19/3/98

Chapter 3.
Testing, testing, testing!

Summary. Testing, correctness, quality oriented testing, when testing is done, finite state machine testing, X-machine testing, design for test, the fundamental theorem, reflections and conclusions.

All software systems are subject to testing - for some of them testing is the major activity in the project. Testing, however, rarely gets the attention it deserves from researchers and developers, partly because its foundations are very weak and ill-understood. The principal purpose of testing is to detect (and then remove) faults in a software system. A number of techniques for carrying out testing, and in particular, for the generation of test sets exist. Many sophisticated (and expensive) tools are available on the market and many look to these to provide a solution to the problems of building fault-free systems. We consider the problem of fault detection and note that few, if any, of the existing methods really address the real issues. In particular no methods allow us to make any statement about the type or precise number of faults that remain undetected after testing is completed. Thus we cannot really measure the effectiveness of our testing activities in any rigorous way. However, by considering testing from a straightforward, theoretical point of view we demonstrate that a new method for generating test cases can provide a more convincing approach to the problem of detecting ALL faults and allows us to make sensible claims about the level and type of faults remaining after the testing process is complete.

3.1. What is testing?

The purposes of software testing have been discussed in a number of places and we will consider a few of the interesting statements and claims that have been made about testing.

Myers [8] states that the purpose of testing is to *detect* faults and the definition of a good test is one that *uncovers* faults. The implication clearly being that a test that fails to uncover any existing faults is inadequate in some sense. This emphasis on the finding of faults could lead one to assume that the more faults detected the better the test. However, one could argue that a test that uncovers a single serious fault might well be more successful than a test that uncovered a number of trivial faults. This immediately leads us into the vexed question of what is a “serious fault” and how these might be defined and identified!

Dijkstra [59] pours scorn on the whole concept of testing and states: “Program testing can be used to show the presence of bugs, but never to show their absence!” This is true of most of the testing methods currently in use and is clearly a major drawback. Dijkstra’s belief, shared by many in the formal methods community is that formal verification, that is mathematical proofs that the system meets all the conditions required of it, is the only solution. However, the massive complexity of today’s applications seems to be beyond the capability of these techniques, researchers have not demonstrated that formal verification is “scalable” and practical in the context of current needs. It is also clear from a number of

surveys that many of the notations, techniques and concepts of formal methods are regarded as very difficult to use. The quality of many tools to support the methods is often poor. Furthermore, some example systems which have supposedly been formally verified have been shown to have faults as a result of testing! (See [60]).

There is another issue, if we are really interested in building correct systems. The clients and users need to be able to confirm that the specification is correct, this is impossible if it is written in a complex mathematical notation that they do not understand. If the specification is wrong no amount of formal verification will deliver a correct system.

Types of testing strategy.
Testing techniques can be summarised as falling into 3 main categories: *functional testing, structural testing* and *random testing*. In a sense these are distinguished by the sort of information that is used to generate a test set. A test set is a set of input sequences, or test vectors, that can be applied to the system to establish its behaviour. Given such a test set it is necessary to have some idea of what the correct response of the system should be. Thus testing, in its simplest form, involves applying the test vectors to the implementation and deciding if the resulting behaviour is what is required. This process needs to be supported by a variety of test tools in a suitable environment, so that test generation tools have to be used alongside test application systems, test oracles and test evaluation tools.

Functional (or black box) testing, see for example, [61], should start with a functional specification or description of what the desired system should behave like. The test set is then constructed on this basis and the result of applying the test set is evaluated and compared with the desired result deduced from the specification. It is difficult to construct test sets from informal specifications, it has to be done by hand. Some attempts have been made to use formal specifications, for example written in Z [62], [63], VDM [64], OBJ [65] as a basis for automatically generating test sets but without a great deal of success. It seems reasonable to assume, however, that basing testing on functional testing using a precise formal specification is likely to be the most effective and trustworthy approach.

Structural (or white box) testing, see [66], is based on the design of the implementation. If one has to use a tool to generate a very large set of test vectors it necessitates the use of a formal description of the system that can be input into the test generation tool. In the absence of a formal specification the only thing available is the source code of the application under test. This can also be used for static analysis, a process which can provide a lot of information about the construction and behaviour of the implemented software, [67]. Structural testing techniques usually involve the analysis of the flow of activity in the program and can be based on finding suitable test vectors that will exercise different parts of the code in various ways, such as ensuring that every decision point is visited at least once, that every procedure is called at least once etc. The coverage of the test is then calculated on the basis of this type of analysis. The fundamental objection to this type of method is that it assumes that the code is sufficiently close to the requirements that it it can detect enough faults to provide some confidence that the system is correct. However, it is not possible to make this final deduction, there is no inductive link between the code and the specification, in fact there is no detailed specification in most cases, just informal language descriptions of desired behaviour.

Random testing proceeds on a different basis [68]. The input domain is specified as best as it can and the random test generation tools then generates test vectors in a random way. It is possible to finesse the method if a knowledge of the probability distribution of the likely inputs is available. The test evaluation process is then based on evaluating the program behaviour on the basis of an informal specification.

A number of experiments have been carried out to try to establish which of these methods is the most effective - in the complete absence of any theoretical foundation which might predict this. Typically the experiment will involve a small piece of code which is seeded with known faults and the success of the different methods in detecting these faults is measured. Further analysis could be done on the type of faults each method was good or poor at detecting. Faults are classified for this purpose in a number of categories. Results from this type of survey can be useful in establishing the relative strengths and weakness of different, specific methods of test set generation. However, the situation is essentially artificial and it is not clear what can be said in general. The method is unable to prove, for example, that either approach is better at detecting naturally occurring or unseeded faults (the seeded ones may not be typical of real faults), or to identify conditions under which a method detects all faults. Thévenod-Fosse and Waeselynck [69], for example, concluded that the most successful approach involved the use of a combined functional and statistical approach. Reid [70] demonstrated that boundary value testing (a type of functional testing) was the most successful. We will comment on some of our experiments later in this chapter.

3.2. Fundamental issues of correct system design.

There is a case for hoping that building systems from *dependable components* will overcome some of the problems of building correct systems and it may be possible that current testing or formal verification methods can be used *together* to provide the foundation for a scientifically based engineering approach to software development. At present, however, there is little sign of this happening. The software engineering industry is spending more and more time on testing in the hope of ensuring quality and yet real scientific evidence that they are being successful is minimal.

The subject of reuse and object oriented design methods are areas where great hopes have been expressed for a genuine component based approach to software construction. The libraries of classes that have become available, however, have often been a disappointment. It is hard to establish what the specification of the components in these libraries are, worst still the software extracted often behaves in an unpredictable way when integrated into a new system. These object-oriented methods are often held to be a great advance in the pursuit of quality software because of their supposed foundation on the principle of software or code reusability. However, it is pointed out in Binder [15] that, far from obviating the need for so much testing, the object-oriented methods require very substantial amounts of effort devoted to the task. Inheritance and dynamic binding provide many opportunities for introducing faults, the large numbers of interfaces between objects and the loose state control environment also compound the problem. Binder states “*the hoped for reduction in object-oriented testing due to reuse is illusory*”. In fact many new testing issues are raised by this paradigm. As a result the need for testing will undoubtedly grow and the industry will become more and more dependent on the successful pursuit of this activity.

One could argue that our current testing methods are not based on a firm scientific approach or theory, they tend to be methods that have evolved, informal *ad hoc* approaches that have been rationalised after the event rather than carefully constructed engineering methods based on a defensible well-founded strategy. We test because we need to but we do not demand from testing the sort of rigour that the product demands. Testing must be made more to do with reassuring *clients* and *users* than with reassuring *testers*, *designers* and their *managers*! We must focus on *test effectiveness* rather than test *effort.*

One of the claims made here and elsewhere, is that this will not be possible unless there is a more integrated approach to the design process; and until the testing methods used satisfy some fundamental requirements dictated by a careful investigation of the nature of software systems.

Although it might appear that we are belittling the achievements of software testing as a process this is not the intention. Clearly the application of existing testing methods has resulted in the identification and subsequent removal of many important faults in software systems, however the question remains about what faults are left undetected until the system is in service. It is to try to address this issue that we now turn.

One of the most successful approaches to scientific understanding is the *reductionist* philosophy. This entails the reduction of one problem to the solution of simpler ones or to the understanding of a lower level, perhaps more microscopic situation. Since the current popular design methods in software engineering tend to emphasise the construction of systems from modules, objects and other simpler components one might hope that a similar reductionist approach to testing might be possible. This is not addressed by the different techniques of unit test and system/integration test or the test management process, it is much more fundamental than that. In such a reductionist approach we would consider a system and produce a testing regime that resulted in the *complete* reduction of the test problem for the system to one of looking at the test problem for the components or reduced parts. However few testing methods support this view since we would have to be able to make the following statement:

> "the system $\boldsymbol{S}$ is composed of the parts $\boldsymbol{P_1, ..., P_n}$;
> as a result of carrying out a testing process on **S** we can deduce that **S** is fault-free *if* each of $\boldsymbol{P_1, ..., P_n}$ are fault free.

Here we define fault-free in the natural sense - that is the system completely satisfies the behavioural requirements as detailed in the specification. A prerequisite is that the specification should be available as some formal, mathematical description to ensure that we can actually establish what the requirements are in an unambiguous sense. The exercising of tests on the system $\boldsymbol{S}$ involves the testing of the complete system, its internal and external interfaces as well as the the components $\boldsymbol{P_i}$.

A corollary is that the approach could be applied to each component $\boldsymbol{P_i}$ thus enabling the reductionist method to be continued downwards for as far is appropriate and feasible. Ultimately we might end up at a point where the components in question are *tried and trusted*, having been extensively analysed and used over a number of years and thus to have survived through a process of "natural selection".

Fault detection.

For this approach to work we need to find some mechanism whereby we can establish that if all the components are free from fault then so is the complete system. This is something that is quite beyond most current testing methods. The claim “the system/component is fault-free” is quite beyond current testing methods. In practise all we can usually say is that we have uncovered a number of faults over a period of testing effort and the graph of the number of faults against the period or amount of testing, measured suitably, indicates that the *growth rate* is reducing, see Figure 3.1.

.

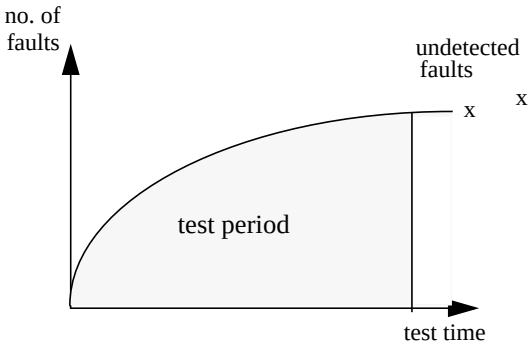

Figure 3.1 The relationship between fault detection and test time

The trouble is we do not know that **no** further faults are in the system at any particular time. Also, in general we cannot assert that the only faults remaining are located in a specific module or component. A general formula for this curve is not known, if one existed it would probably depend on the type of system, on the type of test methods and perhaps on the people doing and managing the testing as well as wider issues relating to the management of the design project, the attitudes of the clients, the implementation vehicle, the design methods and so on. High quality empirical results obtained over a very long period would be needed to make proper use of this approach even then it is less than ideal. The work of Littlewood and others [71], provides some statistical approaches to the question of software reliability but in some senses it is used as substitute because of the failure of software testing to deliver methods able to provide precise answers to the question “what faults are left?”.

Probable correctness.

Hamlet [72] introduced the idea of “probable correctness”. Essentially, he considered the question of what we can say once a system has “passed” all the tests applied to it. He argued from a simple probabilistic point of view about the way in which a value could be placed on the likelihood of faults remaining in the system. This is an attractive idea that deserves more research. Hamlet’s discussion was based on a number of very restrictive assumptions about the distribution of faults in the system and there is no discussion of the type and seriousness of faults. After all, if the likelihood of any faults remaining was quite small, say 1.0×10^{-5}, then we may or may not be satisfied but it would be so much more reassuring if we could say that the likelihood of a *serious* fault was 1.0×10^{-10}, however we define serious - perhaps we might identify certain safety considerations that must be met and orient our tests towards uncovering faults that cause these to be violated. We may not mind too much if a screen display omitted a full-stop in some text - on the other hand if the display was giving instructions on medical dosage to paramedics a full-stop (decimal point) missing could be disastrous. So the importance of a fault depends on the application and the working environment. Treating all faults as of equal import is an

unsatisfactory basis for this type of argument.

A simple theory of testing.

Goodenough and Gerhard [73] introduce an outline theory for testing. They treat a software system as a (partial) function from an input set to an output set and the testing process consists of constructing "revealing domains" that expose faults in this function - in other words they are looking for values for which the specified function and the implemented function differ. This is a very general and abstract approach which provides a useful set of simple concepts and terminology for the discussion of some aspects of testing without giving too many clues as how to construct effective testing strategies.

We claim, therefore, that few of the existing testing methods measure up to our demands and the main theoretical attempts are inadequate for the purpose that we explained above - the philosophy of reductionist testing.

When testing is done.

Eventually the test phase must end and the product released. When should this be and what will we be able to say about the quality of the system that we deliver once testing has ended? If our core objective is that the system we are releasing has as few faults as possible then we need to be able to have some estimate of the number and type of faults that may remain once testing has ended. Suppose that we have applied our test process to the product with the result that the system has passed all of the tests in the final test phase. We would like to claim that the system is now fault free but this is going to be very hard to justify. We might try to estimate this important quality of the product - that is, its freedom from faults - by measuring some attribute of the test process - typically its coverage. However the relationship between test coverage and the location of faults is complex and unclear, it is possible to have a high test coverage which fails to detect crucial faults, faults which could be detected with a test strategy with a much lower coverage. In the absence of a detailed and precise functional specification the tests may not be detecting crucial functional faults anyway, since we may not have a clear idea of what these are! So we are faced with a number of dilemmas, we need to plan our testing strategy so that we can produce software of sufficient quality in an economic way and we need to know when we have achieved this quality target. In other words we need to be able to decide whether our test strategy is adequate and when our testing can end - but how can we relate this to our quality aspirations?

Most decisions on the test strategy to be used in a project are based on the available information we have about the system under test and on the tools that are to hand. Under these conditions decisions on when to terminate testing will be made on the basis of: time running out (delivery pressures); money running out (economic pressures); personnel becoming unavailable (management pressures) and so on. None of these are a suitable basis for quality-oriented decisions. Quality-oriented testing is testing that relates to the issue of what faults remain after testing and where they may be located. We may try to rationalise the situation by collecting data on the detection of faults during the test process and use some rough reliability model for estimating the number of remaining faults but it is rare that these are accurate and we will not know anything about the severity or location of the faults remaining. This must compromise the quality of our product. The "saving grace" is that it is the same for everyone and so our competitive edge is not threatened, since other companies do not have access to test methods that can overcome these problems! Or do they?

3.3. An approach to quality-oriented testing.

Some practical considerations.

Let us try to identify the sort of approach that might break through these limitations. First we have to define what the system we are testing is supposed to do, in other words we need a precise and complete statement of its functional requirements. This is hard enough but without it we are already in trouble. If the information we are putting into the testing process is incomplete then the outcome cannot compensate for this weakness, in other words if we don't know what a system is supposed to do how can we establish whether it does it?

Formal specification languages, the academics' solution to this problem have promised much and delivered little, primarily due to their unusability and their inability to scale up to industrial problems. It is possible to develop formal descriptions from the popular diagrammatic techniques if they are annotated suitably, see, for example, Chantatub & Holcombe, [74], and purely visual versions of the formal language Z have been developed with powerful and user friendly tools [75].

The next requirement is a method for generating complete functional test sets from this specification. Most attempts at doing this, such as those based on Z, VDM, OBJ, still fail under the requirement of being able to say something about the faults that remain when testing stops since they are based on techniques such as category-partition and similar approaches, (when generating test sets we have to make random or at best statistics driven choices which cannot guarantee that all faults have been covered). The test generation process can be continued indefinitely without any guarantee that the fault detection capability of the test set is increasing in a commensurate manner. We may know what the system is supposed to do but we cannot say anything about when to stop testing and what faults may remain using these techniques.

The next issue is that of refinement. Most systems are constructed from components, and in the brave new world of Java these may be applications, applets, objects, classes etc. chosen from a variety of sources and integrated, possibly at run time, into a complete, dynamic software system. Some of these components may be highly reliable - tried and trusted - and to all intents and purposes can be regarded as fault free under suitable conditions. We would like to have a testing strategy that exploited this situation, which detected ALL possible faults in the integration (top level system) and which had a clear termination. Subsequent testing of the components is then carried out in an appropriate way - if a particular component is not tried and trusted then it can be broken down into simpler components and its integration fully tested, with the sub-components themselves being tested separately and so on. If a genuine market in components emerges and enough information about the reliability (correctness) of these components is available - together with a formal description of their functionality - then this will represent a major advance in the quest for software quality at an economic cost. The next Chapter considers these issues, further.

Finally, we need to develop an approach which is not too intrusive in terms of the in-house methods and tools used. One of the weaknesses of academic research into software engineering is that it usually comes

up with a novel series of methods, involving novel and often difficult languages which have little relationship with what industry currently does and uses and supported, if at all, by poorly built and conceived tools. The expectation is that industry will see the benefits of the new, untried, methods and will completely reorganise, retrain and retool to utilise them. Unfortunately this is a high risk strategy and is unlikely to be a serious option. The expense, disruption and risk associated with such a strategy would deter even the most adventurous company.

3.4 Testing based on a computational modelling approach.

The process of software design, including within that activity all phases of requirements capture, specification, design, prototyping, analysis, implementation, validation, verification and maintenance is one that is oriented, or should be, around the construction of computational solutions to specific problems.

A fundamental strategy.
When we are constructing a software system (this also applies to hardware) we are attempting to construct something that will, when operating, carry out some computable function. Consequently it is worth considering what this means. Essentially, computable functions have been identified as the functions computed by Turing machines. The only assumption we will make is that whatever implementation we produce it can be considered to behave like a Turing machine that halts. In other words we will not try to deal with those systems that regress into an infinite loop from which no output emanates, for our purposes these systems will be deemed to be unacceptable anyway. A way to establish that a system is not of this form is to identify a period of time which is the maximum that the system can run for without producing any detectable output. We will also assume that the specification of the system is also of this form, namely a Turing machine that halts under its intended operating conditions. Real-time systems are covered by this definition since we require that the specified system does have detectable behaviour under all conditions. This is a kind of *design for test* condition that we will see more of later.

We then have two algebraic objects, the Turing machine representing the specification of the desired system and the Turing machine representing the complete implementation. A testing method would then try to ascertain if these two machines computed the same function. This is a basic strategy that we will develop, however, not in the context of a Turing machine which is too low level and unwieldy, but in the context of a more useful, elegant and equivalent model.

In so doing we will quote some important theoretical results that justify what we are doing. It is important to stress that the method of finite state machine testing proposed by Chow [76], and developed by a number of other authors since, e.g.. [77], is based on a similar sort of philosophy, the difference being that they have to make very strong assumptions about the nature of the implementation machine. However, their work did act as an important inspiration for our own.

Finite state machines revisited.
Finite state machines have been a popular and effective way of specifying and designing a number of different types of system. For example, simple control systems and real time systems are often described in this way, so are hardware devices - particularly sequential systems and some aspects of microprocessors. The ease with which finite state machines can be used to convey the dynamics of user interaction in elementary menu systems has been exploited in many examples. The definition of a

number of protocols for communications systems as finite state machines is also standard practice. The control structure of simple software systems can also be described using this modelling technique and automatic generation of executable code is possible in some simple cases. Often these machines are used in conjunction with other methods, for example in systems analysis they are often combined with data modelling techniques.

Some of the advantages of finite state machines as a specification language are:

- they are intuitive, being based on a simple, dynamic model of computation;
- they can be represented in convenient ways such as diagrams and tables;
- a lot is known about the theory of these machines;
- many people have met them in University courses.

There are a number of disadvantages which we will discuss later.

We begin with a restatement of a basic definition.

Definition 3.4.1 A *finite state machine with outputs* consists of:

A finite set Q, of internal states;
An initial state, q_0;
A finite set *Input*, of inputs;
A finite set *Output*, of outputs;
A next state function $\mathbf{F} : Q \times Input \rightarrow Q$;
An output function $\mathbf{G} : Q \times Input \rightarrow Output$.
Typically, finite state machines are described using diagrams or tables.

We will sometimes say $M = (Q, Input, Output, \mathbf{F}, \mathbf{G}, q_0)$ is a finite state machine with outputs.

Example 3.4.1

Consider a simple machine with 4 states.

Let *Input* = { *a, b* } and *Output* = { *x, y* }

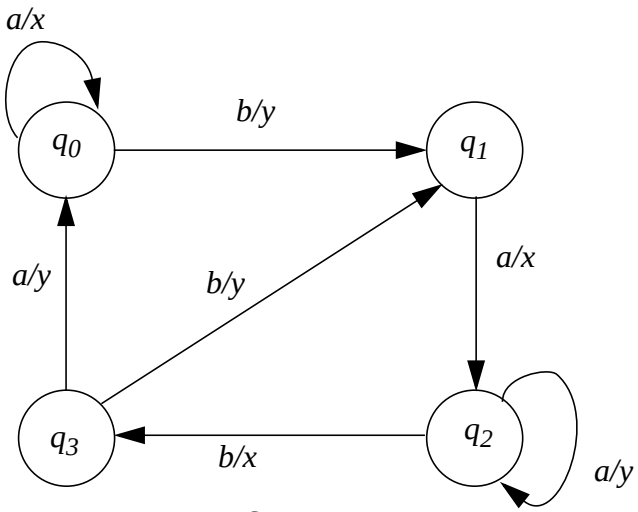

Figure 3.2 A simple finite state machine.

recall from Chapter 1 the convention in the diagram where the arrows and their labels indicate the functions **F** and **G**. For example, the arrow labelled by *b*/*y* from state q_0 to q_1 tells us that:

$\mathbf{F}(q_0, b) = q_1$
$\mathbf{G}(q_0, b) = y$

We might regard q_0 as an initial state, perhaps corresponding to an idle or off state in a system.
If the system is started in state q_0 and supplied with an input of *a*, the machine outputs *x* and remains in q_0;
if presented with an input *b* while in state q_0 it moves to state q_1 and outputs *y*. And so on.

Putting in a stream of inputs, say *a*::*b*::*a*::*b*::*a*::*b* the result is a transversal of the diagram to state q_1 producing, along the way, the output stream *x*::*y*::*x*::*x*::*y*::*y*. If there is a state where a given input produces no result the machine will halt, e.g.. *a*::*b*::*b* starting from q_0 halts in state q_1 after the second input is read.

Testing based on finite state machines.
The basic idea is that we have a specification described in terms of a finite state machine. We also have an implementation which we wish to establish is correct with respect to the specification. If we can assume that the implementation is also a finite state machine then we reduce the verification problem to:
establishing whether the two finite state machines - the specification and the implementation - behave in an identical manner.

There is some theory that can assist in this task, [76]. The essential ideas are based around deriving sequences of inputs that can detect whether there are any extra states or missing states in the implementation, whether there are any faulty or missing transitions and so on. The method proceeds by constructing suitable sequences of inputs that will produce different outputs in the two machines when applied to them both- if the implementation is faulty and these types of fault are present.

Constructing a test set.
The test generation process proceeds by examining the state diagram, minimising it (a standard procedure) and then constructing a set of sequences of inputs. This set of sequences is constructed from certain preliminary sets, the basis being the theory of Chow. It is to this work that we must now turn. We will sketch the procedure in this chapter, introducing a number of concepts relatively informally. More details can be found in Chapters 6 and 7.

Definition 3.4.2. Let *L* be a set of input sequences, and *q*, *q*' two states then *L* is said to *distinguish* between *q* and *q*' if there is a sequence *k* in the set *L* such that the output obtained when *k* is applied to the machine in state *q* is different to the output obtained when *k* is applied when it is in state *q*'. If the two state *q* and *q'* are not distinguished by *L* we say that they are *L-equivalent*.

Definition 3.4.3. A machine is called *minimal* if it doesn't contain redundant states. The formal defini-

tion is Definition 7.1.6.

There are algorithms that produce a minimal machine from any given machine - the minimal machine has the same behaviour in terms of input-output as the original.

Example 3.4.2

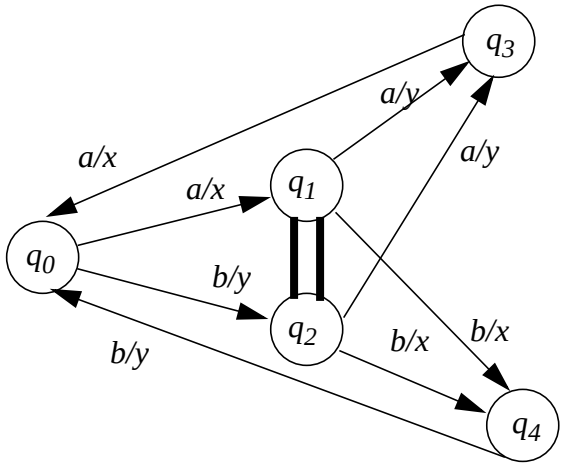

Figure 3.3 A simple finite state machine which is not minimal.

In this machine states q_1 and q_2 can be *merged* to form a machine with fewer states and the same input-output behaviour.

Let us consider, from now on, a minimal finite state machine.

Definition 3.4.4 A set of input sequences, W, is called a *characterisation set* if it can distinguish between any two pairs of states in the machine.

Example 3.4.3 In the machine in Example 3.4.1 $W = \{ a, b \}$ is a characterisation set (the machine is minimal).

Definition 3.4.5 A *state cover* is a set of input sequences L such that we can find an element from L to get into any desired state from the initial state q_0.

$L = \{ <>, b, b::a, b::a::b \}$ is a state cover for the machine in Example 3.4.1, <> represents the null input.

Definition 3.4.6 A *transition cover* for a minimal machine is a set of input sequences, *T*, which is a state

cover and is closed under right composition with the set of inputs *Input*, so we can get to any state from q_0 by using a suitable sequence *t* from *T* and for any input *a* in *Input* the sequence *t* :: *a* is in *T*.

Here *T*= { < >, *a, b, b*::*a, b*::*b, b*::*a*::*a, b*::*a*::*b, b*::*a*::*b*::*a, b*::*a*::*b*::*b* } is a transition cover for Example 3.4.1.

There are algorithms for constructing all of these sets for specific machines.

Generating a test set.
We first need to estimate how many more states there are in the implementation, than in the specification. Let us assume that there are k more states and let *W* be a characterisation set.

We construct the set

$$Z = Input^{k} \bullet W \cup Input^{k-1} \bullet W \cup .... \cup Input^{1} \bullet W \cup W,$$

recall that if A and B are sets of sequences from the same alphabet then A• B means the set of sequences that are constructed from a sequence from A followed by a sequence fromB. Thus the set *Z* is formed from a collection of sets where we first form the set of sequences obtained by using all input sequences of length k followed by sequences from *W*, then add to this collection the sequences formed using input sequences of length *k-1* followed by sequences from *W*, continue building up a set of sequences in this way.

The final *test set* is: $T \bullet Z$ where *T* is a transition cover.

The basic theorem (Chow) states:

Theorem 3.4.1 If we have two minimal finite state machines, *M* and *M '*,one with initial state q_0 and the other with initial state q'_0, and *T* is a transition cover and *W* a characterisation set for one of these machines then they are isomorphic (behave identically) if q_0 and q'_0 are $T \bullet Z$-equivalent. This means that if they produce the same outputs when the sequences from *T.Z* are applied then they are equivalent machines.

Note that *T.Z* is a *finite set of finite sequences*. We can therefore say something quite strong about the second machine, the implementation. If it behaves as the specification requires on *all* the input sequences from the test set $T \bullet Z$ it is then correct in the sense that for *every circumstance* defined by the specification the implementation produces the expected result.

Example 3.4.3.

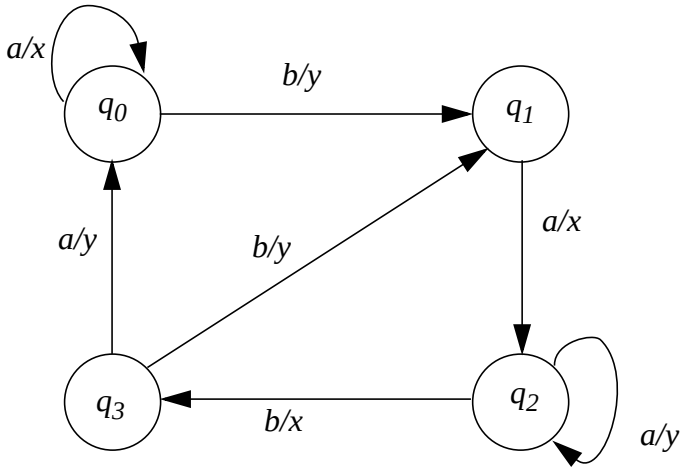

Figure 3.4 A simple machine.

The following represents an implementation with one extra state, a missing transition and a faulty transition label.

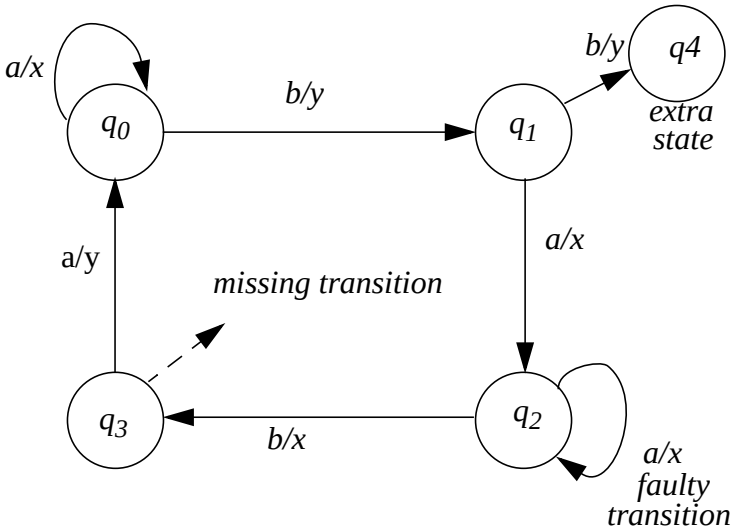

Figure 3.5 A faulty version of the machine in Figure 3.4

The value of *k* is assumed to be 1 for this example, then $Z = Input \bullet W \cup W = \{ a, b \}. \{ a, b \} \cup \{ a, b \} = \{ a, b, a::a, a::b, b::a, b::b \}$ and the test set $T \bullet Z$ is thus:

$T \bullet Z = \{ <>, a, b, b::a, b::b, b::a::a, b::a::b, b::a::b::a, b::a::b::b \} \bullet \{ a, b, a::a, a::b, b::a, b::b \}$

$= \{ a, b, a::a, a::b, b::a, b::b,$
$a::a, a::b, a::a::a, a::a::b, a::b::a, a::b::b,$
$b::a, b::b, b::a::a, b::a::b, b::b::a, b::b::b,$
$b::a::a, b::a::b, b::a::a::a, b::a::a::b, b::a::b::a, b::a::b::b,$
$b::b::a, b::b::b, b::b::a::a, b::b::a::b, b::b::b::a, b::b::b::b,$
$b::a::a::a, b::a::a::b, b::a::a::a::a, b::a::a::a::b, b::a::a::b::a, b::a::a::b::b,$
$b::a::b::a, b::a::b::b, b::a::b::a::a, b::a::b::a::b, b::a::b::b::a, b::a::b::b::b,$
$b::a::b::a::a, b::a::b::a::b, b::a::b::a::a::a, b::a::b::a::a::b, b::a::b::a::b::a, b::a::b::a::b::b,$
$b::a::b::b::a, b::a::b::b::b, b::a::b::b::a::a, b::a::b::b::a::b, b::a::b::b::b::a, b::a::b::b::b::b \}$

= {*a, b, a*::*a, a*::*b, b*::*a, b*::*b, a*::*a*::*a, a*::*a*::*b, a*::*b*::*a, a*::*b*::*b, b*::*a*::*a, b*::*a*::*b, b*::*b*::*a, b*::*b*::*b, b*::*a*::*a*::*a, b*::*a*::*a*::*b, b*::*a*::*b*::*a, b*::*a*::*b*::*b, b*::*b*::*a*::*a, b*::*b*::*a*::*b, b*::*b*::*b*::*a, b*::*b*::*b*::*b, b*::*a*::*a*::*a*::*a, b*::*a*::*a*::*a*::*b, b*::*a*::*a*::*b*::*a, b*::*a*::*a*::*b*::*b, b*::*a*::*b*::*a*::*a, b*::*a*::*b*::*a*::*b, b*::*a*::*b*::*b*::*a, b*::*a*::*b*::*b*::*b, b*::*a*::*b*::*a*::*a*::*a, b*::*a*::*b*::*a*::*a*::*b, b*::*a*::*b*::*a*::*b*::*a, b*::*a*::*b*::*a*::*b*::*b, b*::*a*::*b*::*b*::*a*::*a, b*::*a*::*b*::*b*::*a*::*b, b*::*a*::*b*::*b*::*b*::*a, b*::*a*::*b*::*b*::*b*::*b* }

on removing repetitions.

The extra transition is exposed by the input *b*::*b* which produces an output *y* - in the specification and *y*::*y* in the implementation; the missing transition is exposed by *b*::*a*::*b*::*b* and the faulty transition by *b*::*a*::*a*.

The transition cover ensures that all the states and transitions of the specification are present in the implementation and the set *Z* ensures that the implementation is in the same state as the specification after each transition is used. The parameter *k* ensures that all the extra states in the implementation are visited.

An algorithm for determining the transition cover.
Much of the process for generating and applying test sets to real cases can be automated. We consider, here, the case of constructing the transition cover. This is revisited in more detail in Chapter 7.

We first build a *testing tree* with states as node labels and inputs as arc labels.
From each node there are arcs leaving for each possible input value. The root is labelled with q_0. This is level *0*.
We now examine the nodes at level *m* from left to right;

if the label at the node is a repeat of an earlier node then terminate the branch;
if the node is labelled "*undefined*" then terminate that branch.
if the label at the node is a state such that an input s is not defined then an arc is drawn, labelled by *s*, to a node labelled "*undefined*".
if an input s leads to a state *q'* then insert an arc, labelled by *s*, to a node labelled *q'*.

Example 3.4.4

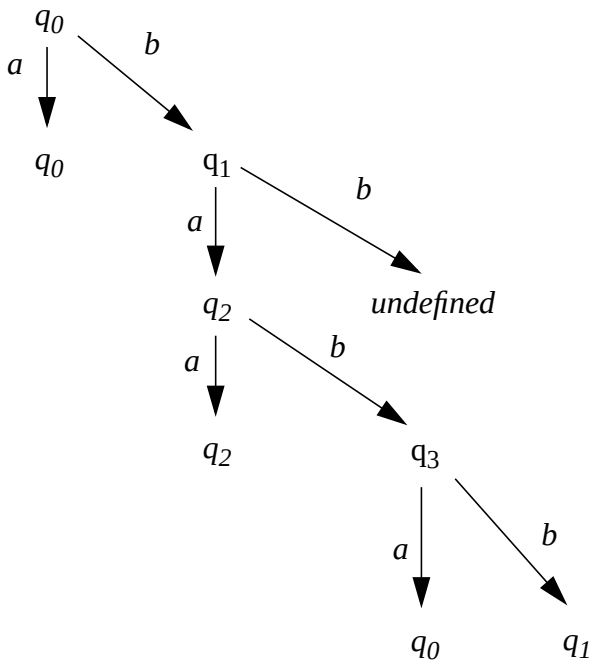

Figure 3.6 The testing tree for the machine in Figure 3.4

The labels of all partial paths in the tree form the transition cover.

Various authors have devised improved methods based on the use of fewer input sequences in the test set or by combining test sequences together to improve efficiency. Usually, the gain in obtaining a smaller test set is partly offset by the increased complexity of the process of generating it. These issues are revisited in Chapter 7.

An evaluation of finite state machine testing.
Unfortunately, Chow's method is only directly applicable to finite state machines and not to more complex machines involving explicit data processing and internal memory. It is often difficult to model many systems using finite state machines alone in a compact manner. The method can be used to test the control structure of some complex systems with the data structure and processing functions being tested in some other way. This last method is unrealistic except in very special cases since the control is rarely independent of data state. By expanding the state space massively it is possible to construct better models but they rapidly become unusable. The assumption in the theorem that the implementation is a minimal machine can be removed easily. The assumption that the implementation is a finite state machine (that is, there is no hidden memory) is very doubtful in practice. In some cases, for example, in communications protocol testing, very strong assumptions are typically made, e.g. $k = 0$ is required for some of the methods for generating more efficient test sets to work.

The use of Statecharts [31] has improved the capability of finite state machine modelling at the expense, however, of a coherent semantics. Statecharts also lack a convenient method for describing the semantics of the individual transitions, some extensions have been introduced [33] which provide a more powerful modelling language. using these extended versions of statecharts, some considerable progress

has been made on developing a powerful testing method, see Bogdanov & Holcombe [78].

3.5 Stream X-machines and the fundamental theorem of testing.

Stream X-machines (recap).
The results that we will discuss are based on the class of stream X-machines. These represent a very wide class of systems and can, essentially, describe all feasible computing systems.

Recall that a stream X-machine has an *input* set, an *output* set, a *memory* set, a set of *states* and a collection of *transitions* labelled by *basic* (or processing) *functions* that are dependent on the input and the memory.

Each transition function removes the head of the input stream and adds an element to the rear of the output stream, and, furthermore, no transition is allowed to use information from the tail of the input or any of the output. m_0 is the *initial* state of the memory.

For a stream *X*-machine to be *deterministic*, there must be a single start state and the set of basic functions, Φ must be such that given any state and any input value and any memory value there is only one function that can be applied. Formally this is expressed as:

Definition 3.5.1 A stream X-machine, M , is *deterministic* if
$\forall\ \phi, \phi' \in \Phi$, if $\exists\ q \in Q, m \in M, in \in Input$ such that $(q, \phi) \in$ domain F, $(m, in) \in$ domain ϕ and $(m, in) \in$ domain ϕ', then $\phi = \phi'$.
(Here domain *f* refers to the domain of a partial function *f*.).

So each computation from the initial state to any other state is completely determined by the input sequence and the initial memory value. A deterministic stream *X*-machine will compute a partial function f: $Input^* \rightarrow Output^*$.

We proceed now with some general strategic remarks. The theory of finite state machines, as we have seen from section 3.4, includes a result that describes how to test whether two finite state machines are isomorphic. Isomorphism means that they are algebraically similar and if we wish we can convert from one to another by using a “renaming” which respects the algebraic structure and the behaviour of the machines. Under these conditions their behaviour is the same. We can convert an *X*-machine into a finite state machine by treating the elements of Φ as abstract input symbols. We are, in effect, “forgetting” the memory structure and the semantics of the elements of Φ. If we call this the *associated automata* of the *X*-machine we have the following result:

Theorem 3.5.1 Let M and M' be two deterministic stream *X*-machines with the same set Φ of basic functions, f and f' the functions computed by them and let A and A' be their associated automata. If A and A' are isomorphic then $f = f'$.
Proof. See [79].

Now proving that the two functions computed by the two stream *X*-machines, one the specification and the other the implementation, are equal is precisely what we want. If we can do this with a finite test set we will have a very powerful method indeed. This is our aim. To achieve this we need to consider how to prove that the associated automata of two stream *X*-machines are isomorphic.

Drawing on the techniques used in state machine testing will enable us to progress with this aim. We need some further terminology for stream *X*-machines.

Design for test conditions.
It turns out that if we wish to get the most from our testing strategy there are certain constraints that must be observed. In hardware design the idea of design fir test is well understood. If the designer ignores the fact that the system will eventually need testing he/she can create a design that is very hard, if not impossible, to test thoroughly. The same is true for software. The following conditions represent a formalisation of the idea of design for test. They are conditions that must be satisfied if the complete test set we will be introducing is to be constructed. They do not result in any limitation since any stream X-machine can be made to satisfy these conditions - at the cost of including some extra test based functionality.

Now consider any basic function $\phi \in \Phi$, so ϕ: $M \times Input \rightarrow Output \times M$, suppose that m is any memory value that can be attained, is it possible to find an input $in \in Input$ that could cause this function ϕ to operate? This is motivation for the following definition.

Definition 3.5.2 A type Φ, is called *test-complete* (or *t-complete*) if $\forall\ \phi \in \Phi$ and $\forall\ m \in Memory$, $\exists\ in \in Input$ such that $(m, in) \in$ domain ϕ.

This condition prohibits “dead-ends” in the machine. We can turn an X-machine into one which is t-complete by introducing special test inputs and extending those functions that are not t-complete by defining them to behave suitably when these special test inputs are applied. The test inputs are not used during normal operation.

Now consider the case when a basic function has operated in a given state with a memory value and an input. we observe an output. What caused this output, we know it was a basic function but which one? We cannot see these directly, only through their effect on the output and so we must ensure that there is no other basic function which could have produced the same output under identical conditions. This leads to our other condition.

Definition 3.5.3 A type Φ is called *output-distinguishable* if:
$\forall\ \phi_1, \phi_2 \in \Phi$, if $\exists\ m \in M, in \in Input$ such that $\phi_1(m, in) = (out, m_1')$ and $\phi_1(m, in) = (out, m_2')$ with m_1', $m_2' \in M, out \in Output$, then $\phi_1 = \phi_2$.

What this is saying is that we must be able to distinguish between any two different processing functions in an X-machine by examining outputs. If we cannot then we will not be able to tell them apart. So

we need to be able to distinguish between any two of the processing functions (the ϕ's) for all memory values. The mechanism for achieving output distinguishability is by introducing some special test outputs which are used in those cases where two functions would not normally be distinguishable. So some of functions are augmented with these special indicator outputs that are only used in testing. In normal usage they will never appear.

These two conditions are required of our specification machine, we shall refer to them as *design for test* conditions. They are quite easily introduced into a specification by simply extending the definitions of suitable Φ functions and introducing extra output symbols. These will only be used in testing. As we will see in Chapter 7, these definitions only require checking with respect to *attainable* memory, the subset of memory that can be reached from the initial state.

Now we consider how test sets can be constructed that will show that the two associated automata are isomorphic. Consider the associated automata of the X-machine, M . This is a finite state machine, A with the basic function labels from the set Φ labelling the machine transitions. We now apply the Chow method of generating a test set using these labels instead of inputs. So we create a transition cover and a characterisation set in exactly the same way and form a test set as in 3.4. We have constructed a set of sequences of elements from Φ^* that will be the basis for our testing process, a set that will establish whether the two stream X-machines compute the same function. If T and W are a transition cover and a characterisation set, respectively, of the associated automaton A of M and we put

$$Z = \Phi^k \bullet W \cup \Phi^{k-1} \bullet W \cup ... \cup W,$$

where k is the estimated number of extra states in the implementation compared to the specification. Then $T \bullet Z$ is the set we have constructed to act as the test set for the associated automata.

However, this set is not really very convenient, we really want a set of *input* sequences from *Input** rather than a set of basic function labels from Φ^*. We thus need to convert sequences from Φ^* into sequences from *Input**. In some cases there is a one to one correspondence between the basic functions and the inputs. Then the transformation of the set of function sequences into sets of input sequences is immediate. We do this transformation, in the general case, by using a fundamental test function, $t : \Phi^* \rightarrow Input^*$, defined recursively with reference to the specification machine. This is defined next. Note that the test function is not uniquely determined, many different possible test functions exist and it is up to the designer to construct it. There is an issue of choosing one which is of maximal efficiency, a subject that we have yet to investigate. However it is very likely that a test function could be constructed and evaluated automatically.

The test functions.
What we have to do is replace the test sequences of function labels with sequences of machine inputs that will cause these sequences of functions, i.e. paths in the X-machine, to operate. This is done by constructing a function $t : \Phi^* \rightarrow Input^*$ that carries out this translation.

Consider a sequence of basic functions $\phi_1::...::\phi_{n+1}$ either this represents a path in the machine starting at the state q with memory value m.

$$q \xrightarrow{\phi_1} q_2 \xrightarrow{\phi_2} q_3.... \; q_{n+1} \xrightarrow{\phi_{n+1}} q_{n+2}$$

or there is no such path.

In the former case we look at the function $\phi_1$and choose an input in_1 with the property that this input together with the memory value m is in the domain of ϕ_1 (that is, in_1 causes ϕ_1 to operate in this state q and memory value m). The t-completeness of Φ ensures that this is possible. Now we look at the next function in the sequence, ϕ_2 and repeat the process and so on.

If the sequence of basic functions does not represent a path then we have two possibilities. Consider the sequence with the last element removed. If this does represent a path then we choose an input as above and terminate the process. We now have a sequence of inputs that exercises the path up to the penultimate element and then the final input causes the operation to halt as there is no transition triggered by that input. If the sequence with the last element removed still does not represent a path we remove the last element of the sequence and obtain a sequence ϕ_1::...:: ϕ_n and repeat the procedure until we have a path and then proceed as above.

This process of converting sequences of basic functions into sequences of inputs can be represented by a (recursively defined) function $t_{q,m} : \Phi^* \rightarrow Input^*$ (see chapter 7). Then $t_{q,m}$ is called a *test function* of M w.r.t. q and m. If $q = q_o$ and $m = m_o$, $t_{q,m}$ is denoted by t and is called a *fundamental test function of* M.

If

$$q \xrightarrow{\phi_1} q_2 \xrightarrow{\phi_2} q_3 .... q_n \xrightarrow{\phi_n} q_{n+1}$$

is a path in M , then $s = t_{q,m}(\phi_1 ... \phi_n)$ will be an input string which, when applied in q and m, will cause the computation of the machine to follow this path. If there is no arc labelled ϕ_{n+1} from q_{n+1}, then $t_{q,m}(\phi_1 ... \phi_n \phi_{n+1}) = s :: in$, where in is an input which would have caused the machine to exercise such an arc if it had existed (i.e. therefore making sure that it does not exist). Also, for all $\phi_{n+2}, ..., \phi_{n+k} \in \Phi$, $t_{q,m}(\phi_1 ... \phi_n \phi_{n+1} ... \phi_{n+k}) = t_{q,m}(\phi_1 ... \phi_n \phi_{n+1})$ (i.e. therefore only the first non-existing arc in the path is exercised by the value of the test function).

We can say that a test function tests whether a certain path exists or not in M (hence the name).

Once all of this mechanism is in place we can generate a test set mechanically. Thus we construct explicitly **T** and **W** the transition cover and the characterisation set, respectively, of the associated automaton of the specification and then generate the set:

$$\mathbf{Z} = \Phi^k \bullet \mathbf{W} \cup \Phi^{k-1} \bullet \mathbf{W} \cup ... \cup \mathbf{W}$$

and form the set of input strings $t(\mathbf{T} \bullet \mathbf{Z})$, where t is the fundamental test function as described above. This is the test set we are seeking. The value of k is chosen to represent the difference between the known state size of the specification and the (unknown) state size of the implementation. In practice this is not usually large, for especially sensitive applications one can make very pessimistic assumptions about k at the cost of a large test set.

Thus the test set is $t(\mathbf{T} \bullet \mathbf{Z})$ and this is a set of input sequences that will be applied to any implementa-

tion of the specification that we are trying to validate.

The fundamental theorem of testing, [79] (see Chapter 7) allows us to conclude that if all the tests are passed then the implementation is correct subject to the assumption that the basic functions have been correctly implemented in the implementation. We will examine the consequences later in this section.

Testing the Estimator example.
The Estimator example considered in the previous chapter satisfies the design for test conditions and so we can apply the test generation method immediately since the machine is deterministic and minimal.

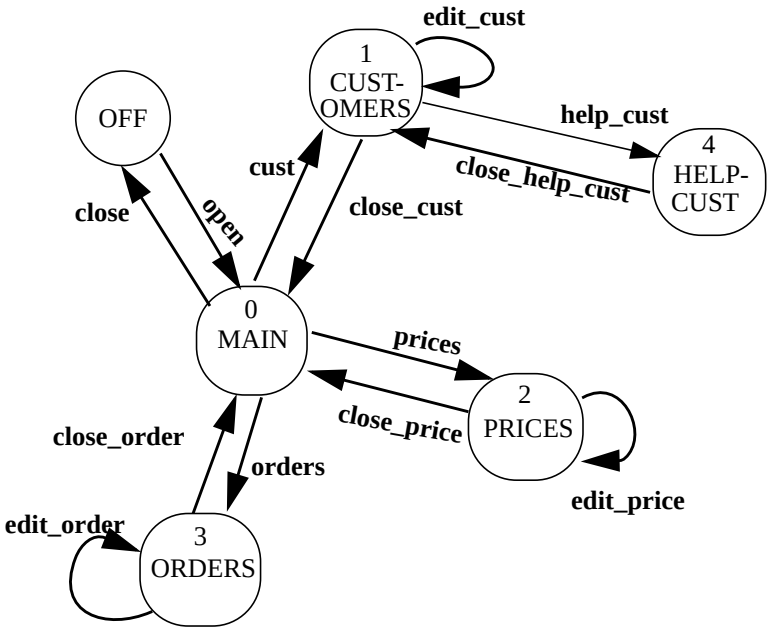

Fig. 3.5 The **Estimator** - top level system design.

The Estimator test set, where Φ = {**close, open, cust, close_cust, edit_cust, help_cust, close_help_cust, prices, edit_price, close_price, orders, edit_order, close_order**}

State cover: {< >, **close** , **cust, help_cust**, **cust**, **prices**, **orders** }

W consists of 5 sequences: {**close**, **open, close_cust, close_help_cust, edit_price** }

Fundamental test set: 105 sequences

close close
open
close_cust
close_help_cust
edit_price

close open close
close close_cust
close close_help_cust
close edit_price
cust close
cust open
cust close_cust close
cust close_help_cust
cust edit_price
edit_cust
help_cust
prices close
prices open
prices close_cust
prices close_help_cust
prices edit_price close
close_price
orders close
orders open
orders close_cust
orders close_help_cust
orders edit_price
edit_order
close_order
close open open
close open close_cust
close open close_help_cust
close open edit_price
close cust
close edit_cust
close help_cust
close prices
close close_price
close orders
close edit_order
close close_order
cust help_cust close
cust help_cust open
cust help_cust close_cust
cust help_cust close_help_cust close
cust help_cust edit_price
cust help_cust cust
cust help_cust edit_cust
cust help_cust help_cust
cust help_cust close_help_cust open
cust help_cust close_help_cust close_cust
cust help_cust close_help_cust close_help_cust
cust help_cust close_help_cust edit_price
cust help_cust prices
cust help_cust close_price
cust help_cust orders
cust help_cust edit_order
cust help_cust close_order
cust cust
cust close_cust open
cust close_cust close_cust
cust close_cust close_help_cust
cust close_cust edit_price

cust edit_cust close
cust edit_cust open
cust edit_cust close_cust
cust edit_cust close_help_cust
cust edit_cust edit_price
cust prices
cust close_price
cust orders
cust edit_order
cust close_order
prices cust
prices edit_cust
prices help_cust
prices prices
prices edit_price open
prices edit_price close_cust
prices edit_price close_help_cust
prices edit_price edit_price
prices close_price close
prices close_price open
prices close_price close_cust
prices close_price close_help_cust
prices close_price edit_price
prices orders
prices edit_order
prices close_order
orders cust
orders edit_cust
orders help_cust
orders prices
orders close_price
orders orders
orders edit_order close
orders edit_order open
orders edit_order close_cust
orders edit_order close_help_cust
orders edit_order edit_price
orders close_order close
orders close_order open
orders close_order close_cust
orders close_order close_help_cust
orders close_order edit_price

In this list, which was generated by a test generation tool developed for this method, **open close_help_cust** means **open :: close_help_cust .** From this set of function sequences we can generate the complete test set by replacing the functions by their corresponding inputs. Thus **open** is replaced by *on*, **cust** by *1*, **close_cust** by *close_1* etc..

The example examined above was such that there was a one to one correspondence between inputs and basic functions. This is not always the case and then the construction of the test functions is more complicated. Some examples of these cases will be found in Chapter 7.

Reflections on the test generation method.
For the method to work we must make the following further assumptions:

1. The specification is a deterministic stream *X*-machine;
2. The set of basic functions Φ is output-distinguishable and t-complete;
3. The associated automata are minimal;
4. The implementation is a deterministic stream *X*-machine with the same set of basic functions Φ.

Of these assumptions the first three lie within the capability of the designer. An algorithm for ensuring that a stream *X*-machine satisfies condition 2 is given in Ipate & Holcombe [80]. Any stream *X*-machine can be replaced by a machine satisfying condition 2. We refer to these conditions as "design for test" conditions, without them it is going to be difficult to test a system properly, there may be hidden behavioural faults in the implementation which cannot be exposed. The procedures are quite straightforward and intuitive.

Condition 3 requires some comment. It is clear that the designer can arrange for the associated automata of the specification *X*-machine to be minimal, standard techniques from finite state machine theory are available. The problem remains with the requirement that the implementation's associated automata is minimal. Since we do not have an explicit description of the implementation as an *X*-machine we cannot analyse its associated automata to see if it is minimal. We do know, however, that there is a minimal automata with the same behaviour as the automata of the implementation. It is this that will feature in the application of the fundamental theorem. Thus we have a test set that determines whether the behaviour, that is the function computed, by the specification equals the function computed by the implementation - providing that both implementation *X*-machine and the specification *X*-machine have the same basic function set Φ.

The final condition is the most problematical. Establishing that the set of basic functions, Φ, for the implementation is the same as the specification machine's has to be resolved, however. In practice this will be done with a separate testing process, either an application of the method explained above since the basic processing functions are computable and thus expressible as the computations of other, presumably much simpler, *X*-machines or by using some other testing method for testing simple functions - perhaps the category-partition method [61] or a variant. If the basic processing functions are tried and tested with a long history of successful use - perhaps they are standard procedures, modules or objects from a library, then their individual testing could perhaps be assumed done. If we assume that the Φs are implemented correctly this carries with it the consequence that the implementation is a stream X-machine since these functions are the processing functions of a stream X-machine by construction.

What sort of restriction is this? It is quite legitimate to assume that the implementation is a deterministic stream *X*-machine because of the computational generality of this class of machines. This does not mean that all Turing machines can be represented as a stream X-machine, but that there exists a hierarchy of stream X-machines with progressively more complex basic functions which will approach the generality of the Turing model. In likely applications of the method we will successively apply the test method to the hierarchy of stream X-machines that are created when we consider the basic functions Φ at each level. Thus, testing a specific function ϕ, will involve considering it as the computation defined by a simpler stream X-machine and so on. This is illustrated in Figure 3.7.

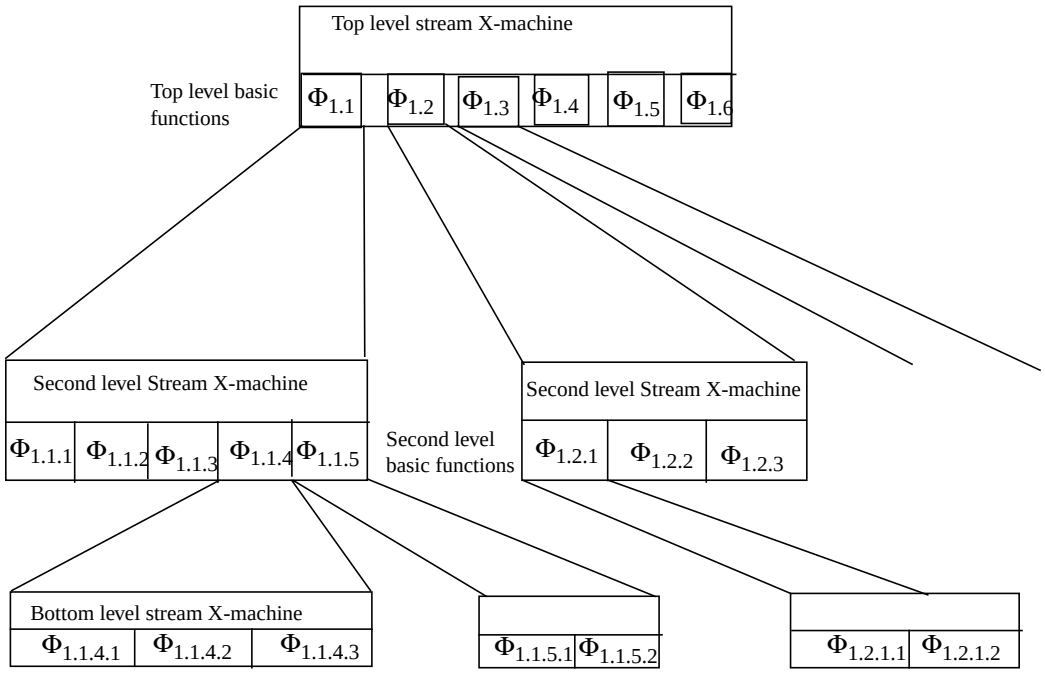

Figure 3.7 Diagram illustrating how the basic functions can themselves be specified using simpler s X-machines in a natural heirarchical manner.

Ultimately, at the bottom level we need to test the basic functions in some suitable way - or assume that they are implemented correctly.

In many of the case studies that we have looked at the basic functions that need to be used are typically very straightforward ones that carry out simple tasks on simple data structures, inserting and removing items from registers, stacks etc., carrying out simple arithmetic operations on simple types and processing character strings in well understood ways. There is probably little point in devoting a great deal of testing effort to this aspect of the problem, it may seem to be slightly dangerous to say so but we know how to implement such simple operations correctly.

The benefits that accrue if the method is applied are that the entire control structure of the system is tested and *all* faults detected *modulo* the correct implementation of the basic functions.

3.6 Discussion.

Let us examine these results in the context of the issues raised in the earlier part of the paper. In particular, does this new method really deliver a principled method of testing? We now can say something about any faults remaining after the successful application of the full test set generated by the method (assuming that the specification X-machine satisfies all of the requirements). Any such faults are restricted to the basic processing functions. This approach does satisfy the reductive requirement that

we expressed at the start. We no longer need to consider test coverage issues since we have full coverage with respect to faults - outside of the set of basic functions Φ.

Recall that the reductionist philosophy means that we can replace the problem of testing for all faults in a stream *X*-machine to one of detecting faults in something simpler - namely the processing functions Φ. These, themselves, can be represented as the computations of even simpler stream X-machines if desired and the reduction continued further. Alternatively these may be functions obtained from a library of dependable objects that the designers are confident are essentially fault-free.

The main point from all of this is that if this testing method is used and the implementation passes all of the tests in the test set then it is known to be free of faults, *providing the Φ's have been implemented in a fault-free fashion*.

One could argue that all we are doing is shifting the major part of the testing burden onto testing the ϕ functions. This not the case, however. The method involves a gradual process of testing based on the hierarchy of the design whereby, at each level in the hierarchy, the functions utilised at that level are assumed to be fault free. The testing at that level then focuses on the correct integration of these functions.

Another important issue concerns the estimate required for the number of states in the implementation. This can be done in a number of ways. if the implementation has been based on the structure of the specification then, providing the transformations into the implementation language have been done systematically and transparently, it should be possible to make this estimate accurately. It is always possible to overestimate the number of states at the expense of a larger than necessary test set. This may be economically justified if the application is of a critical nature.

Alternatively, assuming that the code has been developed in an *ad hoc* nature and the specification is only used as a reference point for compliance then it is possible to analyse the code and to construct its X-machine model from this information by analysing the control flow and the memory (both global and local variables) involved. In fact, recent investigations into extracting a detailed X-machine specification from industrial safety-critical Fortran code has indicated that the process can be substantially automated.

The final question that needs to be addressed is concerned with the practicality of the method. For example, how complex is the test generation algorithm? This is consdered in Chapter 7. The test sets generated by the method for all the case studies we have considered are manageable as is the test application process. Clearly the method has to be supported by automated systems and suitable tools which are under development.

In [81] we proposed the use of X-machines as the foundation of a completely integrated design, verification and test methodology. Because the approach lends itself to a rigorous refinement based development process it is possible to construct test sets for intermediate versions of the implementation,

providing that they satisfy all the design for test conditions. At subsequent stages, after the model has been refined, perhaps by expanding some of the processing functions and reducing the abstraction in a coherent way, the refined machine may be tested in way that utilises test information from the earlier version together with tests generated from the components used to carry out the refinement succinctly.

We have described a new approach to the testing of software systems which is reductionist in the sense that the approach provides a method for generating a finite test set which will detect *all* faults in the system *providing that the basic processing functions are implemented correctly.* The complexity of the test sets is better than that of a number of existing methods. The precondition for the application of the method is a formal specification of the system as a stream X-machine, a method that we and colleagues have introduced recently and evaluated with case studies drawn from a wide variety of different applications.

The scope of the method is quite general and is relevant for real-time applications as well as traditional information system applications. It is independent of the nature of an implementation also. The recent interest in object-oriented systems and the urgent need to develop effective testing methods for them, provides a new and exciting opportunity. The fact that objects are computational phenomena means that we can identify them with suitable classes of X-machines. Representing the control structure more explicitly by way of the rich semantics of the X-machine model will provide a firm basis for the development of a principled testing method for object-oriented software.

Refinement and components.

We have seen that the process of developing the specification incrementally and hierarchically is a convenient and manageable process. It also enables us to distribute the testing effort so that we can test components corresponding to suitable partitions (the submachines defined by the lower level diagrams and functions) and then test their integration using the method applied at the top level. This is more economic, too, as will be shown in Chapter 8 (see also a less general approach in [82]). At some point the developer will be able to use modules, classes, routines from a library (provided that their specification and the environment under which they can be used are known). If these are dependable, perhaps they have been used over a long period and proved reliable, perhaps they have been formally verified correct, they will form the basis of our testing as well as our implementation. In other words we will assume that they are correct. This will then enable us to make conclusions about the correctness of the systems and subsystems that we build with them.

Application of the method.

This method of testing has been used in a number of different situations. The system described in Chapter 5 was built using the complete X-machine specification and testing method. The process was done incrementally, testing the various sub-machines and their integration as separate processes, this exposed a number of faults which were corrected. Beneath it all was the assumption that the low level functions in Access2 etc. were correct. A very robust and sophisticated system was developed over a short period of time, the way in which the method was able to capture the clients real requirements in a simple and effective way, together with the simple nature of the formal language used in the specification and the powerful test generation method are all important strengths of the approach. Using modern software development tools (4GLs and a propriety database system) all add to the attractions and practical nature

of the method. The test generation process was done by hand, but tools have now been developed to automate this. So far the client has reported no faults and is entirely satisfied with the system. His company is now totally dependent on it.

A number of experiments have also been carried out using the method to specify simple but complete systems, and to generate test sets. The systems were then implemented fully and a number of different faults *seeded* into the applications. The faulty systems were then tested using the test sets and in all cases *all* of the faults were detected. Although the systems were small, between 500 and 1000 lines of code in Pascal. The systems included time related ones - an alarm clock; a vending machine; a set of traffic lights, and a calculator example. Furthermore, where faults were also seeded into the basic functions, these were detected by the test set also. Thus the test sets can identify faults in the basic functions. This area needs further research.

Conclusions.

The issue of quality-oriented testing was addressed by insisting on a complete formal specification so that we knew what it was we were trying to deliver. The recognition that software construction is component based and needs to be functionally tested is the motivation for the hierarchical, refinement approach that we have developed. The issue of finding all faults has been addressed through the method by the realisation that if the components are dependable and the implementation passes all the tests then it has the same functional behaviour as the specification - provided that the specification has been augmented to ensure that it satisfies the design for test conditions. It is thus fault free!

Ultimately, at the bottom level we need to test the basic functions in some suitable way - or assume that they are implemented correctly. In many of the case studies that we have looked at the basic functions that need to be used are typically very straightforward ones that carry out simple tasks on simple data structures, inserting and removing items from registers, stacks etc., carrying out simple arithmetic operations on simple types and processing character strings in well understood ways. The benefits that accrue if the method is applied are that the entire control structure of the system is tested and all faults detected modulo the correct implementation of the basic functions. The reductionist philosophy means that we can replace the problem of testing for all faults in a stream X-machine to one of detecting faults in something simpler - namely the processing functions F. These, themselves, can be represented as the computations of even simpler stream X-machines if desired and the reduction continued further. Alternatively these may be functions obtained from a library of dependable objects that the designers are confident are essentially fault-free.

The main point from all of this is that if this testing method is used and the implementation passes all of the tests in the test set then it is known to be free of faults, providing the basic functions (the Φs) have been implemented in a fault-free fashion. If all the basic functions have been formally verified then the total method is one of formal verification, if one basic function has only been validated through testing then the overall process is one of testing.

A further consideration is the cost of this method. It is important that high-quality tools are available to support the specification and test generation process. Even so there is no doubt that the method generates quite large sets, but the benefit from the method is based on what can be deduced if all the tests are passed. Quality does not always come cheap!

Our final conclusion is that if the aim is one of delivering quality systems, and we can construct a specification in the manner described that satisfies the design for test conditions, if, further, we are confident that our bottom level components are correctly implemented, then the test set will find all functional faults, if the system passes all the tests we can stop testing with confidence that the system is as good as its basic components allow.