

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 behaviour¹

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”:

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

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

[Omitted]

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**.

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 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

[Omitted]

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

[Omitted]

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