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.

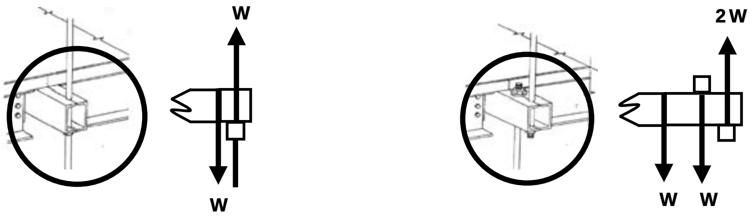
We will restrict our attention to functional dependability: that is, dependability in the observable behaviour of the system and in its effects in the world. So safety and security, which are entirely functional in nature, are included, but such concerns as maintainability, development cost, and time-to-market 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.



Original design

Modified design

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.

The set of possible program behaviours afforded by structured programming was no smaller than the set available in flowchart programs. The essential benefit lay in transparency—the greatly improved clarity and human understanding that it motivated and supported. Any flowchart program can be recast in a structured form [3] merely by encoding the program text pointer in an explicit variable, and a somewhat better recasting could be obtained by applying a standard procedure for converting a finite-state machine to a regular expression. But as Dijkstra pointed out in his letter:

> *The exercise to translate an arbitrary flow diagram more or less mechanically into a jump-less one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.*

Another important advantage of structured programming over flowcharting is that it allows a systematic and constructive approach to program design. A precondition and postcondition give the program specification: that is, its required functional property. For each component, starting with the whole program, the developer chooses a structured construct configured so that satisfaction of its parts' specifications guarantees satisfaction of the specification of the component. As the design proceeds, the required functional property of the program is refined into required properties of each component, explicitly stated in preconditions, postconditions and loop invariants. As Dijkstra put it: the program and its proof of correctness were to be developed hand in hand [11; 14].

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.

4 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 their environments 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. The designer must fit the car’s properties and behaviour to this environment very closely. The environment, or problem world, comes to seem less significant only because for such products the purposes and the environment are so fully standardised and so well understood that knowledge of them becomes tacit and implicit. That knowledge is embodied in well-established product categories—a family hatchback saloon, a 4x4 sports utility vehicle, a luxury limousine, and so on—and in the parameters of the corresponding normal design [34] practices. The purchaser or user of a car in a particular category doesn’t need to ask whether it is suitable for the intended purpose and environment unless something consciously unusual is intended: transporting contestants to a sumo wrestling competition, perhaps, or making an overland trip into the Sahara. The car designer does not need to reconsider these requirement and environment factors afresh for each new car design: they are built into the normal design standards.

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. When William J LeMessurier was led to re-evaluate his design for the Citicorp Center and found it inadequate, a major criterion was based on the New York City weather records over the previous century: he discovered that a storm severe enough to destroy his building had a high probability of occurring as often as once in every sixteen years. Although the building had already been finished and was already occupied, he confessed his design error and immediately arranged for the necessary—and very expensive—strengthening modifications to be put in hand [23].

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.

4.1The formal machine and the non-formal world

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

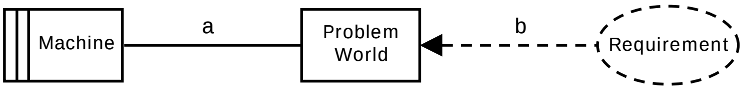


Fig. 2. A problem diagram: Machine, Problem World, and Requirement

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 task was succinctly described by Scherlis [28]:

> *... one of the greatest difficulties in software development is formalization—capturing in symbolic representation a worldly computational problem so that the statements obtained by following rules of symbolic manipulation are useful statements once translated back into the language of the world. The formalization problem is the essence of requirements engineering ...*

We can read Scherlis's statement as a description of the development work: first formalise, then calculate formally, then reinterpret the calculated results as statements about the world. But we can also read it as an account of the relationship between development and execution: the developers formalise and calculate, then the machine, in operation, executes the specification we have calculated, blindly producing the effects of translating our calculated results back into the language of reality in the 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, in specifying the machine behaviour that is to satisfy the requirement: 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.

5 Simple structures describing the problem world

The intimate and determinative relationship between the formal machine and the informal world places a special stress on understanding and formalising the properties of the problem world. Structural abstraction is a chief intellectual tool in this understanding. It supports separation of concerns at two levels. At one level the choice of parts and relationships in the reality to be understood reflects a separation of just those aspects from other aspects of the problem world. At the next level, each part and each connection is clearly separated from the other parts and connections. In this section we briefly illustrate and discuss some other aspects of structural description of a given reality. Here the primary concern is to obtain an appropriate structural abstraction of something that already exists: it is description rather than design. Discussion of design—the composition of multiple structures into one—is deferred to later sections.

As in any descriptive activity, the goal is to make a description that captures the properties of interest for the purpose in hand. In the non-formal problem world this is not a trivial matter: Cantwell Smith [5] pointed it out as a potential source of failure: the technical subject of model theory studies the relationship between formal descriptions and formal semantic domains, but there is no good theory of the relationship between formal semantic domains and properties of the problem worlds we must describe.

An example of simple structural description is the formalisation of the layout of an urban Metro railway, of which a fragment is shown in Fig. 3:

[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.

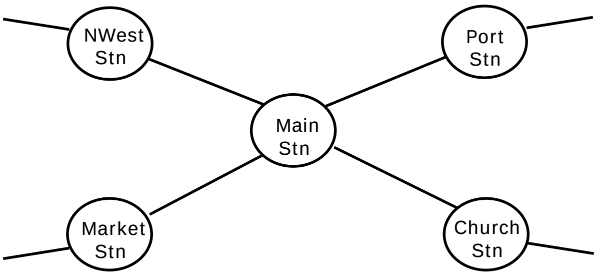


Fig. 3. A fragment of a local Metro railway structure

One purpose of making such a structure is to allow us to compose it later with another structure. For example, we may want to relate the railway connections to the road connections by bus in a part of the system that plans journeys. Another purpose is to allow us to reason about the mathematical properties of the graph and thus to deduce properties of the railway. For example, if the graph is connected we can infer that every station is reachable from every other station. If it is acyclic we can infer that between any two stations there is exactly one non-looping path, and hence that if one of the connecting tracks is destroyed or put out of action by a major accident the stations will thereby be partitioned into two disconnected subsets.

Whatever our purpose, we must be clear about the relationship between this formal representation and the reality of the subject matter. The graph formalisation is in terms of nodes, node names, edges, and an incidence relation. Each node represents a station of the Metro. But what exactly does this mean? Is every station represented by a node? If a disused station lies on a track represented by an arc—for example, the old Exchange station is on the line between Main and Market—does it appear in the graph? If two stations are connected by an underground pedestrian passageway are they represented by two nodes or by one? Are all the connecting tracks represented by arcs of the graph? Is the Metro connected by tracks to any other railway system? The arcs are undirected. Does that mean that all connecting tracks allow trains in both directions? Presumably we are not to assume that the track layouts at the stations allow all possible through trains—for example, that each of the 6 possible paths through Main Station can be traversed by a train in either direction. But what—if anything—is being said about this aspect of the structure?

Clarity about this relationship between the abstraction and the reality is essential if the structure is to serve any useful purpose. And we must be clear, too, about the purpose we want it to serve. What useful statements about the problem world do we hope to obtain by our reasoning? Are we concerned to plan train operating schedules? To analyse possible passenger routes? To plan track maintenance schedules? Different purposes will demand different formalisations of the problem world. As John von Neumann observed in The Theory of Games [35]: “There is no point in using exact methods where there is no clarity in the concepts and issues to which they are to be applied”. In Scherlis’s terms, a formal abstraction with an unclear purpose and an unclear relationship to its subject matter may allow symbolic manipulation: but the results of that manipulation cannot yield useful statements about the world.

Another example of a simple structure, this time of a software domain, is the object class structure shown in Fig. 4: it represents an aliasing scheme to be used by an email client. Each potential recipient of an outgoing email message is represented by an address, an alias can refer, acyclically, to one or more addresses or aliases, aliases and addresses are generalised to ids. Each email message has a target, which consists of a non-empty set of include ids and a set of exclude ids: this feature is considered convenient because it allows the sender to target, for example, a set of work colleagues while excluding any who are personal friends.

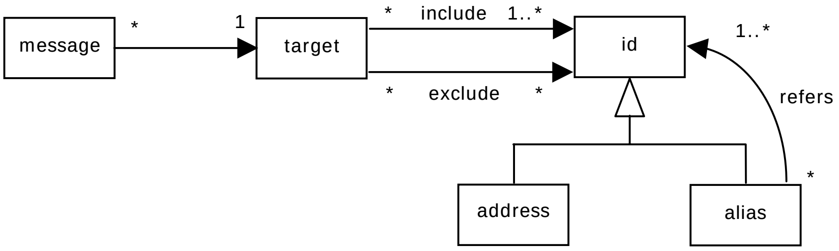


Fig. 4. A structure for e-mail recipients

For the conscientious specifier of requirements, and certainly for the designer of the email client software, the question will arise: How should the target of a message be resolved? Does it matter whether the difference set *include – exclude* is formed before or after resolution of alias references? The question is about a putative property of the structure, and it certainly does matter. The developer must find a way of answering this question reliably.

One way of answering the question is by formal or informal reasoning. Another is model-checking. For example, the significant parts of the structure, and the assertion of the putative property, can be formalised in the relational language of the Alloy model-checker [17] as shown in Fig. 5.

```
module aliases
sig id { }
sig address extends id { }
sig alias extends id{refers: set id}
fact {no a: alias | a in a.^refers}          // aliasing must not be cyclic
sig target {include, exclude: set id}
fun diffThenRefers (t: target): set id {t.(include - exclude).*refers - alias}
fun refersThenDiff (t: target): set id {(t.include.*refers - t.exclude.*refers) - alias}
assert OrderIrrelevant {
  all t: target | diffThenRefers(t) = refersThenDiff(t)
}
```

Fig. 5. Model-checking a putative object structure property

In the Alloy language an object class is represented by a signature, and its associations by fields of the signature, + and – denote set union and difference, and ^ and * denote transitive and reflexive transitive closure of a relation. The assertion OrderIrrelevant asserts the equality of the result regardless of the order of evaluation: the model-checker will find, if it can, a counter-example to this assertion.

Running the checker produces the trivial counterexample of Fig. 6. alias0 refers to address0, and target0 includes alias0 and excludes address0. If the difference set *include – exclude* is computed first, the target is address0, but if aliases are resolved first, then the target is empty.

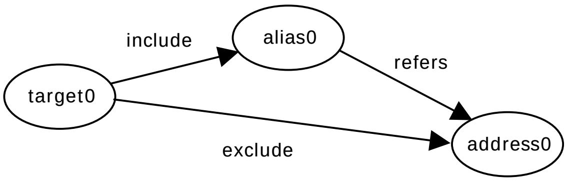


Fig. 6. A counterexample found by model-checking a structure

As this tiny example shows, faults in data structures are not limited to misrepresenting the reality the structure is intended to capture. Data structures, like program structures, have properties; by stating the expected or desired properties explicitly, and checking them carefully, the designer can eliminate or reduce a significant source of failure.

In the Metro example completeness is clearly likely to be a vital property. For almost all purposes the structural description must show all the stations and track connections, not just some of them. For a behavioural structure completeness is always a vital property. A dramatic failure[2] to accommodate some possible behaviours in an entirely different kind of system was reported [22] from the 2002 Afghanistan war. A US soldier used a Precision Lightweight GPS Receiver (a 'plugger') to set coordinates for an air strike. Seeing that the battery low warning light was on, he changed the battery, before pressing the Fire button. The device was designed, on starting or resuming operation after a battery change, to initialize the coordinate variables to its own location. The resulting strike killed the user and three comrades.

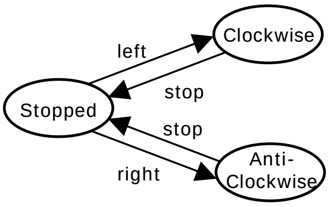


Fig. 7. A behavioural structure of a vending machine carousel

[2] I am grateful to Steve Ferg for bringing this incident to my attention.

Figure 7 shows an incomplete structural description of the behaviour of a carousel mechanism of a vending machine.

This incompleteness is a serious flaw in a descriptive structure: it invites misunderstandings that may give rise to development faults and consequent failures. One putative justification for incompleteness can be firmly dismissed at the outset—a claim that the control program for the carousel, described elsewhere, never emits the signals that would correspond to missing transitions. Such a claim would be irrelevant to the vending machine behaviour because it relies on an assertion that is not about the domain of the description in hand: we are describing the behaviour of the vending machine's carousel mechanism, not its controller.

In such a description, an input signal may be missing in a state for any one of three reasons. The carousel mechanism itself may be capable of inhibiting the signal (this is unlikely to be true in the example we are considering now). The omitted signal may be accepted by the mechanism but cause no state change. Or the response of the carousel mechanism to the signal may be unspecified: it causes the mechanism to enter an unknown state in which nothing can be asserted about its subsequent behaviour. The user of the description must be told the reason for each omitted transition.

One technique is to add a global annotation to indicate that any omitted outgoing transition is implicitly a transition to the unknown state, or that any omitted outgoing transition is implicitly a self-transition to the source state. Alternatively, each of the three cases can be explicitly represented in the syntax of the structure. For an inhibited input an annotation can be added to the state symbol. For an accepted signal that causes no state change a self-arc can be added to the state. An unspecified transition can be represented by an explicit transition to an additional Unknown state. Fig. 8 shows one possible completion of the carousel behavioural structure.

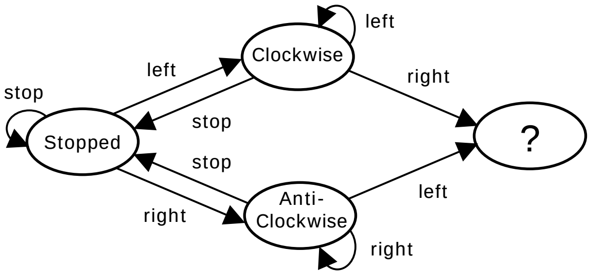


Fig. 8. A complete behavioural structure of a vending machine carousel

In the Stopped state, stop signals are accepted and ignored. In the Clockwise state, left signals are accepted and ignored and right signals cause a transition to the Unknown state. Similarly in the Anti-Clockwise state, right signals are accepted and ignored and left signals cause a transition to the Unknown state. The Unknown state has no outgoing transitions because exit from the Unknown state is, by definition, impossible. Figure 8, of course, is merely an example. Whether it is the correct explicit completion is a factual question about the particular carousel mechanism being described.

The value of providing context in a structure is not limited to program texts. Helping the reader to understand complexity is often a decisive factor in choosing a structural representation, and a decisive factor in understanding complexity is often the context in which each structural part must be understood. Figure 9 shows the structure of a monthly batch billing file in a system used by a telephone company.

For convenience in producing the bills, which are to be printed and bulk-posted, the file is sorted by postal region and, within that, by customer number. For each customer in the file there is an account record giving details of the customer's name and address, a plan record detailing the payment plan applying to the month, and, where applicable, a record of the balance brought forward. These are followed by records of the calls and messages of the month in chronological order.

The structure is shown as a regular expression in graphical form, with all subexpressions labelled. Without the subexpression labels it is equivalent to:

```
(Addr,Plan,(Bal|nul),Call*,(Addr,Plan,(Bal|nul),Call*)*)*
```

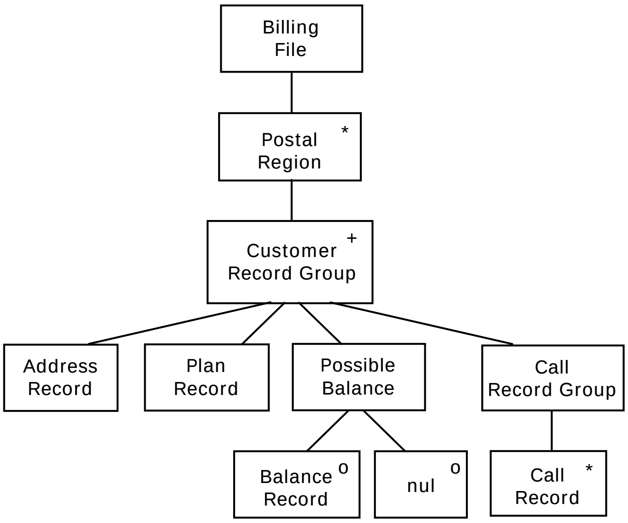


Fig. 9. The regular structure of a monthly billing file

The record sequence described by this structure could instead be described by a finite automaton, but for many purposes the form shown is clearer and more useful. In particular, it offers one of the advantages of structured programs over flowcharts: every part is set in a clear context. By attaching region and customer identifiers as attributes to the nodes we can easily associate each record and record group explicitly with the postal region and the customer to which it belongs—analogous to the coordinate system of a structured program. As a final observation we may add that the graphical representation has important advantages over an equivalent text—for example, over an attributed grammar written in the usual form of a sequence of productions—because it represents the structural context transparently by the graphical layout on the page. Even those programmers who are most devoted to formalism write their program texts in a nested, indented, layout.

One issue to consider in describing the structure of any subject is the description span. How much of the subject matter must be described—in terms of time, or space, or any other relevant coordinate system—to capture the relationships of importance for the purpose in hand adequately, and as clearly and simply as possible?

The issue is particularly important in two respects in behavioural structures. First, when the description shows an initial state it is necessary to say explicitly what segment of the subject's life history is being described and is known to begin in that state. In the description of the carousel mechanism no initial state is shown, so the question does not arise: the structure shown in Fig. 8 describes any segment of the carousel mechanism's life, starting in any of the four states. If an initial state were specified it would demand explanation. Does it represent the mechanism's state on leaving the factory? The state entered each time that power is switched on? The state following receipt of a reset signal not represented in the structure? The choice of span, of course, must be appropriate to the purpose for which the description is being made.

Second, span is important in behavioural structures whenever it is necessary to consider arbitrarily-defined segments of a longer history. This necessity was commonplace in batch data-processing systems, which are now for the most part old-fashioned or even obsolete. But batch processing has many current manifestations, especially where batching is used to improve efficiency—for example, in managing access to a widely-used resource of variable latency, such as a disk drive. In such cases it may be essential to avoid the mistake of trying to describe the behaviour over exactly the span of interest. Instead it may be much clearer to describe the longer span and to specify that the span of interest is an arbitrary segment of this. Many faults were introduced into batch data-processing systems by the apparent need to describe explicitly the possible behaviours of an employee, or a customer or supplier, or an order, over the one-week span that separated one batch run from the next. A description of the whole life history of the entity, accompanied by a statement that the behaviour over one week is an arbitrary week-long segment of this life history, would have been much simpler both to give and to understand[3].

6 Problem decomposition into subproblems

The non-formal richness of the problem world partly reflects a richness in the system requirements, stemming from many sources. It is therefore necessary to decompose the system requirements in some way. This decomposition is not to be achieved by decomposing the software. It is necessary to decompose the whole problem, with its problem world and requirement, into subproblems, each with its own problem world and requirement. The problem world for each subproblem is some subset of the whole problem world: some parts will be completely omitted, and for some or all of the others only a projection will be included.

[3] Any reader who is unconvinced of this point should try the analogous task of describing the structure whose elements are any 50 consecutive records of the billing file of Fig. 9.

This decomposition into subproblems is not in itself a structuring: it identifies the parts of a structure, but specifically does not concern itself with their relationships. Each subproblem is considered in isolation, and the recombination of the subproblem solutions into a solution to the complete problem is deferred until later. We discuss the recombination task in later sections. The justification for this approach, which may at first sight appear perverse, is an ambition to see each subproblem in its simplest possible form: only in this way can subproblems of familiar classes be easily recognised, and the concerns arising in each subproblem be easily identified and addressed.

One goal of problem decomposition is to ensure that each subproblem has a relatively simple purpose of achieving and maintaining some relationship between different parts of its problem world. As an illustration, consider a small traffic control system operating a cluster of lights in accordance with a stored regime that can be changed by the system's operator. A good decomposition of this problem is into two subproblems: one in which the operator edits the stored regime, and one in which the traffic is controlled by setting the lights as stipulated by the regime. The first, editing, subproblem is concerned to relate the operator's edit commands and the regime's changing state. The second subproblem is concerned to relate the states of the lights and the regime. The task of recombining the two subproblems is deferred.

Designing software to create and maintain relationships between different parts of the problem world can be seen as a task of structural composition. Each part of the world, considered from the point of view of the subproblem, has its own structure, and the solution to the subproblem depends on a composition—whether static or dynamic—of those structures. For example, certain simple kinds of program—especially, but not only, terminating programs—can be viewed as transformers of sequential inputs to sequential outputs. A one-pass compiler for a simple language can be of this type. The program in execution must traverse its inputs while using their values, in context, to produce its outputs. If the problem is a simple one, the program structure can be designed as a static composition of the structures of its inputs and outputs. Figure 10 shows a trivial illustration of the idea.

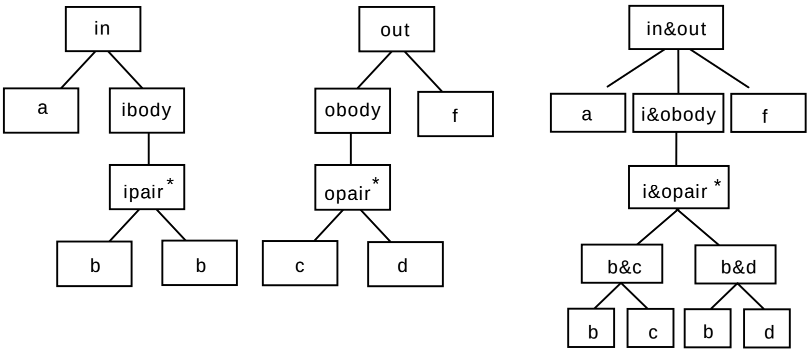


Fig. 10. Two constituent structures and a static composition structure

The regular structures *in* and *out* describe the input and output streams of a program to be designed. The program is required to derive one *opair* from each *ipair*, calculating the value of *c* from the first *b* and the value of *d* from the second *b; f* is a total computed from *a* and all the *ipair*s. The program structure *in&out* has been formed by merging the two constituent structures. Each node of the program structure corresponds to an *in* node whose value it uses, an *out* node whose value it produces, or both, as indicated by its name.

In this example, the required relationships are that the program should produce the output stream incrementally in a single pass over the input, and that the output values should be those specified. The successful accommodation of both constituent structures without distortion can be seen directly. Each constituent structure can be recovered by pruning the composition structure.[4] Using this kind of merging composition structure eliminates a significant source of error: the retention of each constituent structure ensures that the context of each part is preserved intact in the composition.

In this trivial example the composition structure is formed from given constituents whose individual properties are independent of the composition: in forming the composition it is therefore necessary to accommodate every possible instance of each constituent considered independently. A more complex task is the design or description of an interactive behavioural composition structure—that is, of a structure generated by interaction between two or more participating constituents. In such a composition the behaviour of each participant can be affected by the behaviour of the others: the possible choices for each participant are governed not only by that participant's state but also by the requests and demands of the other participants.

Consider, for example, the design of an ACB (Automatic Call-Back) feature in a telephone system. The purpose of an ACB feature is to assist a subscriber *s* who dials the number of another party *u* and finds that it is busy. The system offers the feature to the subscriber. If the offer is accepted and confirmed, the subscriber hangs up ('goes onhook' in the industry jargon). When the called number *u* is free, the system calls back the calling subscriber *s*, and when the subscriber answers it rings the requested number *u*. The system acts rather like a traditional secretary of subscriber *s*, but with one major difference. The secretary would first make the connection to *u* and avoid troubling *s* before *u* is on the line. The ACB feature is more mannerly: since it is *s* who wishes to make the call, *s* must wait for *u*.

The designed structure shown in Fig. 11 was intended to capture the required behaviour of the ACB feature itself. Clearly, this must compose the relevant behaviour and states of the subscriber *s*, of the connection to subscriber *u*, and of the telephone system that is responsible for basic telephone service into which the ACB feature is eventually to be integrated. The basic system is assumed, for purposes of the designed structure, to have no pre-existing features: it only provides connections in response to dialled requests.

[4] It is also necessary, after each pruning, to remove the interposed nodes *b&c* and *b&d*, which, having only one part each after pruning, are not significant in the pruned structures.

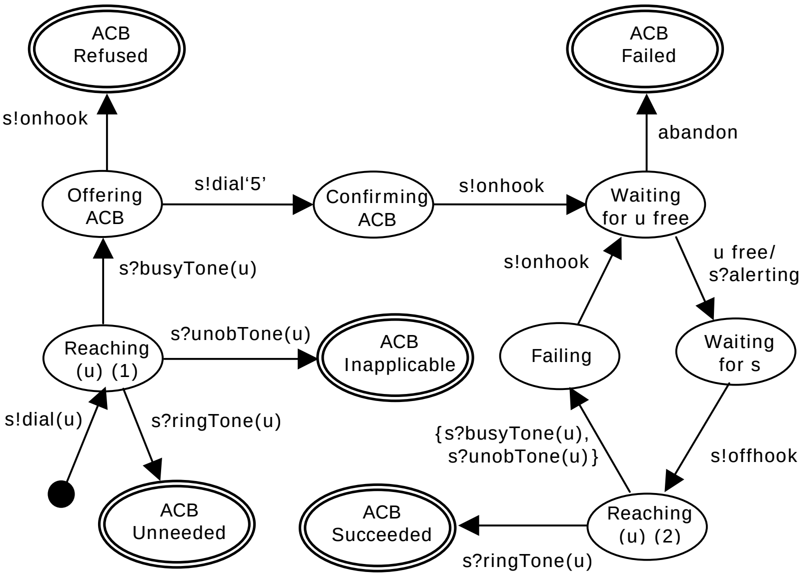


Fig. 11. Proposed designed behaviour of an Automatic Call-Back feature

The interaction begins when *s* dials *u*'s number. The basic system then attempts to reach *u*, with three possible outcomes. If *u*'s number is free, then *s* hears ringTone, indicating that *u*'s phone is alerting, and the ACB feature is not needed. If *u*'s number is unobtainable, then *s* hears unobTone, and the ACB feature is not applicable. If *u*'s number is busy, then *s* hears busyTone, and the ACB feature is applicable. In the last case, the feature is offered by a recorded voice message, inviting *s* to accept the offer by pressing '5' on the dialpad; refusal is indicated by *s* hanging up. If the offer is accepted, a confirming voice message is played and *s* hangs up.

Provision of the ACB service begins by waiting for *u*'s number to become free. When this happens *s*'s phone starts to ring, and when *s* answers it the system attempts to reach *u*. If *u*'s number is free, then *s* hears ringTone, indicating that *u*'s phone is alerting, the ACB feature has succeeded and the service is now completed. If *u*'s number is unobtainable or busy, then *s* hears unobTone or busyTone, and the current attempt to provide the service has failed: when *s* hangs up, further attempts are made until an attempt succeeds or the service is abandoned because too many attempts have been made or too much time has elapsed.

A crucial question about such a structure is: What are its intended designed properties? One essential property is that the structure, like all composition structures, should accommodate all possible behaviours of the constituents—here the participants in the interaction. Each participant has a full repertoire of potentially available actions. The actions actually available at any moment are limited by the participant's local state, and in some cases also by external enabling or disabling of actions. Accommodating all possible behaviours means that at each point in the interaction the composition accommodates every one of the currently possible actions.

The full repertoire for subscriber *s*, for example, is to go onhook, go offhook, or press any button on the dialpad[5], and subscriber *u* can do likewise. Each subscriber's local state limits the available actions according to whether the subscriber's phone is onhook or offhook: when it is offhook, all actions are possible except offhook, when it is onhook, no action is possible[6] except offhook. The basic telephone system, into which the ACB feature is to be integrated, has its own behaviour with its own repertoire of states and actions. For example, when *s* dials *u* initially, the system attempts to connect them, producing one of the three outcomes according to whether *u* is free, busy, or unobtainable.

It is immediately clear that the designer's obligation to accommodate all possible behaviours has not been fulfilled. The omissions include: *s* dials any number in any state, *s* hangs up in the Reaching(u)(2) state, *s* does not hang up in the Confirming-ACB state, *s* does not answer in the Waiting for *s* state, *s* neither hangs up nor accepts in the ConfirmingACB state, *u* dials *s* in the Waiting state. The structure of Fig.11 has many faults.

7 Structure within subproblems

When direct merging of contexts can not be achieved without distortion, a structural difficulty is present for which some resolution must be found. Consider, for example, the problem, well known to accountants, of combining a cycle based on seven-day weeks with one based on the months of the solar calendar. The week context and the month context are incompatible: that is, they can not both be fitted into the same regular structure. One unsatisfactory approach to the difficulty is to create a single composition structure that correctly accommodates only one of the two structures and includes the incompatible parts of the other in a distorted form, as shown in Fig.12.

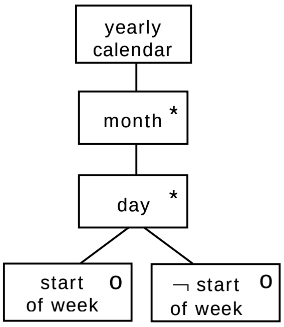


Fig. 12. A composition structure distorting one of its constituent structures

[5] We are assuming a simple old-fashioned handset here, and a simple old-fashioned land-line-based telephone service.

[6] In a certain sense, pressing a dialpad button is always possible, but has no effect when the phone is onhook: no signal is transmitted to the exchange computer, and no change is made to the local state of the phone.

The effect of the distortion is a partial loss of context. The composition structure provides no context corresponding to one week: the discrimination introduced here, between a day (a Monday) that starts a week and a day that does not, is a very poor substitute for this context. Any program computations that depend on the week context—perhaps calculating weekly average or maximum or minimum values of some variable associated with each day—must be fitted somehow into the structure shown. A program based on this structure will be complicated by the multiplicative effect of the month and week contexts: it may become necessary to distinguish the cases in which the month begins on a Monday, those in which it ends on a Sunday, those in which a month contains exactly four weeks, those in which it straddles five, and those in which it straddles six weeks, and so on. In this very small example the complexity may seem manageable, but in practice such complexities in realistic examples are a potent source of error and hence of system failure.

This kind of difficulty can often be handled much more effectively by exploiting one aspect of the malleability of software—the reification of program context in data. Dynamic context, associated in a structured program with a program text location and an appropriate set of execution indices [10], can be captured in one structure, reified in data values, and transported to another structure. In the case of the calendar problem the weekly and monthly contexts can be kept separate. In one program structure, based on weeks, the computations that depend on the week context are performed and the results appended to each day record in an output stream. This output stream furnishes the input to another program structure based on months, which then has no need to provide or reconstruct the week context: it treats the computed weekly values in each day record as attributes of the day. The general technique used here is a particular form of separation or decoupling [8]. It is more effective than forming and using a single inadequate composition structure, because it separates conflicting contexts and so reduces the likelihood of faults.

In such separation, a further level of problem decomposition has been introduced below that of the originally identified subproblems. This decomposition is based not on the identification of subproblems of familiar classes, but rather on a general principle—the separation of conflicting contexts—applied in a particular form to a particular subproblem class [16]: essentially, each of the conflicting contexts must be represented in a separate part of the subproblem.

For some subproblem classes a certain decomposition is completely standard. Consider, for example, a subproblem in which a rolling analysis is to be displayed of the current and historical prices of trades in a stock exchange. The dynamic structure of the trading activity to be analysed is different from the structure of the analysis to be computed and displayed, and the two structures are—almost certainly—in conflict. The conflicting structures are decoupled by introducing an intermediate data structure and decomposing the subproblem into two parts: one, based on the structure of the trading activity, to build and maintain the data structure, and the other, based on the structure of the analysis, to produce and update the display output. The intermediate data structure may take the form of a database on disk or an assemblage of objects in shorter-term memory.

8 Composition concerns

Problem decomposition produces a collection of subproblems that do not yet form a structure because their relationships have been intentionally—albeit temporarily—ignored. To recombine the subproblems is to form them into a complete problem structure by identifying the points at which they must enter into relationships, and determining what those relationships should be. This activity is in some respects similar to the activity of treating a single subproblem. It is concerned with composing existing structures according to some requirement, the requirement often emerging only when the possibilities of composition are considered. In some cases this composition will itself demand to be treated as a subproblem in its own right.

In general, two subproblems must be brought into relationship whenever their problem worlds are not disjoint. The composition of subproblems is not restricted to—and often not even concerned with—the composition of their machines: it is concerned rather with the composition of their effects in the problem world they share. One simple case arises in the traffic control system of the previous section. The regime appears in both subproblems: it is written in the editing subproblem and read in the other subproblem. This raises a classic interference concern: some granularity must be chosen at which mutual exclusion can be enforced. But this is not the only concern. There may be additional requirements governing the changeover from the old regime to the newly edited regime: for example, it may be necessary to introduce an intermediate phase in which all the lights are red.

Another example of a composition concern arises from direct conflict between subproblem requirements. An information system in which the more sensitive information is protected by password control can be decomposed into a pure information subproblem and a password control subproblem. On a particular request for information by a particular user the two subproblems may be in conflict: the information system requirement stipulates that the requested information should be delivered, the password control requirement stipulates that it should not. In forming a composition where conflict is present it is necessary to determine the precedence between the subproblems. Here the password control problem must take precedence.

Subproblem composition concerns can be regarded as the genus to which the feature interaction problem [4] belongs. Features, as they appear in telephone systems, can be regarded as subproblems of the overall problem of providing a convenient and versatile telephone service. It is particularly valuable to consider telephone call-processing features in isolation, because most features can indeed be used in isolation, superimposed only on the underlying basic telephone service. Consider, for example, the (flawed) structure shown in Fig. 11, intended to describe the ACB feature. It composes the interactions of four participants: the two subscribers *s* and *u*, the underlying basic telephone system, and the software machine that implements the ACB feature.

The structuring of telephone systems in terms of a number of such call-processing features developed partly because it is an effective way to decompose the complete functionality of such a system, but more cogently because telephone systems of this kind evolved over time under competitive pressure. The customers of the companies developing such systems were local telephone service companies, who wanted to compete by offering their subscribers more features and better features than the available alternative service suppliers. The call-processing (and other) features were added to the competing systems in a fast-paced succession of versions of the electronic switches on which they were implemented. In this way the switch developers were constrained by commercial forces to structure at least the functional requirements of their switches in terms of these features. From the beginning, producing a new version by complete redevelopment and redesign of the millions of lines of switch software was entirely impractical. Development was inevitably incremental, and the features were the increments.

The nature of the telephone system, in which users can invoke any of a large set of features, and any user can call any other user, gave rise to a huge number of potential feature interactions. Suppose, in the ACB example, that while *s* is initially dialling *u*, *u* is similarly dialling *s*. Each subscriber will find the other busy, and will be offered ACB service. If both accept the offer, there will then be two instances of the ACB feature with interleaved complementary behaviours. Further, the original assumption that the basic system has no pre-existing features, but only provides connections in response to dialled requests, is unrealistic and cannot stand. The designed structure of the ACB feature behaviour must therefore coexist not only with other instances of itself but also with other features.

The feature interaction problem is difficult both for requirements and for software design and construction, and can affect the dependability of a system in both respects. For the software designer it can give rise to huge and continually increasing complexity. Unmastered, this complexity will have the usual consequences: the system behaviour will sometimes be disastrous—for example, the system may crash; sometimes it will merely fail, as software may, to perform its specified functions. For the system user it can give rise to behaviour that is arguably in accordance with the specified—or at least the implemented—functionality, but is nonetheless surprising. In the formulation and analysis of requirements the complexity often manifests itself as conflict between one required function and another.

Consider, for example, the OCS (Originating Call Screening) feature, which allows a subscriber to specify a list of numbers to which calls will not be allowed from the subscriber's phone. This feature is particularly useful for subscribers with teenage children. Initially, the requirement is: "If the number dialled is a listed number the call is not connected." Then a new SD (Speed Dialling) feature is introduced. The subscriber specifies a 'SpeedList', mapping 'speedcodes' of the form '#xx' to frequently dialled numbers. The subscriber's teenage child adds to the subscriber's SpeedList a speedcode that maps to a forbidden number, and the OCS ban is now bypassed by the SD feature. Alternatively, the teenager can rely on a friend whose phone has the CF (Call Forwarding) feature. The friend arranges to forward calls to the forbidden number, which the teenager can now reach by calling the friend's number. Or the teenager can simply rely on a friend who has the 3-Way Calling feature (3WC), and is willing to set up a conference call between the teenager and the forbidden number. In effect, the OCS feature is actually or potentially in conflict with the SD, CF and 3WC features: satisfying their requirements prevents—or appears to prevent— satisfaction of the OCS requirements.

9 Top-down and bottom-up architecture

In software development the word *architecture* can mean many things [25; 2; 29]. Here we mean the identification of software components, their arrangement into one or more larger structures of chosen types, and the choice of component types and connecting interfaces.

This is, in a sense, programming in the large, a term introduced by DeRemer and Kron [9]. As DeRemer and Kron recognised, software architecture can be approached top-down, or bottom-up, or in some combination of the two. In a top-down approach, the components are identified and specified just well enough for the developer to feel confident—or at least hopeful—that their eventual implementations will fit as intended into the large structure. In a bottom-up approach the components are investigated in detail and their analysis and design carried to a point not far short of full implementation before their detailed interfaces and relationships are chosen and the larger structure is determined. In practice some combination of top-down and bottom-up is inevitable, not least because development is always to some extent iterative [24].

The traditional approach to software development favours an approach that is primarily top-down. It has an important appeal both to the manager, who must allocate the development work among several developers or groups of developers, and to the chief designer, who would like to sketch out the broad structure of the system implementation at the earliest possible stage. But it suffers from an important disadvantage that is apparent from the very nature of any kind of composition structure. A top-down approach to designing a composition structure of N components involves simultaneous engagement in at least N+1 intellectual tasks: the N component design tasks, and the task of composing them. An error in the conception or design of a component is likely to entail reconsideration of its relationships with its neighbours in the structure, with further consequences for those components, for their neighbours in turn, and for the whole structure. Because a full redesign is economically infeasible, the development must proceed with known design defects that will give rise to complexities, faults and a reduction in dependability of the whole system. This point was made by the physicist Richard Feynman in his contribution to the Rogers Committee's report on the Challenger space shuttle disaster, where he castigated the top-down development of the space shuttle main engine [6; 12]:

> *In bottom-up design, the components of a system are designed, tested, and if necessary modified before the design of the entire system has been set in concrete. In the top-down mode (invented by the military), the whole system is designed at once, but without resolving the many questions and conflicts that are normally ironed out in a bottom-up design. The whole system is then built before there is time for testing of components. The deficient and incompatible components must then be located (often a difficult problem in itself), redesigned, and rebuilt—an expensive and uncertain procedure. ... Until the foolishness of top-down design has been dropped in a fit of common sense, the harrowing succession of flawed designs will continue to appear in high-tech, high-cost public projects.*

Scepticism about a top-down approach to architecture is well-founded, and is reinforced by recognition of the need for problem analysis and for decomposition into subproblems. Software architecture can be seen as the composition of subproblem machines, and it is hard to see how that composition can be reliably and finally determined before the machines to be composed have been identified and understood.

9.1 Uniform architectures

Nonetheless, for some systems, certain kinds of architectural decision can usefully be made at an early stage, when the subproblems have not yet been identified. These are decisions about the design and adoption of a uniform architecture, based on a clearly identified need to master an overwhelming complexity by casting every subproblem into a form in which the complexity of its machine's interactions with other subproblem machines can be tightly controlled.

One example of such a use of uniform architecture is the recovery-block scheme for fault tolerance [26]. This is based on a recursive uniform architecture in which each component has the same idealised structure. A component embodies one or more software variants, each intended to satisfy the component specification. The controller within the component successively invokes variants, in some order of decreasing desirability, until either one succeeds or no further variant is available. In the latter case the component has failed, causing the failure of the higher-level component by which it was itself invoked. By introducing this uniform scheme the potential complexities of error detection and recovery at many points of a large software structure can be brought under control.

Another example of using a uniform architecture to master a potentially overwhelming complexity is found in the DFC abstract architecture for telephone—or, more generally—telecommunications systems [18]. The complexity comes from two distinct sources. First, the system contains many call-processing features, all accessed through the same narrow interface of a telephone handset and all therefore demanding use or control of its relatively few phenomena. Second, and more important, manufacturers of telephone switches compete by bringing new features to market in product releases that follow one another in quick succession. To address the feature interaction problem, it is therefore essential to be able to add new features quickly and easily without breaking the features already provided.

Each feature in DFC is regarded as a filter program, in the broad spirit of a pipe-and-filter architecture, whose input and output streams carry both the signals and the media content necessary for communication. The structure is dynamic, feature 'boxes' being assembled incrementally into a structure by the system router in response to 'virtual calls' placed by the boxes themselves. When a box places a virtual call, the router connects it to another feature box or to a line interface box, the connection forming another 'pipe' in the structure. In a very simple case the result may be what is shown in Fig. 13.

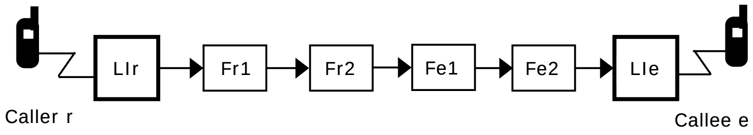


Fig. 13. A simple subproblem composition structure of telephone features

The caller's and callee's phones have persistent interfaces to the system, provided by line interface boxes *LIr* and *LIe*. The call shown has been assembled by the router by including the caller's features *Fr1* and *Fr2* (in response to successive virtual calls from *LIr* and *Fr1*), and the callee's features *Fe1*and *Fe2* (in response to successive virtual calls from *Fr2* and *Fe1*). During progress of the whole connection, any of the feature boxes in the assemblage may place further calls that will cause the structure to change and grow. For example, if *Fe2* is the CFB (Call Forward on Busy) feature, and the callee is busy when the original call is placed, *Fe2* will terminate its connection to *LIe* and place another call to the forward number. Boxes for the features of the forward number will then be inserted into the structure by the router.

A uniform scheme of this kind provides a high degree of isolation for the features. Conceptually, each feature box can be specified and designed as if it were the only box between the caller and callee line interfaces, which corresponds closely with the way the users of the system may think of the feature. To allow for the presence of other feature boxes it need only act transparently for signals and media that are not significant to its own feature behaviour, passing them on unchanged from its input to its output connection. In general, the behaviour of a feature box, like a filter program in a conventional pipe-and-filter architecture, does not depend on the behaviours of its neighbours, and because the specification of the connections is universal—the same for all virtual calls—the features can be assembled in different orders[7]. This possibility of assembling the features in different orders provides a dimension of control over their interactions: the DFC router inserts feature boxes into a usage according to a specified precedence ordering. If the CFB (Call Forwarding on Busy) feature has a higher precedence than the VMB (Voice Mail on Busy) feature, it will be placed closer to the callee's line interface box. It will therefore be able to respond to a busy signal, satisfying its requirement, and the signal will not reach the VMB box, which has lower precedence.

Use of a uniform composition architecture has a strong backwards influence on the structure and content of the system requirements. It becomes natural and desirable to structure the requirements—that is, the problem—in the form of a set of functionalities fitting naturally into the architectural scheme. Thus in DFC the notion of a feature becomes identified with what can be implemented in a DFC feature box[8].

[7] In fact, DFC feature boxes, like filters, can have more than two connections, and can be assembled not only into linear configurations but more generally into directed acyclic graphs.

[8] More exactly, a feature in DFC is identified with one or two feature box types. The EBI (Emergency Break In) feature, for example, allows the operator at an emergency service station to break into a normal subscriber call, requires one box associated with the subscriber and a box of a different type associated with the emergency service. Some features are associated with different box types for incoming and outgoing calls.

This influence can be seen as a beneficial rather than harmful kind of implementation bias: the form of a DFC box matches closely the form that the feature would take if it were used in isolation. The same effect can be seen in recovery blocks. The recovery block structure, with the controller, adjudicator (a component to determine whether the specification has been satisfied) and set of variants, provides a natural pattern for components in a high-dependability system. The controller's rule for selecting the next variant to be tried reflects, like the DFC router's behaviour, a precedence ordering between what may be regarded as distinct subproblems.

9.2 Component relationships

The primary concern in software architecture is the composition of subproblem machines in a way that satisfies required relationships among them. These relationships may emerge from the subproblem composition concerns, but they are also subject to other demands and influences.

One particular kind of relationship lies close to the heart of system dependability. Precedence between subproblems whose requirements are in conflict is addressed along with other subproblem composition concerns. But there is another kind of precedence, based on the criticality of the purpose served by the subproblem. The most critical functions must be the most dependable. It follows that correct execution of the machines providing those functions must not be allowed to depend on the behaviour or correctness of less critical subproblem machines. A dramatic error of this kind was made in the software architecture of a medical therapy machine. One requirement was that whenever the operator's safety button is pressed the treatment beam should be immediately turned off. Another requirement was command logging: the system was required to provide an audit trail of all commands issued to the equipment by the computer either autonomously or when requested by the operator. A partial data flow view of the architecture of the chosen design is shown, in a highly simplified form, in Fig.14.

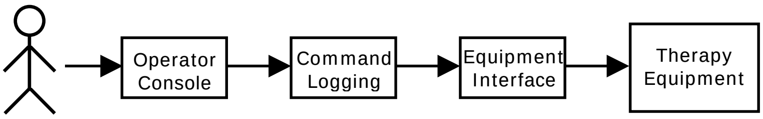


Fig. 14. A design guaranteeing logging of all commands

The design does guarantee that any command reaching the equipment has passed through the Command Logging component, but it also has the property of making all command execution dependent on the Command Logging module. Examination of the program code showed that the Command Logging module fails if the log disk is full, and does not then pass on the commands from the Operator's Console to the Equipment Interface. The console emergency button is therefore disabled when the log disk is full. It would be wrong to attribute this egregious fault to the faulty design or implementation of the Command Logging component. The design fault lies simply in making the emergency button function—arguably the most critical of all functions of the system—depend on any part of the structure from which it could instead have been made independent. An interesting discussion of this point, in the context of a lift control system, can be found in [30].

10 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.

References

[1] Baker FT (1972) System Quality Through Structured Programming, AFIPS Conference Proceedings 41:I, pp339-343.

[2] Bass L, Clements P, Kazman R (1998) Software Architecture in Practice, Addison-Wesley

[3] Böhm C, Jacopini G (1966), Flow diagrams, Turing machines and languages with only two formation rules. Communications of the ACM 9:5, pp366-371

[4] Calder M, Kolberg M, Magill EH, Reiff-Marganiec S (2003) Feature interaction: a critical review and considered forecast, Computer Networks 41:1, pp115-141

[5] Cantwell Smith B (1995) The Limits of Correctness, Prepared for the Symposium on Unintentional Nuclear War, Fifth Congress of the International Physicians for the Prevention of Nuclear War, Budapest, Hungary

[6] Report of the Presidential Commission on the Space Shuttle Challenger Accident, http://history.nasa.gov/rogersrep/51lcover.htm

[7] Constantine LL, Barnett TO eds (1968) Modular Programming: Proceedings of a National Symposium, Cambridge, MA, Information & Systems Press

[8] Conway ME (1963), Design of a separable transition-diagram compiler, Communications of the ACM 6:7, pp396-408

[9] De Remer F, Kron H (1976) Programming-in-the-large versus Programming-in-the-small, IEEE Transactions on Software Engineering 2:2, pp80-87

[10] Dijkstra EW (1968) A Case Against the Go To Statement, EWD 215, published as a letter to the Editor (Go To Statement Considered Harmful): Communications of the ACM 11:3, pp147-148

[11] Dijkstra EW (1976) A Discipline of Programming, Prentice-Hall

[12] Feynman RP (2001) What Do You Care What Other People Think? As told to Ralph Leighton, Norton, paperback edition

[13] Godden Structural Engineering Slide Library, http://nisee.berkeley.edu/godden/, National Information Service for Earthquake Engineering at the University of California, Berkeley

[14] Gries D (1981) The Science of Programming, Springer-Verlag

[15] Holloway CM (1999) From Bridges and Rockets, Lessons for Software Systems. In: Proceedings of the 17th International System Safety Conference, Orlando, Florida, pp598-607

[16] Jackson MA (1975) Principles of Program Design, Academic Press

[17] Jackson D, Shlyakhter I, Sridharan M (2001) A Micromodularity Mechanism. In: Proceedings of the ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference (FSE / ESEC '01)

[18] Jackson M, Zave P (1998) Distributed Feature Composition: A Virtual Architecture For Telecommunications Services, IEEE Transactions on Software Engineering 24: 10, Special Issue on Feature Interaction, pp831-847

[19] Jones CB (2003) *The Early Search for Tractable Ways of Reasoning about Programs*, IEEE Annals of the History of Computing 25:2, pp26-49

[20] Journal of Structural Engineering November 2002, 128:11, pp1367-1490

[21] Levy M, Salvadori M (1994) Why Buildings Fall Down: How Structures Fail, W W Norton and Co

[22] Loeb V (2002) ‘Friendly Fire’ Deaths Traced To Dead Battery: Taliban Targeted, but US Forces Killed, Washington Post 24 March 2002, p21

[23] Morgenstern J (1995) The Fifty-Nine-Story Crisis, The New Yorker, May 29 1995, pp45-53

[24] Nuseibeh B (2001) Weaving Together Requirements and Architectures, IEEE Computer 34:3, pp115-117

[25] Perry DE, Wolf AL (1992) Foundations for the Study of Software Architecture, ACM SE Notes October 1992, pp40-52

[26] Randell B (1975) System Structure for software fault tolerance, IEEE Transactions on Software Engineering 1:2, pp220-232

[27] Risks Digest (1990) http://catless.ncl.ac.uk/Risks/9.61.html

[28] Scherlis WL (1989) responding to E W Dijkstra “On the Cruelty of Really Teaching Computing Science”, Communications of the ACM 32:12, p407

[29] Shaw M, Garlan D (1996) Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall

[30] Shelton CP, Koopman P (2001) Developing a Software Architecture for Graceful Degradation in an Elevator Control System. In Proceedings of the Workshop on Reliability in Embedded Systems (in conjunction with SRDS), October 2001

[31] Sterling B (1992) The Hacker Crackdown: Law and Disorder on the Electronic Frontier, Bantam Books

[32] Turing AM (1949) Checking a large routine. In Report on a Conference on High Speed Automatic Calculating Machines, Cambridge University Mathematical Laboratory, pp67-69

[33] Turski WM (1986), And No Philosopher’s Stone Either. In: Information Processing 86, Proceedings of the IFIP 10th World Computer Congress, Dublin, Ireland, North-Holland, pp1077-1080

[34] Vincenti WG (1993) What Engineers Know and How They Know It: Analytical Studies from Aeronautical History, paperback edition, The Johns Hopkins University Press, Baltimore

[35] von Neumann J, Morgenstern O (1944) Theory of Games and Economic Behaviour, Princeton University Press

[36] Wilkes MV (1980) Programming Developments in Cambridge. In: A History of Computing in the Twentieth Century, N Metropolis, J Howlett and G-C Rota eds, Academic Press

Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective, Besnard D, Gacek C and Jones CB eds, pages 16-45, Springer, ISBN 1-84628-110-5, 2006