Testing the Machine In the World

Michael Jackson

The Open University & Newcastle University
jacksonma@acm.org

A central aim of software testing is assurance of functional correctness and dependability. For many software-intensive systems, including administrative, embedded, enterprise and communication systems, functional correctness means achieving the desired effects in the physical world, and dependability means dependability of those effects. For example, an administrative system for a lending library is required to ensure that only members can borrow books; that a member who has reserved a book and has been notified that the book is now available for collection in the library will not find that it has been lent to another member; that the catalogue gives reliable information about what is currently on the shelves, and so on. Similarly, a system to control a lift must ensure that the lift comes when summoned and takes the user to the desired floor; that the building manager can specify service priorities in terms of express lifts and time-dependent needs such as rush hours at the beginning and end of the working day; and that failure of the mechanical equipment does not endanger life.

The Machine, the World, and the Requirement

Functional correctness of such systems means that their *requirements* in the world are satisfied by a cooperation between the *machine*, which is the software executing on the computer, and the physical *problem world* itself. To demonstrate satisfaction it is necessary to formulate and reason about three distinct subjects. First, the requirements themselves, describing the desired effects in the world. Second, the specified behaviour of the computer at its interface with the problem world. Third, the given properties of the problem world on which the computer behaviour can rely to satisfy the requirement. These three are related by an entailment: if a computer with the specified behaviour is installed in a world with the given properties as described, then the requirement will be satisfied. For example: the computer behaviour is specified in terms of input and output ports at which it can switch the motor on and off and set its polarity, monitor the request buttons and the floor sensors, and so on; it is a given problem world property that a user wishing to summon the lift will press a request button, that if the motor is on and its polarity is set upwards then the lift rises in its shaft, and that when the lift is at the home position at a floor the corresponding sensor is on; and it is a requirement that when the lift is summoned it comes to the floor.

In a software-intensive system the problem world is inherently non-formal. We must reason about it to assure ourselves and our customers that the requirements will be satisfied, but this reasoning is always fragile. We reason on the basis of abstractions that are inevitably imperfect in the sense that we can never absolutely exclude all possibility of a counterexample to any formal assertion about the world:

no bound can be set to the considerations that may affect the truth of a formal assertion. Further, the problem world, unlike an abstract mathematical world, will typically exhibit autonomous, and easily neglected, state changes: library members may change their names, or become bankrupt, or emigrate or die, and the books may be lost or destroyed; users who have requested the lift may change their mind and walk away, or may place an obstruction between the lift doors while they go back and forth between the open lift and their office. A functionally correct and dependable software-intensive system must find a way to deal adequately with all these evident obstacles to reliable formal reasoning.

The World Can Not Be Avoided

It may, therefore, seem attractive to eliminate the messy non-formal world from our consideration as software engineers. Can we not treat our task simply as the development of software to satisfy a formal specification of the computer's behaviour at its interface with the problem world, leaving the messy non-formality of the world outside our *cordon sanitaire*? This was the view of Dijkstra, who regarded the specification as a 'logical firewall' between the non-formal concerns of the world outside and the formal task of program development. [1]

Unfortunately this neat division is not possible for a sfotware-intensive system. Specification of the computer behaviour to be evoked by the software will make little sense when divorced from a clear description of the given properties of the problem world and of the effects that the system must produce there. To understand the stipulation that when line 17 goes high the computer must set line 23 low we must talk in terms of the meaning of line 17—that a request button has been pressed on floor 3—and of line 23—that the motor polarity is being set to upwards—and of the association that the problem world and the system requirement impose between these otherwise unrelated phenomena. In other words, we must reason, as before, about the requirements, problem world properties, and computer specification.

Decomposition Into Subproblems

The complexity of the system and its problem world demands mastering by decomposition. An appropriate form of decomposition is to decompose the original *problem* into *subproblems*. Like the original problem, each subproblem has a requirement, a machine, and a problem world. For example, the lift control problem may be decomposed into three subproblems. The first, the service subproblem, provides lift service according to the priorities currently established by the building manager; the second, the display subproblem, maintains a display in the ground floor lobby showing the position of each lift and the outstanding service requests; the third, the safety subproblem, monitors the equipment and, on detecting a fault, applies the emergency brake that prevents an uncontrolled free fall of the lift car to the bottom of the shaft. Each subproblem is concerned only with certain parts of the original problem world. For example, the service subproblem is not concerned with the emergency brake or with the lobby display; and the safety subproblem is not

concerned with the display or with the request buttons. Further decomposition will be needed. For example, the lift service subproblem must be decomposed into an editing subproblem, in which the building manager edits a representation of the scheduling priorities, and a scheduling subproblem, in which the latest edit priorities are used to govern decisions about lift dispatch.

Normal Design and Subproblem Concerns

This kind of decomposition is based on two complementary principles. First, each subproblem captures a coherent and intelligible subfunction of the system, where a subfunction can often be loosely identified with a feature. Second, each subproblem matches a recognised problem pattern. For example, editing the priority rules is a problem of the same general kind as a simple text or graphics editor. There is an analogy here, between subproblems identified in this way and the familiar components of a physical system such as a motor car. This decomposition is typical of what Vincenti [2] calls *normal design*: "The engineer engaged in such design knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task." A vital characteristic of normal design is that the unbounded potential difficulties posed by the problem world are brought under a good degree of control by accumulated experience. The engineering community knows what concerns must be addressed to obtain a dependable product, and those concerns are addressed partly by the standard normal design itself, and partly by the practice of normal design in which the engineers pay explicit attention to the concerns that have proved important in the past. Departure from normal design norms is a recipe for failure—sometimes catastrophic, as in the famous collapse of the Tacoma Narrows Bridge [3].

An example of such concerns in a component of a software-intensive system is the *initialisation concern*. The need for initialisation of program variables is well known, and schemes have been developed to ensure that failure due to accessing an uninitialised variable can be reliably avoided. A similar concern applies to the relationship between a software component and its problem world. When the lift control program begins execution, for example, it may be necessary to ensure that the lift car is at the ground floor with the doors open. Or, alternatively, it may be possible for the software to detect the state of the problem world and to adjust its own initial behaviour accordingly. There are many variations on this theme. What matters is that the developers must be aware of the concern and know how to address it adequately.

Subproblems as System Components

If we regard subproblems as system components we must recognise that they do not interact solely within the machine. On the contrary, much of their interaction takes place through the medium of the problem world itself, as one component affects the state of a part of the problem world that it shares with another subproblem. These interactions in the problem world give rise to the need for explicit attention to

subproblem composition and to the composition concerns that accompany it. For example execution of two subproblems that share a part of the problem world must be controlled to obtain an appropriate interleaving: the newly edited representation of the lift scheduling priorities must at some point be adopted by the lift service subproblem. A very different example is the possibility of requirement conflict. The safety subproblem may determine that the mechanical equipment has developed a fault, that the emergency brake must be applied, and that the motor must be switched off and held off; the lift service subproblem may at the very same time determine that there is a service request for which the lift must be sent to a certain floor, and that the motor must therefore be switched on. The conflict must, of course, be resolved by a consideration of the relative precedence of the two requirements, and the chosen precedence must be implemented in the composed system. Another example of a composition concern is the need to ensure that failure of a non-critical function—possibly by erroneous design or programming of the software—cannot cause failure of a critical function. The lobby display subproblem, however badly, or even perversely, implemented, must not be able to obstruct correct functioning of the safety subproblem.

A Different View of Functional Correctness

These composition concerns compel us to consider a different, and more nuanced, notion of functional correctness. In place of the single simplistic entailment relating machine specification, given properties of the problem world, and requirement, we have a corresponding entailment for each subproblem, along with the need to compose and reconcile the different views of the problem world that have been adopted for each individual subproblem. For the lift service subproblem the mechanical equipment is fault-free; but for the safety subproblem it is potentially faulty. For the lobby display the lift movement is autonomous, but for the lift service it is the object of control. A comprehensive universal description of the problem world properties, accommodating the point of view of every subproblem, would be intractably complex and obscure. We are compelled to retain the view of the whole problem and its world as an assemblage of components, locating our view of each component in its place in the structure of interactions induced by our addressing of the manifold composition concerns.

The unbounded richness of the problem world precludes a fully comprehensive enumeration of all possible subproblem interactions. But it is useful to recognise two general interaction categories, somewhat in the spirit of the treatment of feature interactions in telecommunication systems. Some *positive interactions* must be reliably realised by the implemented system. For example, the lift service subproblem must use the newest edited version of the priority rules, and the lobby display subproblem must recognise that the outstanding requests for a floor have been services exactly when the lift service subproblem has indeed caused the lift to visit that floor and to permit the requesting users to enter or leave the lift car. These positive interactions are, in general, readily identifiable. But there is also a set of potential *negative interactions*, in which the interaction of two subproblems causes undesired and even catastrophic effects. For example, it is imaginable that the

changeover from an older to a newly edited version of the priority rules might be able to deadlock the lift service problem. Such negative interactions, of which there are potentially an unbounded number, must be identified and addressed by the application of normal design practices, in which are embodied the lessons learned by long experience.

Some Implications For Software Testing

Ultimately, functional testing of a software-intensive system must take place when the software has been installed in the problem world. Nothing less can bridge the gap that is opened up between the non-formal nature of the problem world and the formal—or quasi-formal—world of a well-engineered computer executing software written in a well-defined programming language under a reliable operating system. But of course such full integration testing is very expensive, and may sometimes be literally impossible. So, for this reason alone, it is certainly necessary to conduct smaller and cheaper unit tests, substituting simulation for the relevant parts of the real problem world.

Since different subproblems typically take different views of the parts of the problem world they have in common, the problem world simulation must embody different properties according to the subproblem under test. One simulation of the problem world cannot be enough.

Another reason for unit testing is the need to decompose the system function into subfunctions in order to reduce the number of test cases needed. The hope here is that the decomposed subfunctions, once tested, can be reassembled in a compositional fashion. That is, that if subfunction A and subfunction B are both known to be correct, then their composition into a combined function A+B must be correct. In its naïve form this hope is too optimistic. Even if the interactions between the subproblem software parts within the computer can be fully mastered, their interactions through the medium of the problem world parts they share are potentially more problematical. The sad story of the de Havilland Comet 1 aircraft showed the difficulty clearly. The aircraft body had been fully tested for behaviour under compression and decompression; and it had been fully tested for behaviour under flexing and torsional stress: both tests showed that the fuselage design fully met its objectives in each respect. But it had never been tested for both simultaneously, and in practice the combination of both kinds of stress was a major contributor to the aircraft's failure in flight.

Testing is, from one point of view, a searching process, in which faults potentially leading to failures are sought. The search should, ideally, be conducted in the light of the richest possible knowledge of how faults come about and hence where they are likely to be found. To take a very simple example from program coding, it is well-known that programmers are prone to write the assignment operator '=' in place of the equality condition '=='; so a code inspection should search specifically for such errors. In the same way, in a software-intensive system, such knowledge is the fruit of the experience embodied in normal design. To know that subproblems of a particular class raise an initialisation or identities concern, or that a particular subproblem

composition raises a switching or interleaving concern, is to know that testing should explicitly search for failure to address those specific concerns in an adequate fashion.

References

[1] E W Dijkstra; On the Cruelty of really Teaching Computer Science; CACM 32,12, December 1989, pp1398-1404.

[2] W G Vincenti; What Engineers Know and How They Know It; Johns Hopkins University Press, 1993.

[3] C Michael Holloway; From Bridges and Rockets; Lessons for Software Systems; Proceedings of the 17th International System Safety Conference, 1999.