Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective

Denis Besnard, Cristina Gacek
and Cliff B. Jones (Eds)

Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective

Denis Besnard
Cristina Gacek
Cliff B. Jones

School of Computing Science
Claremont Tower
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU
UK

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2005934527

ISBN-10: 1-84628-110-5 e-ISBN: 1-84628-111-3 Printed on acid-free paper
ISBN-13: 978-1-84628-110-5

© Springer-Verlag London Limited 2006

Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licences issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.

The use of registered names, trademarks, etc., in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.

The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

Printed in the United States of America (MVY)

9 8 7 6 5 4 3 2 1

Springer Science+Business Media
springer.com

Preface

Computer-based systems are now essential to everyday life. They involve both technical (hardware/software) components and human beings as active participants. Whenever we fly aboard an aircraft or withdraw money from a cash point, a combination of humans, machines and software is supporting the delivery of the service. These systems and many others benefit from the miniaturisation and cost reduction of the hardware which has made it possible for computers to be embedded everywhere. An equally remarkable development is the software involved: today, systems are built which were literally unthinkable twenty or thirty years ago. Measured in terms of their function, the productivity of their creation has also advanced enormously (largely because of the software infrastructure). Even the dependability of the best of todays software is praiseworthy when one considers the complexity of the functionality provided. Solid engineering and the increasing adoption of methods based on firmly established theory are to be thanked here. However, in large and complex systems, there remain major challenges to achieving dependability when complex interactions exist between technical and human components.

Large and complex things are understood as assemblages of simpler components: the way these components fit together is the *structure* of the system. Structure can be real and physical, or a subjective mental tool for analysis. It is often possible to view a complex system as having different structures: one useful view of the eventual structure will often be strongly related to the way in which the system was built (or came into being). This book addresses many aspects of the way in which structure can affect the dependability of computer-based systems. It is, for example, essential to be able to identify checking and recovery components and layers in the structure of a system; it is equally important to be able to analyse the dependability of a complex system in terms of the dependability of its components and of the way those components can affect each other within the system structure.

The work on the connection of structure with dependability of systems is one outcome of a large interdisciplinary project, DIRC[1], which started in 2000. The DIRC researchers come from a number of disciplines including computing science, psychol-

[1] The Dependability Interdisciplinary Research Collaboration: visit DIRC at http://www.dirc.org.uk

ogy, sociology and statistics. Within this project, funded by EPSRC[2], it is assumed that computer-based systems are multi-faceted in such a way that their design and analysis (in the largest sense) require skills that reach far beyond the computational and informatics aspects of the problem. To guarantee delivery of an acceptable service many dimensions must be taken into account – for instance, how humans use computers, security concerns, and the sociological implications of integrating machines into a pre-existing environment. All of these dimensions have structural aspects. The diverse set of researchers engaged over five years in the DIRC collaboration offers a unique opportunity for an interdisciplinary book on this fundamental topic of structure, drawing on a broad range of contributions. We hope that it will provide practitioners, commissioners and researchers with an important resource.

The two introductory chapters (by Jones & Randell, and Jackson, respectively) provide complementary visions. Jones and Randell discuss a well-known analysis framework (the dependability taxonomy) that categorises both problems and their mitigation. This framework defines a number of terms that readers will encounter throughout the book. In contrast to this theoretical view of computer-based systems, Jackson demonstrates that software engineering requires a decompositional analysis of a problem.

In the *System properties* section, Felici describes a central feature of systems: evolution. He puts forward the idea that there is not just one evolution, but several in parallel, ranging from a piece of software to the entire organisation, that together define the life of the computer-based system. The second contribution to the *System properties* section deals with time. Burns and Baxter develop Newells theory of time bands and demonstrate how this framework can help to analyse time-related events at a variety of granularities within computer-based systems.

The *Human components* section specifically adopts a human-centred view of systems. Besnard focuses on how dependability can be enhanced or hindered by the application of rules. The application of procedures and programs by the corresponding agent (human or machine) is addressed by considering a number of industrial cases. An industrial case-based approach also drives the chapter from Besnard and Baxter in their analysis of cognitive conflicts. They demonstrate how important it is that humans interacting with an automated system should understand the principles of its functioning.

To specify which features a given system should have, one must be able to understand what the system is doing, whatever its scale. This is the scope of the *System descriptions* section. In this section, the authors discuss examples of description techniques at four different granularities. The first level is software-based. Gacek and de Lemos discuss architectural description languages (ADLs) for capturing and analysing the architecture of software systems and how they can help in achieving dependability. The second level of granularity is about reasoning and visualising problems. Gurr describes how diagrams can be used to convey an intuitive understanding of complex logical relations between abstract objects. The third level of granularity is ethnographic: Martin and Sommerville discuss the relationship between the social structure of work and the technical structure of a system. They emphasise the importance of un-

[2]EPRSC is the UK Engineering and Physical Sciences Research Council: visit EPSRC at http://www.epsrc.org.uk

derstanding this relationship when designing dependable socio-technical systems. The fourth and last level is organisational: Andras and Charlton use abstract communications systems theory to demonstrate that the various adaptations and failure modes of organisations can be described in terms of the communications exchanged.

Guaranteeing dependability is the fifth and final section of the book. It deals in a practical manner with the task of ensuring that dependability is achieved – or understanding why it might not be achieved. Bryans and Arief address the topic of security. They adopt a human perspective on computer-based systems. Security is too often regarded as a purely technical issue, thereby leaving unprotected paths in the surrounding human system to be exploited by malicious attacks. Jackson tackles system weaknesses from an engineering point of view. In his approach, system failures are partly due to the way software engineers capture requirements. An intellectual structure for software development is proposed and discussed on the basis of a concrete example. But certifying that a system will behave dependably is also related to information gained *after* it is developed. Bloomfield and Littlewood consider critical systems certification. On the basis of a statistical analysis, they address the question "What happens when you want to certify a system and must combine arguments drawing on different statistical information"? The question of arguments is also addressed by Sujan, Smith and Harrison, who investigate the choice and combination of arguments in dependability claims.

This book could have been much longer and covered yet more topics. But it was never the goal to provide an exhaustive view of the structure of computer-based systems. Rather, the driving force was a desire to shed light on some issues that we considered particularly important. The editors hope that by bringing together varied topics with a common theme they have provided readers with a feel for the importance of interdisciplinarity in the design, commissioning and analysis of computer-based systems. Inevitably, practice changes only slowly. But we hope we have been able to offer a wider vision, showing how seemingly distinct concerns are sometimes closely interwoven, and revealing to practitioners of one discipline the relevance and importance of the problems addressed by another. Customers, stakeholders and users of computer-based systems may also benefit from an understanding of the underlying connections among different facets of their system.

The authors themselves have benefited from writing this book: it gave an increased opportunity to cross the boundaries of our disciplines and share views with researchers from different backgrounds. We are now more than ever convinced of the value of different perspectives derived from different backgrounds: this is where the importance of interdisciplinarity lies. We also hope that this book, beyond any direct influence of its contents, will disseminate a certain philosophy of undertaking science in the field of dependability. After all, through the knowledge it builds, the work done in dependability has a mission of contributing to the construction of our society's future. How could this be better achieved than through knowledge sharing and integration?

Acknowledgements:

Individual authors wish to express acknowledgements as follows

Burns and Baxter: The time band formulation owes much to the input of Colin Fidge and Ian Hayes. This particular description has benefited from comments of Denis Besnard and Cliff Jones.

Besnard: The author is grateful to Cliff Jones for his initial effort on Chapter 5 and his continuous help. Brian Randell, Gordon Baxter and Andrew Monk should also be thanked for their very constructive comments on earlier versions of this text.

Jackson: Talking and working with colleagues in the DIRC project and at the Open University has greatly helped my understanding of the matters discussed in Chapter 12. The irrigation problem has been extensively discussed with Cliff Jones and Ian Hayes, and is the subject of a jointly authored paper. I am grateful to Tom Maibaum for introducing me to Walter Vincenti's illuminating book. Cliff Jones, Ian Hayes and Denis Besnard read an earlier draft of this chapter and made many valuable comments.

Bloomfield and Littlewood: The work discussed in Chapter 13 was partially supported by the DISPO-2 (DIverse Software PrOject) Project, funded by British Energy Generation Ltd and BNFL Magnox Generation under the IMC (Industry Management Committee) Nuclear Research Programme under Contract No. PP/40030532. An earlier version of this paper was presented at the International Conference on Dependable Systems and Networks (DSN2003) in San Francisco.

Sujan, Smith and Harrison: Are grateful to Lorenzo Strigini and Michael Hildebrand for insightful comments and discussions.

As a closing word, we emphasise that the idea of dependability as an interdisciplinary scientific domain – an idea that underlies the variety of the chapters of this volume – rests firmly on the work of the authors. They provided the material for our message; they invested their time and expertise in the achievement of this book and should be thanked warmly. Behind the scenes lies a partner that made this collaboration possible: EPSRC. The Research Council made the gathering of the interdisciplinary team possible and thus provided the essential conditions for this book to be written. All of the authors and editors are grateful to EPSRC for the funding of DIRC ('Interdisciplinary Research Collaboration in Dependability of Computer-Based Systems').

The editors would also like to thank sincerely their long-suffering "production team" Joanne Allison and Joey Coleman without whom the chapters would probably have never made it into a book – certainly not one so well presented as this. Thanks are also due to our publisher Springer.

Denis Besnard
Cristina Gacek
Cliff B Jones
Newcastle upon Tyne, July 2005

Contents

Preface v

Contributors xi

Introduction

1 The role of structure: a dependability perspective
Cliff B Jones, Brian Randell 3

2 The role of structure: a software engineering perspective
Michael Jackson 16

System Properties

3 Structuring evolution: on the evolution of socio-technical systems
Massimo Felici 49

4 Time bands in systems structure
Alan Burns, Gordon Baxter 74

Human Components

5 Procedures, programs and their impact on dependability
Denis Besnard 91

6 Cognitive conflicts in dynamic systems
Denis Besnard, Gordon Baxter 107

Systems Descriptions

7 Architectural description of dependable software systems
Cristina Gacek, Rogério de Lemos 127

8 Computational diagrammatics: diagrams and structure
Corin Gurr 143

9 Ethnography and the social structure of work
David Martin, Ian Sommerville 169

10 Faults, errors and failures in communications: a systems theory perspective on organisational structure
Peter Andras, Bruce Charlton 189

Guaranteeing Dependability

11 Security implications of structure
Jeremy Bryans, Budi Arief 217

12 The structure of software development thought
Michael Jackson 228

13 On the use of diverse arguments to increase confidence in dependability claims
Robin Bloomfield, Bev Littlewood 254

14 Qualitative analysis of dependability argument structure
Mark A Sujan, Shamus P Smith, Michael D Harrison 269

Index 291

Contributors

Peter Andras
School of Computing Science
Claremont Tower
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU
peter.andras@ncl.ac.uk

Budi Arief
School of Computing Science
Claremont Tower,
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU
l.b.arief@ncl.ac.uk

Gordon Baxter
Department of Psychology
University of York
Heslington
York YO10 5DD
g.baxter@psych.york.ac.uk

Denis Besnard
School of Computing Science
Claremont Tower
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU
denis.besnard@ncl.ac.uk

Robin Bloomfield
Centre for Software Reliability
City University, London
Northampton Square
London EC1V 0HB
reb@csr.city.ac.uk

Alan Burns
Department of Computer Science
University of York
Heslington
York YO10 5DD
burns@cs.york.ac.uk

Jeremy Bryans
School of Computing Science
Claremont Tower
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU
jeremy.bryans@ncl.ac.uk

Bruce Charlton
School of Biology
Henry Wellcome building
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU
bruce.charlton@ncl.ac.uk

Rogério de Lemos
Computing Laboratory
University of Kent
Canterbury
Kent CT2 7NF
r.delemos@kent.ac.uk

Massimo Felici
School of Informatics
University of Edinburgh
2 Buccleuch Place
Edinburgh EH8 9LW
mfelici@inf.ed.ac.uk

Cristina Gacek
School of Computing Science
Claremont Tower
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU
cristina.gacek@ncl.ac.uk

Corin Gurr
School of Systems Engineering
The University of Reading
Whiteknights
Reading RG6 6AY
c.a.gurr@reading.ac.uk

Michael D Harrison
Institute of Research in Informatics
Devonshire building
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU
michael.harrison@ncl.ac.uk

Michael Jackson
101 Hamilton Terrace
London NW8 9QY
jacksonma@acm.org

Cliff B Jones
School of Computing Science
Claremont Tower
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU
cliff.jones@ncl.ac.uk

Bev Littlewood
Centre for Software Reliability
City University, London
Northampton Square
London EC1V 0HB
b.littlewood@csr.city.ac.uk

David Martin
Computing Department
South Drive
Lancaster University
Lancaster LA1 4WA
d.b.martin@lancaster.ac.uk

Brian Randell
School of Computing Science
Claremont Tower
University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU
brian.randell@ncl.ac.uk

Ian Sommerville
Computing Department
South Drive
Lancaster University
Lancaster LA1 4WA
is@comp.lancs.ac.uk

Shamus P Smith
Department of Computer Science
University of Durham
Science Laboratories, South Road
Durham DH1 3LE
shamus.smith@durham.ac.uk

Mark A Sujan
Department of Computer Science
University of York
Heslington
York YO10 5DD
sujan@cs.york.ac.uk

Introduction

Chapter 1

The role of structure: a dependability perspective

Cliff B Jones and Brian Randell

University of Newcastle upon Tyne

1 Introduction

Our concern is with the Dependability of Computer-based Systems; before tackling the question of structure, we take a look at each of the capitalised words. For now, an intuitive notion of **System** as something "that does things" will suffice; Section 2 provides a more careful characterization. The qualification **Computer-based** is intended to indicate that the systems of interest include a mix of technical components such as computers (running software) and human beings who have more-or-less clearly defined functions. It is crucial to recognise that the notion of (computer-based) system is recursive in the sense that systems can be made up of components which can themselves be analysed as systems. An example which is used below is the system to assess and collect Income Tax. One might at different times discuss the whole system, an individual office, its computer system, separate groups or even individuals. Our interest is in how systems which comprise both technical and human components are combined to achieve some purpose.

It is normally easier to be precise about the function of a technical system such as a program running on a computer than it is to discuss human components and their roles. But it is of interest to see to what extent notions of specifications etc. apply to the two domains. In attempting so to describe the role of a person within a system, there is no reductionist intention to de-humanise – it is crucial to observe where the differences between humans and machines must be recognised – but it is also beneficial to look at the roles of both sorts of component.

Section 2 emphasises how important it is to be clear, in any discussion, which system is being addressed: failure to do so inevitably results in fruitless disagreement. Of course, people can agree to shift the boundaries of their discussion but this should be made clear and understood by all participants.

The notion of systems being made up of other systems already suggests one idea of *Structure* but this topic is central to the current volume and more is said about it at the end of this section.

Let us first provide some introduction to the term *Dependability* (again, much more is said below – in Sections 3 and 4). Intuitively, *failures* are a sign of a lack of dependability. Section 3 makes the point that the judgement that a system has failed is itself complicated. In fact for such a judgement to be made requires other systems. (In order to be precise about the effects of the failures of one system on another, Section 3 adds the terms fault and error to our vocabulary.) Once we have been sufficiently precise about the notion of a single failure, we can characterize dependability by saying that no more than a certain level of failures occurs.

The interest in *Structure* goes beyond a purely analytic set of observations as to how different systems interact (or how sub-systems make up a larger system); a crucial question to be addressed is how the structure of a system can be used to inhibit the propagation of failures. Careful checks at the boundaries of systems can result in far greater dependability of the whole system.

2 Systems and their behaviour[1]

A **system**[2] is just an entity – to be of interest, it will normally interact – what it interacts with is other entities, i.e. other systems. We choose to use the term system to cover both artificially-built things such as computers (running software) and people (possibly groups thereof) working together – so we would happily talk about an "income tax office" as a system – one made up of humans and computers working together.

System and component boundaries can be very real, e.g. the cabinet and external connectors of a computer, or the skin of an individual human being. But they can also be somewhat or wholly arbitrary, and in fact exist mainly in the eye of the beholder, as then will be the structure of any larger system composed of such system components. This is particularly the case when one is talking about human organisations – although management structure plans may exist indicating how the staff are intended to be grouped together, how these groups are expected/permitted to interact, and what the intended function of each group and even individual is, these plans may bear little resemblance to actual reality. (Presumably the more "militaristic" an organisation, the more closely reality will adhere to the intended organisational structure – an army's structure largely defines the actual possibilities for exchanging information and effectively exercising control. On the other hand in, say, a university, many of the important communications and control "signals" may flow through informal channels, and cause somewhat uncertain effects, and the university's "real" structure and function may bear only a passing resemblance to that which is laid down in the university's statutes, and described in speeches by the Vice-Chancellor.) Regrettably, this can also be the case when talking about large software systems.

The choice to focus on some particular system is up to the person(s) studying it; furthermore, the same person can choose to study different systems at different times.

---

[1] The definitions given here are influenced by [1]; a more formal account of some of the terms is contained in [3].

[2] Definitions are identified using bold-face, with italics being used for stress.

It will normally be true that a system is made up of interacting entities which are the next lower level of system that might be studied. But the world is not neatly hierarchical: different people can study systems which happen to partially overlap. An electrician is concerned with the power supply to a whole building even if it houses offices with completely different functions.

What is important is to focus any discussion on *one* system at a time; most confusion in discussing problems with systems derives from unidentified disagreement about which system is being discussed. The utmost care is required when tracing the causes of failures (see Section 3) because of the need to look at how one system affects another, and this requires precision regarding the boundaries of each.

So, we are using the noun “system” to include hardware, software, humans and organisations, and elements of the physical world with its natural phenomena. What a system interacts with is its **environment** which is, of course, another system.

The **system boundary** is the frontier between the system being studied and its environment. Identification of this boundary is implicit in choosing a system to study – but to address the questions about *dependability* that we have to tackle, it is necessary to be precise about this boundary. To focus first on computer systems, one would say that the *interface* determines what the inputs and outputs to a system can be. It is clearly harder to do this with systems that involve people but, as mentioned above, the main problem is to choose which system is of current relevance. (For example, the fact that the employees in the income tax office might need pizza brought in when they work overtime is not the concern of someone filing a tax return.) We will then fix a boundary by discussing the possible interactions between the system (of current interest) and its environment. Viewed from the position of someone filing a tax return, the input is a completed document and the output is a tax assessment (and probably a demand for payment!).

Given our concern with *dependability,* the systems that are of interest to us are those whose boundaries can be discerned (or agreed upon) and for which there exists some useful notion of the system’s **function**, i.e. of what the system is *intended* to do. A deviation from that intended function constitutes a **failure** of the system.

When discussing artificial systems which have (at some time) to be constructed, it is useful to think in terms of their **specifications –** these will not of course always be written down; much less can one assume that they will be written in a formal notation. One can ask a joiner to build a bookcase on a particular wall without documenting the details. But if there is no specification the created thing is unlikely to be what the customer wanted. The more complex the system is, the truer this becomes. While there is a reasonably narrow set of interpretations of a “bookcase”, the class of, say, airline reservation systems is huge. (An issue, addressed below, is that of system “evolution” – this is again of great concern with things such as airline reservation systems.)

Specifications are certainly not restricted to technical systems: one can say what the income tax office is intended to do for the general public (or, separately, for government revenue collection) and each person in this office might have a job description. It is of course true that a computer program is more likely to produce the same results in the same condition (which might or might not satisfy its specification!) than a human being who can become tired or distracted. The difference be-

tween one person and another is also likely to be large. But none of this argues against looking at the humans involved in a system as having some expected behaviour. An income tax officer for example is expected to assess people's tax positions in accordance with established legislation.

Many human systems evolve rather than being built to a specification – an extreme example of such evolution is (each instance of) a family. Be that as it may, many systems that involve humans, such as hospitals, *do* have an intended function; it is thus, unfortunately, meaningful to talk of the possibility of a hospital failing, indeed in various different ways.

Systems (often human ones) can create other systems, whose behaviours can be studied. This topic is returned to in Section 7 after we have discussed problems with behaviour.

3 Failures and their propagation

Our concern here is with failures and their propagation between and across systems. We firstly look at individual breaches of intended behaviour; then we consider how these might be the result of problems in component systems; finally we review descriptions which document frequency levels of failure.

A system's function is described by a **functional specification** in terms of functionality and performance (e.g. that a reactor will be shut down within twenty seconds of a dangerous condition being detected). A **failure** is a deviation from that specification; a failure must thus at least in principle be visible at the boundary or interface of the relevant system. Giving a result outside the required precision for a mathematical function is a failure; returning other than the most recent update to a database would also be a failure, presumably technical in origin; assessing someone's income tax liability to be tenfold too high might be a failure resulting from human carelessness. A failure is visible at the boundary or interface of a system.

But failures are rarely autonomous and can be traced to defects in system components. In order to make this discussion precise, we adopt the terms error and fault whose definitions follow a discussion of the state of a system.

An important notion is that of the **internal state** of a system. In technical systems, the use of a state is what distinguishes a system from a (mathematical) function. To take a trivial example, popping a value from a stack will yield different results on different uses because the internal state changes at each use (notice that the internal state is not necessarily visible – it affects future behaviour but is internal). A larger example of a technical system is a stock control system which is *intended* to give different results at different times (contrast this with a function such as square root). This notion carries over to computer-based systems: a tax office will have a state reflecting many things such as the current tax rates, the forms received, the level of staffing, etc. Some of these items are not fully knowable at the interface for tax payers but they certainly affect the behaviour seen there.

Systems don't always do what we expect or want. We've made clear that such "problems" should be judged against some form of specification; such a specification will ideally be agreed and documented beforehand but might in fact exist only in, or

be supplemented by information in, the minds of relevant people (e.g. users, or system owners). A specification is needed in order to determine which system behaviours are acceptable versus those that would be regarded as **failures**. A failure might take the form of incorrect externally visible behaviour (i.e. output); alternatively, or additionally, such output might be made available too late, or in some cases even unacceptably early.

One system might be composed of a set of **interacting components**, where each component is itself a system. The recursion stops when a component is *considered to be* **atomic**: any further internal structure cannot be discerned, or is not of interest and can be ignored. (For example, the computers composing a distributed computing system might for many purposes be regarded as atomic, as might the individual employees of a computer support organisation.)

In tracing the causes of failures it is frequently possible to observe that there is a latent problem within the system boundary. This is normally describable as an erroneous state value. That is, there is something internal to the system under discussion which might later *give rise* to a failure. Notice that it is also possible that an error state will not become visible as a failure. A wrongly computed value stored within a computer might never be used in a calculation presented at the interface and in this case no failure is visible. An **error** is an unintended internal state which has the potential to cause a failure. An employee who is overly tired because she has worked too long might present the danger of a future failure of the system of which she is part.

One can trace back further because errors do not themselves arise autonomously; they in turn are caused by **faults**. It is possible that a wrongly stored value is the result of a failure in the memory of the computer: one would say that the error was caused by a hardware fault. (We see below that the chain can be continued because what is seen by the software as a hardware fault is actually a failure of that system.) In fact, such hardware faults rarely propagate in this way because protection can be provided at the interface – this is an essential message that is picked up below.

Whether or not an error will actually lead to a failure depends on two factors: (i) the structure of the system, and especially the nature of any redundancy that exists in it, and (ii) behavior of the system: the part of the state that contains an error may never be needed for service or the error may be eliminated before it leads to a failure. (For example, errors of judgment might be made inside a clinical testing laboratory – only those that remain uncorrected and hence lead to wrong results being delivered back to the doctor who requested a test will constitute failures of the laboratory.)

A failure thus occurs when an error 'passes through' the 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":

$$. . . \rightarrow \text{failure} \rightarrow \text{fault} \rightarrow \text{error} \rightarrow \text{failure} \rightarrow \text{fault} \rightarrow . . .$$

This chain can progress (i) from a component to the system of which it is part (e.g. from a faulty memory board to a computer), (ii) from one system to another separate system with which it is interacting (e.g. from a manufacturer's billing system to a customer's accounts system), and (iii) from a system to one that it creates

(e.g. from a system development department to the systems it implements and installs in the field).

Furthermore, from different viewpoints, there are also likely to be differing assumptions and understandings regarding the system's function and structure, and of what constitute system faults, errors and even failures – and hence dependability. In situations where such differences matter and need to be reconciled, it may be possible to appeal to some superior ("enclosing") system for a judgement, for example as to whether or not a given system has actually failed. However, the judgment of this superior system might itself also be suspect, i.e. this superior system might itself fail. This situation is familiar and well-catered for in the legal world, with its hierarchy in the UK of magistrates courts, crown courts, appeal courts, etc. But if there is no superior system, the disputing views will either remain unresolved, or be sorted out either by agreement or agreed arbitration, or by more drastic means, such as force.

So far, we have discussed single failures as though perfection (zero faults) were the only goal. Little in this life is perfect! In fact, one is normally forced to accept a "quality of service" description which might say that correct results (according to the specification) should be delivered in 99.999% of the uses of the system. Certainly where human action is involved, one must accept sub-optimal performance (see [6]).

It is possible to combine components of a given dependability to achieve a higher dependability providing there is sufficient **diversity**. This is again a question of structure. This brings up the issue of "multiple causes" of failures; several components might fail; if they do so together, this can lead to an overall failure; if the failure rate of the overall system is still within its service requirement, then this is unfortunate but not disastrous; if not (the overall system has failed to meet its service requirement), then either the design was wrong (too few layers of protection) or one of the components was outside its service requirement. (However, it may prove more feasible to change the service requirement than the system.)

A further problem is that of evolution (discussed more fully in Chapter 3 of this volume). A system which is considered to be dependable at some point in time might be judged undependable if the environment evolves. It is unfortunately true that the environments of computer-based systems will almost always evolve. As Lehman pointed out in [4], the very act of deploying most computer systems causes the human system around them to change. Furthermore, there can be entirely external changes such as new taxes which change the requirement for a system. It is possible to view this as a change in a larger system from the consequences of which recovery is required.

4 Dependability and dependence

The general, qualitative, definition of **dependability** is:

the ability to deliver service that can justifiably be trusted.

This definition stresses the need for some justification of trust. The alternate, quantitative, definition that provides the criterion for deciding if the service is dependable:

the ability to avoid service failures that are more frequent or more severe than is acceptable to the user(s).

It is usual to say that the dependability of a system should suffice for the dependence being placed on that system. The **dependence** of system A on system B thus represents the extent to which system A's dependability is (or would be) affected by that of System B.

The dependence of one system on another system can vary from total dependence (any failure of B would cause A to fail) to complete independence (B cannot cause A to fail). If there is reason to believe that B's dependability will be insufficient for A's required dependability, the former should be enhanced, or A's dependence reduced, or additional means of tolerating the fault provided.

The concept of dependence leads on to that of **trust,** which can conveniently be defined as *accepted dependence*. By "accepted" dependence, we mean the dependence (say of A on B) allied to a judgment that this level of dependence is acceptable. Such a judgment (made by or on behalf of A) about B is possibly explicit, and even laid down in a contract between A and B, but might be only implicit, even unthinking. Indeed it might even be unwilling – in that A has no option but to put its trust in B. Thus to the extent that A trusts B, it need not assume responsibility for, i.e. provide means of tolerating, B's failures (the question of whether it is capable of doing this is another matter). In fact the extent to which A fails to provide means of tolerating B's failures is a measure of A's (perhaps unthinking or unwilling) trust in B.

Dependability is in fact an integrating concept that encompasses the following main attributes:

- **availability**: readiness for correct service (e.g. of an Automatic Teller Machine to deliver money);
- **reliability:** continuity of correct service (e.g. provision of an accurate count of the number of currently free beds in a hospital);
- **safety:** absence of catastrophic consequences on the user(s) and the environment (e.g. due to an unintended missile launch);
- **integrity:** absence of improper system alterations (e.g. by corrupt insiders to their recorded credit ratings);
- **maintainability**: ability to undergo modifications, and repairs (e.g. of an accounting system when tax regulations are changed).

When addressing security, an additional attribute has great prominence, **confidentiality**, i.e. the absence of unauthorized disclosure of sensitive information (such as personal health records). **Security** is a composite of the attributes of confidentiality, integrity and availability, requiring the concurrent existence of a) availability for authorized actions only, b) confidentiality, and c) integrity (absence of system alterations which are not just improper but unauthorized).

Many means have been developed to attain the various attributes of dependability. These means can be grouped into four major categories:

- **fault prevention**: means to prevent the occurrence or introduction of faults;
- **fault tolerance**: means to avoid service failures in the presence of faults;
- **fault removal**: means to reduce the number and severity of faults;

- **fault forecasting**: means to estimate the present number, the future incidence, and the likely consequences of faults.

Fault prevention and fault tolerance aim to provide the ability to deliver a service that can be trusted, while fault removal and fault forecasting aim to reach confidence in that ability by justifying that the functional and the dependability specifications are adequate and that the system is likely to meet them.

Another grouping of the means is the association of (i) fault prevention and fault removal into **fault avoidance**, i.e. how to *aim for* fault-free systems, and of (ii) fault tolerance and fault forecasting into **fault acceptance**, i.e. how to *live with* systems that are subject to faults.

In general, the achievement of adequate dependability will involve the combined use of fault prevention, removal and tolerance, while fault forecasting will be needed to demonstrate that this achievement has indeed been gained.

5 On structure

We have already made it clear that one can study different systems at different times; and also that overlapping (non-containing) systems can be studied. We now want to talk about "structure" itself.

A description of a system's *structure* will identify its component systems, in particular their boundaries and functions, and their means of interaction. One can then go further and describe the structure of these systems, and so on. Take away the structure and one just has a set of unidentified separate components – thus one can regard the structure as what, in effect, enables the relevant components to interact appropriately with each other, and so constitute a system, and cause it to have behaviour. (A working model car can be created by selection and interconnection of appropriate parts from a box of Lego parts – without such structuring there will be no car, and no car-like behaviour. The reverse process of course destroys the car, even though all the pieces from which it was created still exist.) For this reason we say that the **structure** of a system is what enables it to generate the system's behaviour, from the behaviours of its components.

There are many reasons to choose to create, or to identify, one structure over another. A key issue for dependability is the ability of a system to "contain errors" (i.e. limit their flow). A structuring is defined as being **real** to the extent that it correctly implies what interactions cannot, or at any rate are extremely unlikely to, occur. (A well-known example of physical structuring is that provided by the watertight bulkheads that are used within ships to prevent, or at least impede, water that has leaked into one compartment (i.e. component) flowing around the rest of the ship. If these bulkheads were made from the same flimsy walls as the cabins, or worse existed solely in the blueprints, they would be irrelevant to issues of the ship's seaworthiness, i.e. dependability.) Thus the extent of the reality, i.e. strength, of a system's

structuring determines how effectively this structuring can be used to provide means of **error confinement**[3].

Two viewpoints are that of the system creator and that of someone studying an extant system. To some extent these are views of "structure" as a verb and as a noun (respectively). When creating either a technical system or a human organisation we structure a set of components, i.e. we identify them and determine how they should interact – these components might already exist or need in turn to be created. When observing a system, we try to understand how it works by recognising its components and seeing how they communicate (see Chapter 10 of the current volume).

There are many reasons to choose one structure over another. If one were creating a software system, one might need to decompose the design and implementation effort. It might be necessary to design a system so that it can be tested: hardware chips use extra pins to detect internal states so that they can be tested more economically than by exhaustively checking only the behaviour required at the normal external interface.

But there is one issue in structuring which can be seen to apply both to technical and computer-based (or **socio-technical)** systems and that is the design criterion that the structure of components can be used to "contain" (or limit the propagation of) failures. The role of an accountancy audit is to stop any gross internal errors being propagated outside of a system; the reason that medical instruments are counted before a patient is sewn up after surgery is to minimise the danger that anything is left behind; the reason a pilot reports the instructions from an ATC is similarly to reduce the risk of misunderstanding. In each of these cases, the redundancy is there because there could be a failure of an internal sub-system. In the case of a military unit, the internal fault can be the death of someone who should be in charge and the recovery mechanism establishes who should take over. The layers of courts and their intricate appeals procedures also address the risk – and containment – of failure.

Probably derived from such human examples, the notion of a "Recovery Block" [5] offers a technical construct which both checks that results are appropriate and does something if they are not.

Although containment of failures might have first been suggested by human examples, such as the rules adopted by banks in the interests of banking security, it would appear that the design of technical systems goes further in the self-conscious placing of structural boundaries in systems. Indeed, Chapter 5 of this volume discusses the role of procedures and suggests that the main stimulus for their creation

[3] Other important characteristics of *effective* structuring, whether of technical or socio-technical systems, relate to what are termed in the computer software world **coupling** (the number and extent of the relationships between software components) and **cohesion** (the degree to which the responsibilities of a single component form a meaningful unit) – other things being equal coupling should be as low, and cohesion as high as possible, and to the **appropriateness** of each component's functional specification. (Appropriateness might take in relative simplicity, adherence to standards, existence of commercially-available examples, etc.) Together, these characteristics will affect the performance of the system, and the ease with which it can be constructed, and subsequently evolve, in particular by being modified in response to changes in its environment and the requirements placed on it.

and/or revision is when errors are found in the structure of existing human systems (cf. Ladbroke Grove [2], Shipman enquiry [7]).

6 Human-made faults

Throughout the preceding discussion we have – especially via our choice of examples – been drawing parallels between technical, social, and socio-technical systems, and attempting to minimize their differences with regard to issues of dependability. But of course there are *major* differences. One might well be able to predict with considerable accuracy the types of fault that at least a relatively simple technical system might suffer from, and the likely probability of their occurrence. The structure of such a system may well be both simple and essentially immutable. However, when human beings are involved, they might not only be very creative in finding new ways to fail, or to cope with failures elsewhere, but also in violating the assumed constraints and limitations on information flow within, i.e. in affecting the assumed structure of the system to which they are contributing. (And when one takes into account the possibility of humans trying to defeat or harm a system deliberately, the situation becomes yet more complex.) But with a complex system, leave alone a computer-based system, our only way of having any hope of properly understanding and perhaps planning its behaviour is by assuming that it has some given, though flexible and evolving, structure. This structure is often defined by fixing processes or procedures (see Chapter 5).

The definition of faults includes absence of actions when actions should be performed, i.e. **omission faults**, or simply **omissions**. Performing wrong actions leads to **commission faults**. We distinguish two basic classes of human-made (omission and commission) faults according to the *objective* of the human in question, who might either be a developer or one of the humans interacting with the system during its use: (i) *non-malicious faults*, introduced without malicious objectives, (either *non-deliberate* faults that are due to *mistakes,* that is, *unintended actions* of which the developer, operator, maintainer, etc. is not aware, or *deliberate* faults that are due to well-intentioned *bad decisions)*, and (ii) *malicious faults*, introduced either during system development with the objective of causing harm to the system during its use, or directly during use.

Not *all* mistakes and bad decisions by *non-malicious* persons are accidents. Some very harmful mistakes and very bad decisions are made by persons who lack professional competence to do the job they have undertaken. The question of how to recognize incompetence faults becomes important when a mistake or a bad decision has consequences that lead to economic losses, injuries, or loss of human life. In such cases, independent professional judgment by a board of inquiry or legal proceedings in a court of law are likely to be needed to decide if professional malpractice was involved. This again relates to the presence of processes or procedures: there is a distinction between a mistake in a procedure (which is followed) and deviation from a procedure.

Human-made efforts have also failed because a team or an entire organisation did not have the organisational competence to do the job. A good example of organisa-

tional incompetence is the development failure of the AAS system that was intended to replace the aging air traffic control systems in the USA [8].

Malicious human-made faults are introduced with the malevolent objective of altering the functioning of a system during use. The goals of such faults are: (i) to disrupt or halt service, causing **denials of service**, (ii) to access confidential information, or (iii) to improperly modify the system.

As systems become more pervasive, so it seems do attempts by intruders (ranging from amateur hackers to well-resourced and highly competent criminals and terrorist organisations) and corrupt insiders to cause system failures of various kinds. An open question is to what extent can the inventiveness that is represented by such attempts be defeated by pre-designed automated defences (e.g. structuring) rather than have to rely on similar human inventiveness of the part of the system defenders – in particular, to what extent will the planned (or assumed) system structure aid the achievement of dependability in the face of such threats, or instead impede the task of understanding how a system was caused to fail via some particularly devious act that in effect "evaded" the structuring.

7 Systems that create other systems

Reference has been made above to the fact that one possibility relevant to issues of system structure is that one system creates another – this section takes a closer look at this issue. It is simple to see how a failure in a component manifests itself as a fault in a larger system; this fault (if not tolerated) might result in a failure of the larger system. But it is possible for one system to create another. For example, a compiler creates object code and a development team creates programs. A failure in the creating system can give rise to a fault in the created system. This fault may or may not result in a failure of the created system. For example

- A fault in an automatic production line might create faulty light bulbs;
- A buggy compiler might introduce bugs into the object code of a perfect (source) program;
- A design team might design a flawed program.

Faults may be introduced into the system that is being developed by its environment, especially by human developers, development tools, and production facilities. Such development faults may contribute to partial or complete development failures, or they may remain undetected until the use phase. A complete **development failure** causes the development process to be terminated before the system is accepted for use and placed into service. There are two aspects of development failures (i) *budget failure*, when the allocated funds are exhausted before the system passes acceptance testing, and (ii) *schedule failure*, when the projected delivery schedule slips to a point in the future where the system would be technologically obsolete or functionally inadequate for the user's needs.

The principal causes of development failures are: incomplete or faulty specifications, an excessive number of user-initiated specification changes, inadequate design with respect to functionality and/or performance goals, too many development faults, inadequate fault removal capability, prediction of insufficient dependability, and

faulty estimates of development costs. All are usually due to an underestimate of the complexity of the system to be developed.

It is to be expected that faults of various kinds will affect the system during use. The faults may cause unacceptably degraded performance or total failure to deliver the specified service. Such service failures need to be distinguished from **dependability failures**, which are regarded as occurring when the given system suffers service failures more frequently or more severely than is acceptable to the user(s). For this reason a **dependability specification** can be used to state the goals for each attribute: availability, reliability, safety, confidentiality, integrity, and maintainability. By this means one might, for example, agree beforehand how much system outage would be regarded as acceptable, albeit regrettable, and hence not a cause for recrimination with the supplier.

Underlying any decisions related to system development and deployment will be a set of (quite possibly arbitrary or unthinking) assumptions concerning (i) the effectiveness of the various means that are employed in order to achieve the desired levels of dependability, and indeed (ii) the accuracy of the analysis underlying any predictions concerning this achievement.

This is where the important concept of coverage comes into play, **coverage** being defined as the representativeness of the situations to which the system is subjected during its analysis compared to the actual situations that the system will be confronted with during its operational life. There are various forms of coverage – for example, there can be imperfections of fault tolerance, i.e. a lack of *fault tolerance coverage,* and fault assumptions that differ from the faults really occurring in operation, i.e. a lack of *fault assumption coverage*. In particular, and perhaps most difficult to deal with of all, there can be inaccurate assumptions, either prior to or during system use, about a system's designed or presumed structure – the trouble with such assumptions is that they tend to underpin all other assumptions and analyses.

8 Summary

Any development aimed at producing a dependable system should pay careful attention to issues of structuring. Any old structuring will not do – poor structuring can harm system performance, and impede system maintenance and evolution. But *weak* structuring can directly impair dependability. Structuring is in fact not an option – it would seem that the only way that humans can recognise entities and attempt to cope with complexity is by presuming – and then relying on – structure. The problem is to ensure that there is an effective reality to back up such presumptions, and that this reality can survive and evolve as needed for the successful continued deployment of the system.

We have attempted to maximize the use of notions from technical systems on whole (computer-based) systems; this is in no way intended to deny or ignore the differences between the ways in which human "components" and technical components contribute to the dependability problems, and solutions, of computer-based systems. However it does, we believe, allow a number of useful general issues to be identified and addressed.

References

[1] Avizienis A, Laprie J-C, Randell B, Landwehr C (2004) Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, pp 11–33

[2] Rt Hon Lord Cullen QC (2000) The Ladbroke Grove Rail Enquiry, HSE Books, see http://www.pixunlimited.co.uk/pdf/news/transport/ladbrokegrove.pdf

[3] Jones Cliff B, A Formal Basis for some Dependability Notions (2003) Formal Methods at the Crossroads: from Panacea to Foundational Support. In: Aichernig Bernhard K, Maibaum Tom (eds) Springer Verlag, Lecture Notes in Computer Science, vol. 2757 pp191–206

[4] Lehman M, Belady LA, (1985) (eds) Program evolution: processes of software change, Academic Press, APIC Studies in Data Processing No. 27, ISBN 012442441-4

[5] Randell B (1975) System Structure for Software Fault Tolerance, IEEE Trans. on Software Engineering, vol. SE-1, no. 2, pp.220-232

[6] J. Reason (1990) Human Error. Cambridge University Press, ISBN 0521314194

[7] Dame Janet Smith QC (2005) Sixth Report: Shipman – The Final Report, HSE Books, see http://www.the-shipman-inquiry.org.uk/finalreport.asp

[8] US Department of Transportation (1998) Audit Report: Advance Automation System, Report No. AV-1998-113, US Department of Transportation, Office of Inspector General

Chapter 2

The role of structure: a software engineering perspective

Michael Jackson

Independent Consultant

1 Introduction

The focus of this chapter is on the dependability of software-intensive systems: that is, of systems whose purpose is to use software executing on a computer to achieve some effect in the physical world. Examples are: a library system, whose purpose is to manage the loan of books to library members in good standing; a bank system whose purpose is to control and account for financial transactions including the withdrawal of cash from ATMs; a system to control a lift, whose purpose is to ensure that the lift comes when summoned by a user and carries the user to the floor desired; a system to manage the facilities of a hotel, whose purpose is to control the provision of rooms, meals and other services to guests and to bill them accordingly; a system to control a radiotherapy machine, whose purpose is to administer the prescribed radiation doses to the patients, accurately and safely.

We will restrict our attention to functional dependability: that is, dependability in the observable behaviour of the system and in its effects in the world. So safety and security, which are entirely functional in nature, are included, but such concerns as maintainability, development cost, and time-to-market are not. The physical world is everything, animate or inanimate, that may be encountered in the physical universe, including the artificial products of other engineering disciplines, such as electrical or mechanical devices and systems, and human beings, who may interact with a system as users or operators, or participate in many different ways in a business or administrative or information system. We exclude only those systems whose whole subject matter is purely abstract: a model-checker, or a system to solve equations, to play chess or to find the convex hull of a set of points in three dimensions.

Within this scope we consider some aspects of software-intensive systems and their development, paying particular attention to the relationship between the software executed by the computer, and the environment or problem world in which its effects will be felt and its dependability evaluated. Our purpose is not to provide a comprehensive or definitive account, but to make some selected observations. We discuss and illustrate some of the many ways in which careful use of structure, in

both description and design, can contribute to system dependability. Structural abstraction can enable a sound understanding and analysis of the problem world properties and behaviours; effective problem structuring can achieve an informed and perspicuous decomposition of the problem and identification of individual problem concerns; and structural composition, if approached bottom-up rather than top-down, can give more reliable subproblem solutions and a clearer view of software architecture and more dependable attainment of its goals.

2 Physical structure

In the theory and practice of many established engineering branches—for example, in civil, aeronautical, automobile and naval engineering—structure is of fundamental importance. The physical artifact is designed as an assemblage of parts, specifically configured to withstand or exploit imposed mechanical forces. Analysis of designs in terms of structures and their behaviour under load is a central concern, and engineering education teaches the principles and techniques of analysis. Engineering textbooks show how the different parts of a structure transmit the load to neighbouring parts and so distribute the load over the whole structure. Triangular trusses, for example, distribute the loads at their joints so that each straight member is subjected only to compression or tension forces, and not to bending or twisting, which it is less able to resist. In this way structural abstractions allow the engineer to calculate how well the designed structure will withstand the loads it is intended to bear.

When a bridge or building fails, investigation may reveal a clear structural defect as a major contributing cause. For example, two atrium walkways of the Kansas City Hyatt Hotel failed in 1981, killing 114 people [21]. The walkways were supported, one above the other, on transverse beams, which in turn were supported by hanger rods suspended from the roof. The original design provided for each hanger rod to give support to both the upper and the lower walkway, passing through the transverse beam of the upper walkway as shown in the left of Fig. 1. Because this design was difficult to fabricate, the engineer accepted the modification shown in the right of the figure. The modification was misconceived: it doubled the forces acting on the transverse beams of the upper walkway, and the retaining nuts on the hanger rods tore through the steel beams, causing the collapse.

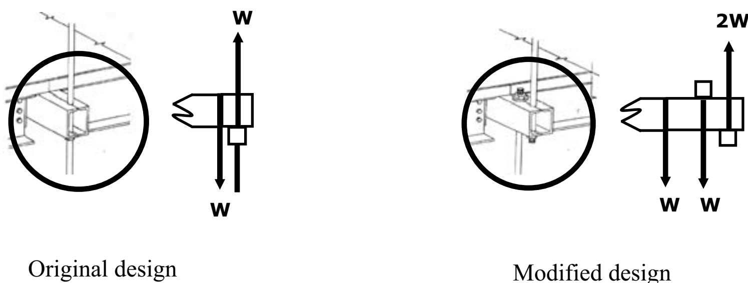

Fig. 1. Kansas City Hyatt: collapse of walkways

The modification should have been rejected, and the engineer responsible for accepting it rightly lost his licence.

The ability to analyse structure in this kind of way allows engineers to design products with necessary or desirable properties in the large while expecting those properties to carry over successfully from the structural abstraction to the final physical realisation of the design. As the Kansas City disaster demonstrates, the established branches of engineering have an imperfect record in the dependability of their products. But it is still a record that software developers should envy. Whether by explicit quantitative analysis, or by applying less formal knowledge derived from a long history of carefully recorded successful experience, engineers have been able to design a large range of successful structures whose properties were reliably predicted from their designs. Roman engineers, for example, exploited the circular arch structure to build aqueducts like the Pont du Gard at Nîmes, which is still standing after nearly two thousand years, and bridges like the Pons Fabricius over the Tiber in Rome, built in 62BC as a road bridge and still in use today by pedestrians.

This possibility, of predicting the properties of the final product from analysis of a structural abstraction, depends on two characteristics of the physical world at the scale of interest to engineers. First, the physical world is essentially continuous: Newton's laws assure the bridge designer that unforeseen addition of parts of small mass, such as traffic sign gantries, telephone boxes and waste bins, cannot greatly affect the load on the major components of the bridge, because their mass is much larger. Second, the designer can specify implementation in standard materials whose mechanical properties (such as resistance to compression or tension or bending) are largely known, providing assurance that the individual components of the final product will not fail under their designed loads.

In the design of software-intensive systems, too, there is some opportunity to base development on structural abstractions that allow properties of the implemented system to be determined or analysed at the design stage. This is true especially in distributed systems, such as the internet, in which the connections among the large parts of the structure are tightly constrained to the physical, material, channels provided by the designed hardware configuration. For such a system, calculations may be made with some confidence about certain large properties. Traffic capacity of a network can be calculated from the configuration of a network's paths and their bandwidths. A system designed to achieve fault-tolerance by server replication can be shown to be secure against compromise of any *m* out of *n* servers. A communications system can be shown to be robust in the presence of multiple node failures, traffic being re-routed along the configured paths that avoid the failed nodes.

Design at the level of a structural abstraction necessarily relies on assumptions about the low-level properties of the structure's parts and connections: the design work is frustrated if those assumptions do not hold. In the established engineering branches the assumptions may be assumptions about the properties of the materials of which the parts will be made, and how they will behave under operational conditions. For example, inadequate understanding of the causes and progression of metal fatigue in the fuselage skin caused the crashes of the De Havilland Comet 1 in the early 1950s. The corners of the plane's square passenger windows provided sites of

stress concentration at which the forces caused by the combination of flexing in flight with compression and decompression fatally weakened the fuselage.

In the case of the Comet 1, a small-scale design defect—the square windows—frustrated the aims of an otherwise sound large-scale design. In a software example of a similar kind, the AT&T long-distance network was put out of operation for nine hours in January 1990 [27; 31]. The network had been carefully designed to tolerate node failures: if a switching node crashed it sent an 'out-of-service' message to all neighbouring nodes in the network, which would then route traffic around the crashed node. Unfortunately, the software that handled the out-of-service message in the receiving node had a programming error. A *break* statement was misplaced in a C *switch* construct, and this fault would cause the receiving node itself to crash when the message arrived. In January 1990 one switch node crashed, and the cascading effect of these errors brought down over 100 nodes.

This kind of impact of the small-scale defect on a large-scale design is particularly significant in software-intensive systems, because software is discrete. There are no Newton's laws to protect the large-scale components from small-scale errors: almost everything depends inescapably on intricate programming details.

3 Small-scale software structure

In the earliest years of modern electronic computing attention was focused on the intricacies of small-scale software structure and the challenging task of achieving program correctness at that scale.

It was recognised very early in the modern history of software development [19] that program complexity, and the need to bring it under control, presented a major challenge. Programs were usually structured as flowcharts, and the task of designing a program to calculate a desired result was often difficult: even quite a small program, superficially easy to understand, could behave surprisingly in execution. The difficulty was to clarify the relationship between the static program text and the dynamic computation evoked by its execution. An early approach was based on checking the correctness of small programs [32] by providing and verifying assertions about the program state at execution points in the program flowchart associated with assignment statements and tests. The overall program property to be checked was that the assertions "corresponding to the initial and stopped states agree with the claims that are made for the routine as a whole"—that is, that the program achieved its intended purpose expressed as a precondition and postcondition.

The invention of the closed subroutine at Cambridge by David Wheeler made possible an approach to programming in which the importance of flowcharts was somewhat diminished. Flowcharts were not an effective design tool for larger programs, and subroutines allowed greater weight to be placed on structural considerations. M V Wilkes, looking back on his experiences in machine-language programming on the EDSAC in 1950 [36], writes:

> *A program structured with the aid of closed subroutines is much easier to find one's way about than one in which use is made of jumps from one block of*

code to another. A consequence of this is that we did not need to draw elaborate flow diagrams in order to understand and explain our programs.

He gives an illustration:

... the integration was terminated when the integrand became negligible. This condition was detected in the auxiliary subroutine and the temptation to use a global jump back to the main program was resisted, instead an orderly return was organised via the integration subroutine. At the time I felt somewhat shy about this feature of the program since I felt that I might be accused of undue purism, but now I can look back on it with pride.

Programming in terms of closed subroutines offered an opportunity to develop a discipline of program structure design, but in the early years relatively little work was done in this direction. Design in terms of subroutines was useful but unsystematic. Among practitioners, some kind of modular design approach eventually became a necessity as program size increased to exploit the available main storage, and a conference on modular programming was held in 1968 [7].

For most practising programmers, the structuring of program texts continued to rely on flowcharts and go to statements, combined with an opportunistic use of subroutines, until the late 1960s. In 1968 Dijkstra's famous letter [10] was published in the Communications of the ACM under the editor's heading "Go To Statement Considered Harmful". IBM [1] and others soon recognised both scientific value and a commercial opportunity in advocating the use of Structured Programming. Programs would be designed in terms of closed, nested control structures. The key benefit of structured programming, as Dijkstra explained, lay in the much clearer relationship between the static program text and the dynamic computation. A structured program provides a useful coordinate system for understanding the progress of the computation: the coordinates are the text pointer and the execution counters for any loops within which each point in the text is nested. Dijkstra wrote:

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process.

Expressing the same point in different words, we may say that a structured program places the successive values of each component of its state in a clear context that maps in a simple way to the progressing state of the computation. The program structure tree shows at each level how each lexical component—elementary statement, loop, if-else, or concatenation—is positioned within the text. If the lexical component can be executed more than once, the execution counters for any enclosing loops show how its executions are positioned within the whole computation. This structure allowed a more powerful separation of concerns than is possible with a flowchart. Assertions continued to rely ultimately on the semantic properties of elementary assignments and tests; but assertions could now be written about the larger lexical components, relying on their semantic properties. The programmer's understanding of each leaf is placed in a structure of nested contexts that reaches all the way to the root of the program structure tree.