Table of Contents and Extract from “The Role of Structure in Dependable Systems”

Michael Jackson

Independent Consultant

Table of Contents

Introduction
Physical Structure
Small-Scale Software Structure
The Product and Its Environment
The Formal Machine and the Non-Formal World
Simple Structures Describing the Problem World
Problem Decomposition into Subproblems
Structure Within Subproblems
Composition Concerns
Top-Down and Bottom-Up Architecture
Uniform Architectures
Component Relationships
Concluding Remarks
References

Extract

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.

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.

●●● ●●●

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.

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.

●●● ●●●

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.

••• •••

This formal work was fruitful in its own area of program development. But its very success contributed to an unfortunate neglect of larger design tasks. The design of systems—seen as large assemblages of programs intended to work cooperatively, especially in commercial or administrative data processing—or even of a very large program, was often regarded as uninteresting. Either it was nothing more than a simple instance of the appealingly elegant principle of stepwise refinement or recursive decomposition, or else it was much too difficult, demanding the reduction to order of what to a casual glance seemed to be merely a mass of unruly detail. Both views were mistaken. Realistic systems rarely have specifications that can be captured by terse expressions inviting treatment by formal refinement. And although some parts of some data processing systems did indeed seem to present a mass of arbitrary detail—for example, payroll rules originating in long histories of fudged legislation and compromises in negotiations between management and unions—the unruly detail more often reflected nothing other than the richness of the natural and human world with which the system must inevitably deal.

The Product and Its Environment

Software engineering has suffered from a deep-seated and long-standing reluctance to pay adequate attention to the environment of the software product. One reason lies in the origins of the field. A 'computer' was originally a person employed to perform calculations, using a mechanical calculating machine, often for the

construction of mathematical tables or the numerical solution of differential equations in such areas as ballistics. The ‘electronic computer’ was a faster, cheaper, and more reliable way of performing such calculations.

The use of computers interacting more intimately with its environment to bring about desired effects there—that is, the use of software-intensive systems—was a later development. Dependability of a software-intensive system is not just a property of the software product. It is a property of the product in its environment or, as we shall say, in its problem world. Dependability means dependability in satisfying the system’s purposes; these purposes are located, and their satisfaction must be evaluated, not in the software or the computer, but in the problem world into which it has been installed.

In considering and evaluating many engineering products—such as aeroplanes and cars—it seems natural to think of dependability as somehow intrinsic to the product itself rather than as residing in the relationship between the product and its environment. But this is misleading. The environment of a car, for example, includes the road surfaces over which it must travel, the fuel that will be available to power it, the atmospheric conditions in which the engine must run, the physical dimensions, strength and dexterity that the driver is likely to possess, the weight of luggage or other objects to be carried along with the passengers, and so on.

●●● ●●●

By contrast, the dependability and quality of some other engineering products, such as bridges, tall buildings and tunnels, is very obviously evaluated in terms of their relationship to their specific individual environments. The designer of a suspension bridge over a river must take explicit account of the properties of the environment: the prevailing winds, the possible road traffic loads, the river traffic, the currents and tides, the geological properties of the earth on which the bridge foundations will stand, and so on.

●●● ●●●

The environment or problem world is especially important for software-intensive systems with a need for high dependability. One reason is that such systems very often have a unique problem world. Each nuclear power plant, or large medical radiation therapy installation, is likely to have its own unique properties that are very far from completely standard. Another reason is that the system may be highly automated: in a heart pacemaker there is no operator to take action in the event of a crisis due to inappropriate software behaviour.

The Formal Machine and the Non-Formal World

The scope of a software-intensive system is shown by the problem diagram Figure XXX.2.

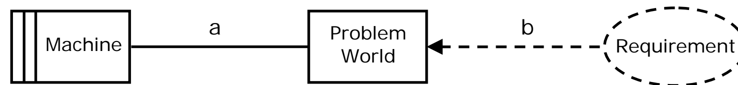

Fig. XXX.2. A Problem Diagram: Machine, Problem World, and Requirement

The machine is connected to the problem world by an interface *a* of shared phenomena—for example, shared events and states. The *requirement* captures the purpose of the system—the effect to be achieved in the problem world, expressed in terms of some problem world phenomena *b*. Because the purposes to be served by a system are almost always focused on parts of the problem world that lie some distance from the interface between the machine and the world, the requirement phenomena *b* are almost always distinct from the phenomena at *a* which the machine can monitor or control directly.

The success of the system therefore depends on identifying, analysing, respecting and exploiting the given properties of the non-formal world that connect the two sets of phenomena. For a lift control system, dependable lift service must rely on the causal chains that connect the motor state to the winding drum, the drum to the hoist cables, the cables to the position of the lift car in the shaft, and the lift car position to the states of the sensors at the floors. A library administration system must depend on the physical correspondence between each book and the bar-coded label fixed into it, on the possible and expected behaviours of the library members and staff, on the physical impossibility of a book being simultaneously out on loan and on its shelf in the library, and on other properties of its problem world.

The concerns of the developers of such a system must therefore encompass both the machine and the problem world. This enlarged scope presents a special difficulty. A central aspect of the difficulty was forcefully expressed by Turski [33]:

> *"Thus, the essence of useful software consists in its being a constructively interpretable description of properties of two ... structures: [formal] hardware and [non-formal] application domain, respectively. ...*
>
> *"Thus, software is inherently as difficult as mathematics where it is concerned with relationships between formal domains, and as difficult as science where it is concerned with description of properties of non-formal domains. Perhaps, software may be said to be more difficult than either mathematics or science, as in most really interesting cases it combines the difficulties of both."*

The difficulty of dealing with the non-formal problem world arises from the unbounded richness, at the scale that concerns us, of the physical and human world. In forming structural abstractions of the software itself we are confronted by the task of finding the most useful and appropriate structures, analysing them, and composing them into a software product. In structuring the problem world we are confronted also by the difficult task of formalising a non-formal world.

●●● ●●●

The difficulty of formalisation of a non-formal problem world is common to all engineering disciplines: a structural abstraction of a physical reality is never more than an approximation to the reality. But in software-intensive systems the formal nature of the machine, combined with its slender interface to the non-formal problem world, increases the difficulty dramatically. The developers must rely on their assumptions about the non-formal problem world, captured in formal descriptions, to specify the machine behaviour that is to satisfy the requirement, and as the desired level of automation rises, those assumptions must necessarily become stronger.

The problem world formalisation, then, is determinative of the system. Because the system behaviour must be designed specifically to interact with the problem world, and the developer's understanding of the properties and behaviour of that world are expressed in the formal model, any defect in the model is very likely to give rise to defective system behaviour. This is not true of a system in which human discretion can play a direct part in the execution of the 'machine'. For example, an airline agent can use common sense to override the consequences of a defective model by allowing a passenger holding a boarding card for a delayed flight to use it to board another flight. Nor is the model of the problem world, or of the product, determinative in the established branches of engineering. Knowing that their models are only approximations, structural designers routinely over-engineer the product, introducing safety factors in accordance with established design precedent and statutory codes. In this way they can avoid placing so much confidence in their model that it leads to disaster.[1] The developer of a software-intensive system has no such safety net to fall back on. The formal machine has no human discretion or common sense, and a discrete system aiming at a high degree of automation offers few opportunities for improving dependability by judicious over-engineering. If the reality of the problem world is significantly different from the developer's assumptions, then the effects of the system are likely to be significantly different from the requirements. There is no reason to expect the deviation to be benevolent.

---

[1] The famous collapse of the Tacoma Narrows bridge in 1940 [15] can be attributed to exactly such overconfidence. Theodore Condron, the engineer employed by the finance corporation, pointed out that the high ratio of span to roadway width went too far beyond currently established precedent, and recommended that the roadway be widened from 39 feet to 52 feet. But other notable engineers, relying on the designer's deflection theory model, persuaded Condron to withdraw his objections, and the bridge was built with the fatal defect that led to its total collapse six months after construction.

••• •••

Concluding Remarks

The most important positive factor for system dependability is the availability and use of an established body of knowledge about systems of the kind that is to be built. This allows the developer to practice normal design, where the developer is concerned to make a relatively small improvement or adaptation to a standard product design satisfying a standard set of requirements. The structure of the product is well known, the problem world properties are essentially standard, the expectations of the product's users are well established and understood. If the design cleaves closely to the established norms there is good reason to expect success. In radical design, by contrast, there is no such established standard of design, requirements and problem world properties to draw on, and the designer must innovate. There can then be 'no presumption of success' [34].

One factor militating strongly against the dependability of software-intensive systems is the proliferation of features. A sufficiently novel combination of features, even if each feature individually is quite well understood, places the development task firmly in the class of radical design in respect of the subproblem composition task. A vital part of the knowledge embodied in a specialised normal design practice is knowledge of the necessary combination of functionalities. A car designer knows how to compose the engine with the gearbox, how to fit the suspension into the body, and how to interface the engine with its exhaust system. A designer confronted with a novel combination of features can not draw on normal design knowledge in composing them.

In the development of software-intensive systems, whether the task in hand is normal or radical design, a pervasive precondition for dependability is structural clarity. The avoidance of faults depends on successful structuring in many areas. Good approximations must be made to problem world properties. Structural compositions must accommodate the composed parts fully without distorting or obscuring the individual structures. Architectural relationships among subproblem machines must respect their precedence and relative criticality.

An aspect of dependability that has so far been entirely ignored in this chapter is the social context in which the development takes place. It is worth remarking here that normal design can evolve only over many years and only in a specialised community of designers who are continually examining each others' designs and sharing experience and knowledge. The established branches of engineering have been able to improve the dependability of their products only because their practitioners are highly specialised and because—as the most casual glance at examples [13, 20] will show— their educational and research literature is very sharply focused.

There are some such specialised communities in the software world, gathering regularly at specialised conferences. The long-term goal of improving dependability in software-intensive systems could be well served by continuing the

purposeful growth of such specialised communities, and embarking on the creation of new ones, each focused on a particular narrowly-defined class of system or subproblem.