and implementation of fault tolerance algorithms and mechanisms: i.e. these experiments contribute to fault removal. Preliminary studies carried out in PDCS1 [Arlat, 1991 #454] concerning the refinement of the fault injection input and output attributes introduced in [Arlat, 1989 #453] have shown that the elaboration of such a test sequence can be obtained from the definition of structural and behavioural simulation models of fault tolerance that serve as bases for the application of the results of the test theory.

5.4. Fault Forecasting

Several important advances have been made in *reliability and availability modelling* in PDCS1. The position now is that we understand fairly well how to obtain reasonably accurate reliability growth predictions in a wide class of circumstances. More importantly, we have techniques that allow us to know in a particular context whether the answers are ones that we should trust. All this has come about through the invention and validation of techniques such as u-plots, prequential likelihood, etc. for the analysis of predictive accuracy [Littlewood, 1990 #500]. These techniques have allowed us to demonstrate that new non-parametric models, with much weaker underlying assumptions than conventional models, can perform with comparable accuracy in many situations [Brocklehurst, 1989 #540; Brocklehurst, 1991 #541]. They have also allowed us to develop a powerful and general new technique of dynamic model recalibration, which can improve the accuracy of certain models in particular circumstances, often with dramatic effect [Brocklehurst, 1990 #427].

Other important results obtained in PDCS1 widen the scope of the models. We have generalized (i) the classical, hardware oriented, reliability and availability theory to incorporate software as well, and (ii) the software reliability growth theory, to incorporate hardware as well [Laprie, 1991 #542]. We are now able to model multi-component systems taking into account reliability growth of their components thanks to the "transformation approach" where the (classical) Markov model of a system in stable reliability is transformed into another Markov model which incorporates reliability growth thanks to some properties of our hyperexponential model [Laprie, 1991 #440]. This approach has been applied to model fault-tolerant software architectures in presence of reliability growth: we have shown that unreliability is strongly dependent on reliability growth factors of the different components [Kanoun, 1991 #543]. Tools have been developed to facilitate the use of these new techniques: SoRel (for software reliability analysis and prediction) and a tool for reliability prediction and recalibration.

Concerning *statistical testing of software*, previous work has focused on the quantitative assessment of reliability. Two different approaches have been proposed depending on whether the evaluation is deduced from a zero-failure test experiment [Parnas, 1990 #533; Miller, 1989 #573], or from failure records during the test phase [Cho, 1987 #530]. The applicability of the second approach is restricted to non-critical software as it involves observed failures without fault fixing. We have therefore put a particular emphasis on the evaluation from a zero-failure experiment for which we propose the use of simple Markov models — both in continuous and discrete time — to quantify the test duration required to ensure a target reliability or availability objective with a given confidence level [Thévenod-Fosse, 1990 #534; Thévenod-Fosse, 1991 #520].

We now have a good general understanding of how to evaluate reliability when relatively modest levels are needed. When it comes to *evaluation for ultra-high depend-ability*, however, the difficulties seem very severe. In PDCS1 we have begun to investigate these limits to evaluation in a formal way [Littlewood, 1991 #551]. We have shown that none of the various approaches alone is able to assure ultra-high dependability. For example, current models are limited to evaluating reliability levels of approximately the order of 10^{-4} failures per hour. However, it remains to be seen how much we can increase our confidence when we combine information from disparate

sources, e.g. formal verification using discrete domain models, operational testing, knowledge of fault-tolerant architecture, etc. [Laprie, 1991 #445].

Several modelling studies of system behaviour in the presence of faults were carried out in PDCS1. These included pipelines of TMR nodes [Ezhilchelvan, 1990 #457], high speed data networks [Mitra, 1990 #437] and multiprocessor systems with arbitrary failure modes [Chakka, 1992 #578]. A number of interesting results were obtained, and in one case a new numerical solution methodology was employed. Because of their generality, Markov and semi-Markov models are able to represent complex interactions among system components (e.g. state dependent failure rate, complex repair policy). Their main limitation is the inability to handle many states, and much further work is needed on *large state space modelling*. For realistic models, the state space requirements often exceed the memory and CPU capabilities that can be expected from computing systems, even in the foreseeable future. Approximations, for instance state space truncation, can be applied to cope with this state space explosion, but the numerical errors introduced are difficult to quantify, especially if the dependability measures of interest are related to rare events (i.e. faults in the present case). We have recently started the study of a new technique to evaluate the eigenvector system of large (possibly infinite) Markov chains [Courtois, 1984 #460; Courtois, 1986 #461; Courtois, 1990 #462], which allows both steady-state and transient system characteristics to be evaluated and so can be applied to determine measures of availability.

Since *coverage evaluation by fault injection* gives *conditional* dependability measures (i.e., conditioned on the occurrence of a fault), models incorporating the fault occurrence process are also needed in order to enable the evaluation of dependability measures. For this purpose, we have defined [Arlat, 1989 #453; Arlat, 1990 #529] an experimental evaluation method aimed at bridging the gap between (i) the analytical modeling approaches used for the representation of the fault occurrence process (e.g. Monte Carlo simulations, closed-form expressions, Markov chains) and (ii) the experimental measures obtained by fault injection approaches characterizing the error processing and fault treatment provided by the fault tolerance algorithms and mechanisms. Two fault injection systems, which we have developed outside of PDCS1, are now available: one uses heavy-ion radiation to inject transient faults into integrated circuits [Gunneflo, 1989 #532] and has been applied to evaluating watchdog error detection schemes, the second

is a pin-level injection system, called MESSALINE that has been applied to the validation of two fault-tolerant systems [Arlat, 1990 #456; Arlat, 1990 #528].

In PDCS1 we have started work on *security modelling* with the intention of obtaining operational measures of security. Although there are many important ways in which the reliability metaphor applies to security (such as the inevitability of a stochastic approach), some of these similarities tend to operate at a high conceptual level and for practical application there are some important differences which need to be recognized [Littlewood, 1991 #611]. For example, it seems that the choice of variable to represent the role played by time in the reliability models needs to be chosen with great care in security: a variable that captures the notion of effort expended, not necessarily in time, seems essential. In addition, the importance of different viewpoints needs to be emphasized in security, since it is unlikely that these will coincide even in the case when each observer has a large amount of information.

6. Object-Oriented Fragmented Data Processing

Reliability/availability and security, though attributes of the generic concept of dependability, are often considered separately because the techniques used to achieve them are usually perceived as being mutually antagonistic. Firstly, *reliability* and *availability* are generally achieved by incorporating mechanisms for tolerating any faults (especially accidental faults) that occur, or that remain despite attempts at fault prevention during the system design process. These techniques will of necessity involve space and/or time redundancy; they can easily take advantage of a distributed computing architecture by means of replicated computation using sets of untrusted (or fallible) processors. Secondly, *security features* are generally achieved by means of fault prevention mechanisms (w.r.t. intentional faults, such as intrusions) whereby critical applications are implemented using physically and/or logically protected computers; such protection is usually based on the *TCB* (Trusted Computing Base) or *NTCB* (Network Trusted Computing Base) concepts.

6.1. Fragmented Data Processing

The technique termed "Fragmented Data Processing" (FDP) [Fray, 1989 #384; Deswarte, 1991 #441; Trouessin, 1991 #383] is an approach to the *combined* provision of overall system security (in the sense of data and processing confidentiality) and reliability in distributed systems. It can provide each of the users of a distributed system with an individual set of processing and storage resources which are to a great extent protected not only from the effects of hardware and software faults but also of so-called "intrusions". By this term we mean (presumably) deliberate attempts by other (possibly unauthorized) users of the system to gain information from, or modify, or deny access to, the user's resources. For example, such attempts could even involve tampering physically with the hardware, or inserting "Trojan Horse" software.

The FDP approach, and the original Fragmentation-Redundancy-Scattering (FRS) scheme [Fray, 1986 #382] on which it is based, are strongly related to conventional fault tolerance techniques. FDP achieves high reliability/availability and security for critical applications by arranging that their execution depends merely on (i) the correct execution of a majority of a set of copies of each of a number of program fragments, and (ii) the reliable storage of a majority of a set of copies of each of a number of data fragments; such fragments are widely distributed across a number of computers in a distributed computing system so as to impede intruders and to tolerate faults, and are defined so as to ensure that an isolated fragment is not significant, due to the lack of information it would provide to a potential intruder.

In effect, fragmentation and scattering is just a form of encryption, though one whose overheads are quite modest, and whose use fits well with general fault tolerance provisions (replication and voting) that are aimed at providing high reliability and availability despite the presence of hardware and software faults. Indeed, the crucial point about FDP is that the services it provides depend not on the integrity of any individual software or hardware components (which would imply the existence of "single points of failure"), but rather on majority voting by members of various sets of components. It simply presumes that such majorities exist (thus assuming a limit on the number of simultaneous faults) and in particular that voting is not being invalidated by either accidental or deliberate collusion between voters.

More specifically, systems employing FDP are, from the point of view of each user, divided into two sets of resources, namely a "trusted" (and it is hoped trustworthy) set and an "untrusted" set. Typically, the untrusted resources form a shared set of processing and storage servers, which users access from their individually trusted personal workstations, and it is in these terms that the technique will be described here.

Two major implemented examples of the application of the original FDP scheme have been completed, both using the DELTA-4 distributed system [Powell, 1991 #722]. These are respectively an archiving system [Ranéa, 1988 #721] and a user authorization service [Blain, 1990 #414; Deswarte, 1991 #441]. These both involved explicit implementation of the fragmentation and scattering mechanisms by the application programmer.

6.2. Object-Oriented FDP and its Implementation

Although FDP was not originally based on the use of object-oriented programming, the object-oriented model gives a reasonably straightforward method of implementing FDP which involves arranging that objects are split into fragments consisting of the subsidiary objects of which they were originally composed. This is done by defining and providing an implementation of the appropriate class characteristic and then choosing which classes of object should inherit this characteristic. Thus just as Arjuna [Shrivastava, 1991 #398] allows all objects of a class to be declared as recoverable, so the objects of a given class could be declared to be "Secured" by being fragmented and scattered. By such means the user can exercise control over the granularity of fragmentation without being involved in the actual implementation of the FDP mechanisms.

Declaring a given class to be "Secured" will of course mean arranging for this class to inherit a set of facilities defined in an appropriate class declaration. These characteristics will, for example (i) ensure that when objects of this given class are created, their constituent sub-objects will be scattered, and (ii) provide each object with any necessary information, such as a "key", needed to control the fragmentation and scattering and the means by which the object's operations access its scattered sub-objects. Some such keys might be used only at compile and generation time, and then deliberately discarded (i.e. for what can be termed "static" fragmentation and scattering). Others are likely to be retained within, or associated with, objects at run time

(e.g. for "dynamic" fragmentation and scattering, in which sub-object names are computed when the sub-objects are invoked). The keys themselves need to be protected - for example using the notion of a threshold scheme [Shamir, 1979 #415].

An object which has been declared to inherit the characteristic "secured" would thus be largely empty, apart from the information necessary for accessing its now remote subsidiary objects, and the code (or a reference to the code) for the various operations (methods). The fact that the subsidiary objects were allocated, in many cases, to separate machines would involve overheads, but would also provide significant potential parallelism for achieving a speeding-up of the original object's methods. (There already exist a number of techniques (under various different names) which are somewhat akin to fragmentation and scattering, aimed at exploiting parallelism for performance purposes rather than at providing security. These include, at the hardware level, so-called "disk striping" and, in object-based programming, the object fragmentation provisions of the SOS system at INRIA [Makpangou, 1991 #403; Shapiro, 1989 #401].)

The actual means by which such forms of fragmentation and scattering can be achieved, e.g. the methods for placing, and later accessing remote subsidiary objects, will depend on the strategy that is being provided to users for handling distribution problems. For example, the programming of the class "secured" might be based on the use of a simple, but rather inflexible, facility of a single virtual name space (e.g. [Bal, 1988 #413]), whose implementation embodies and hides the distribution policies which are in use. Another alternative is the facility provided in SOS for declaring and implementing shared distributed objects (termed "fragmented objects" in SOS) out of elementary objects which are located on different computers. (Clearly, with such an approach the distribution policies remain under user program control.)

6.3. An Electronic Diary Example

Object-oriented fragmented data processing (OOFDP) has to date been investigated using several detailed examples. The first was in fact based on part of the specification of the user authorization service [Blain, 1990 #414; Deswarte, 1991 #441] provided in the DELTA-4 distributed system [Powell,

1991 #722]. A fuller account of this first example to be found in [Randell, 1991 #497].

More recently an example based on a distributed Electronic-Diary has been designed using Eiffel [Meyer, 1987 #411] tools and implemented on top of the DELTA-4 Support Environment (DELTASE). The summary given here is based on a fuller account provided in [Fabre, 1992 #720].

The electronic diary service is described by a small set of classes. These show how the information related to a meeting is composed of a given topic, a group of people attending, a venue and time/date information. Any two or more of such items are considered as constituting confidential information. Otherwise, any person attending is defined by several identification items and can be considered as being public information.

Some of the object classes (and their component objects) forming the E-Diary application object are shown in Figure 1, where an asterisk indicates the possibility of there being several components of a given object class.

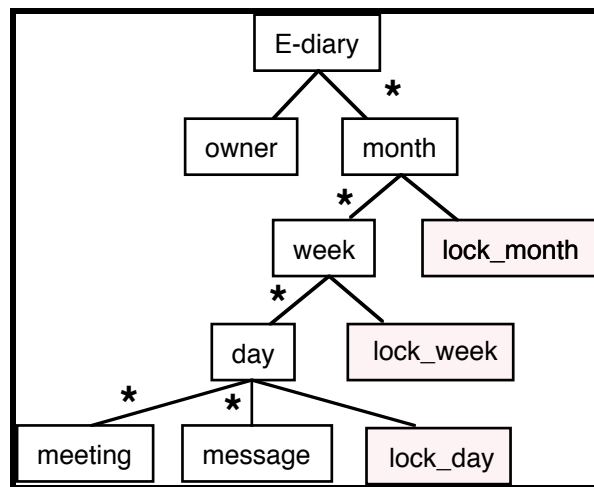


Figure 1: The E-Diary object composition hierarchy

The object hierarchy represented in Figure 1 for the E-diary service is as follows: the **E-Diary** is composed of several **month** objects and is owned by a given user (**owner**). Each month is composed of a number of **weeks** and can be locked (**lock_month**) for a given reason (holidays, for instance). Each week is composed of a number of **days** and can also be locked (**lock_week**) for a given reason (travel abroad, for instance). Any day is

composed of a list of **meetings**, a list of **messages** (note pad) and can be also locked for a given reason (**lock_day**). Any lock set to true implies that no meeting can be allocated in the month, week or day, respectively. The E-Diary is considered as a persistent object and can thus be activated (from persistent storage) after being created. It offers several services to the owner: create, modify, move, delete a meeting, put, release a message in the note pad of a given day, and lock a month, a week, or a day for a given reason.

The object which is of interest in the hierarchy shown above is the meeting object which contains confidential information; the composition hierarchy of this object is presented in Figure 2. A **meeting** is composed of a persons list (**P-list**), a **venue**, **time** and **topic**. The P-list can be implemented in various different ways, possibly using the Eiffel pre-defined class list (of **persons**). Person is composed of three sub-objects in our example: **name**, **address** and **position**.

The object hierarchies presented in Figure 1 and 2 illustrate (in a form similar to Eiffel browser output) the various components in the design of the E-Diary object down to elementary objects (i.e. a combination of Eiffel elementary objects such as integers, booleans, strings...). Some of the elementary objects represented by grey boxes are confidential leaves of the tree that it is assumed for present purposes cannot be usefully decomposed into smaller objects; for instance the topic is a string that is ciphered to ensure confidentiality as soon as it is entered by the user in the system. The same is true for lock objects which correspond to a boolean value and a string that indicates the locking reason.

An example is given below of the effects that can be achieved by arranging that the characteristic "secured" be inherited by a given object class, say the meeting class that was shown in Figure 1. In this and the next figure, the various objects are labelled with numbers indicating the different (sets of) computers they have been allocated to. These numbers have been chosen based on the simplistic rule that the immediate sub-objects of any fragmented object, and the object itself, are allocated to different (sets of) computers, depending on which class(es) of objects have been defined to inherit the characteristic "Secured". Inheritance of the "secured" characteristic is denoted by "Secured: ObjectClass" in the figures.

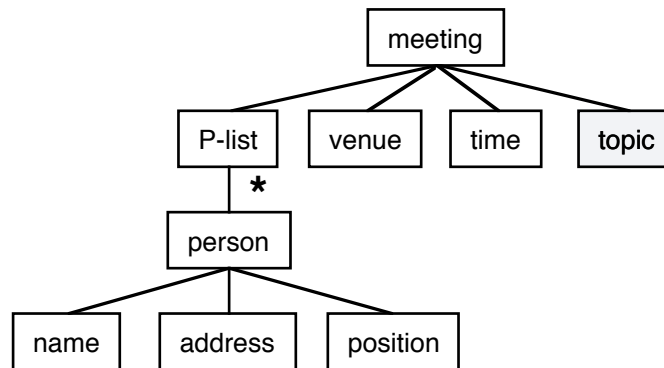


Figure 2: The Meeting object composition hierarchy

Site number 1 represents the user site where the owner is able to execute management operations (in particular input/output operations) and where all the meeting descriptors are located when FDP is not used. In this first case the characteristic "secured" is attached to the *meeting* class. This solution leads to processing (and perhaps storing) *Person list*, *venue*, *time* and *topic* at distinct sites as shown in Figure 3. (The fragmentation and scattering of the P-list objects is discussed later.)

In this case, site 1 is responsible for the management of *meeting* objects. Considering just *P-list* objects, all the *person* objects that appear in the meeting will be managed (and perhaps stored) at the same site (say site 2). An intruder located at site 2 is unable to find out about the *topic* of the meeting (even in its enciphered form) ; the confidentiality of the relation (*person list*, *topic*) is thus preserved by sites 2 and 5. Similarly, an intruder located at site 5 is able to obtain the (enciphered) topic of the meeting but is unable to find out the list of persons attending.

Secured: meeting

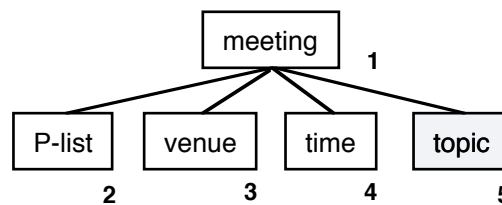


Figure 3: "Secured" meeting descriptors

The characteristic "secured" could however be attached not just to the meetings but also inherited by the *P-list* object as in Figure 4.

This solution provides a complete fragmentation and scattering of *person* objects belonging to the *P-list*. An intruder located at any of sites 6 to 9 is unable to get the P-list information (the list of persons attending the same meeting). At site 2, the P-list object is a collection of references to *person* objects: these references are produced by a naming facility based on one-way functions similar to those used in the archiving system described in [Deswarte, 1991 #441].

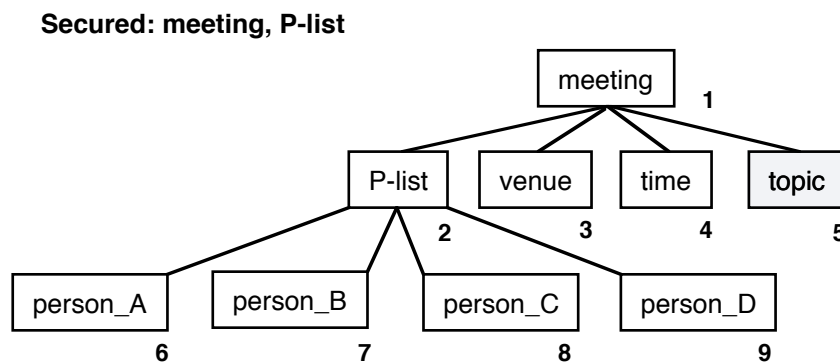


Figure 4: Secured meetings and P-list by inheritance

This last solution is the basis of the current implementation of E-Diary on the DELTA-4 platform. DELTA-4 [Powell, 1991 #722] does not provide an object-oriented layer but provides a run time support for objects as a collection of servers responsible for object management; a server is defined by an interface composed of a set of operations described using an Interface Definition Language and operation activation from the clients is transparent. In our implementation a server is associated with an Eiffel class and is responsible for the management of object instances of this class. For a given class several servers can be created on several sites, such as *person* servers in the above example. The Eiffel design presented in this section has been mapped onto DELTA-4 by hand. The complete application including more objects and more confidentiality constraints (including operator interface, ciphering and naming functions) is currently running using scattering and replication (with majority voting) on a set of Unix workstations at LAAS,

using the DELTASE layer and the DELTA-4 error processing protocols, based on DELTA-4's Multicast Communication System.

6.4. An Initial Assessment

Experiments such as the above indicate to us that FDP, in common with certain other approaches to security so far explored mainly in the database world, can derive significant benefits from being viewed and used in conjunction with a suitable object-oriented structuring scheme. Indeed, its provision as an independently inheritable characteristic alongside the use of several forms of inheritable reliability characteristics (such as "stable" and "atomic") that have already been devised elsewhere, such as in the Arjuna Project, seems perfectly feasible. However, one particularly attractive feature of the FDP technique is that it would seem to have the potential of being simultaneously beneficial not just to security and reliability but also, because of its exploitation of parallelism, to performance - characteristics which are normally mutually antagonistic!

Detailed experimental investigations are however now needed to determine the likely actual cost/effectiveness of OOFDP, as compared both to the FDP that has already been implemented in DELTA-4, and to any other approaches to the joint provision of reliability and security.

7. Concluding Remarks

One of the great difficulties about system dependability is also one of its greatest attractions as a research area. This is the fact that, as mentioned earlier, virtually every aspect of a system and the way in which it is developed is potentially relevant to the system's eventual dependability. Thus, since Newcastle took up the topic nearly twenty-five years ago our work has, at different times, concentrated on programming languages, processor and memory architectures, networking protocols, distributed systems, etc. (A much longer list of topics are covered by the expertise of the PDCS community as a whole.) Another "attraction" of the research topic is that as improved levels of system dependability are achieved, all too often one finds levels of user dependence and of required system functionality increasing correspondingly, so putting further demands on the system dependability research community. This I suppose is perhaps my main excuse for having remained involved in the topic for as long as INRIA has existed - a claim I do

not expect to be invited, or even able, to repeat at INRIA's golden jubilee! However I have much appreciated this chance to help celebrate the Silver Jubilee.

8. Acknowledgements

Self-evidently, the work I have discussed in this paper has been largely carried out by my colleagues, whose texts I have also drawn on extensively - it is a pleasure to acknowledge the debt I owe to them all. The work of PDCS is supported by the CEC's ESPRIT Programme.