Almost all software testing is futile!

Mike Holcombe and Florentin Ipate.
Formal Methods and Software Engineering Research Group (FORMSOFT),
Department of Computer Science,
University of Sheffield,
Regent Court,
211, Portobello Street,
Sheffield, S1 4DP, UK.

Abstract. *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.*

Key words: Testing, faults, testability, test coverage, finite state machine testing, *X*-machines.

Introduction.

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 [1] 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 [2] pours scorn on the whole concept of testing and insists that all a test can tell us is that the system has failed - it cannot tell us that the system is correct. 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.[1] However, formal verification has not been able to demonstrate that it is "scalable" and practical in the context of the massive complexity of today's applications. Furthermore, some example systems which have supposedly been formally verified have been shown to have faults as a result of testing! (See [6])

There is a case for hoping that building systems from dependable components will overcome some of these problems 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.

Object-oriented methods are often held to be a great advance in the pursuit of quality software because of their foundation on the principle of reusability. However, it is pointed out in Binder [3] 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 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*!

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.

2. Fundamental issues of correct software design.

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

1. One could imagine a similar paper entitled "Almost all formal verification is futile" exploring some of the fallacies and illusions pertaining to that activity!

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 ***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 ***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".

2.1. 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.

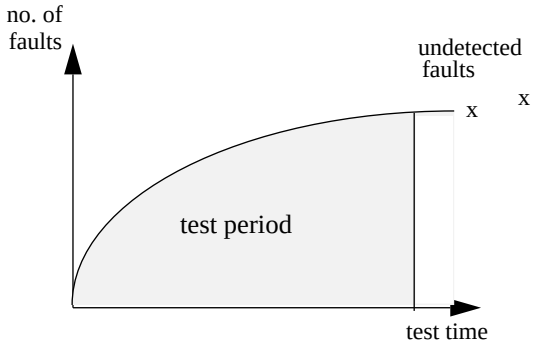


Fig.1 The relationship between fault detection and test time

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 [17], 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?”.

2.2 Test effectiveness.

Other approaches to the problem of measuring the effectiveness of an individual testing project usually depend on estimating the *coverage* of the tests, [1]. For example, if the tests are structurally based, using program charts, say, then popular methods include establishing that every path has been exercised or every decision node has been visited. This does not tell us anything about fault detection, it merely measures *effort* rather than *reward*! The current accepted definition of fault coverage is misleading since it is not the case that a precise measure can be placed on the number of faults that remain after the test process has been applied. In most cases the definition of fault coverage is based on assumptions of the type described above, that we are questioning. In much of the literature the estimates of the fault coverage are obtained by running experiments with simple examples involving implementations that have faults seeded in them and counting the numbers of known faults detected by the methods. These empirical results are of curiosity value only. Miller & Paul, [19] provide a theoretical method for establishing fault coverage of a test strategy, however, they assume that the implementation machine has the same number of states as the specification machine.

As mentioned before, these approaches tend to provide an assurance for the tester rather than a meaningful indication of the success of the tests.

Before we look, briefly, at what progress there has been in addressing some of the theoretical issues of testing we will consider a popular method for the analysis and comparison of the effectiveness of *different* testing methods.

2.3. Test method effectiveness.

A number of authors have sought to compare the effectiveness of, for example, random testing methods with formally based functional testing, e.g.. [4]. Here the method was to take a small system and to insert known faults into it, then to apply the two techniques to establish which was most successful at detecting these faults. Further analysis could be done on the type of faults each method was good or poor at detecting. Faults were 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.

A number of attempts at developing a theory of testing or to analyse the testing situation have been made. We will briefly consider two of them.

2.4. Probable correctness.

Hamlet [5] 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.

2.5. A simple theory of testing.

Goodenough and Gerhard [6] 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.

3. Testing based on an algebraic approach to computational modelling.

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.

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 method will not be applicable to implementations that behave like a Turing machine that does not halt. 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.

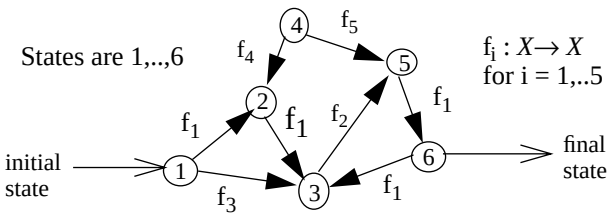
We then have two algebraic objects, the Turing machine representing the specification of the desired system and the Turing machine representing the complete implementation. A testing method would then try to ascertain if these two machines computed the same function. This is a basic strategy that we 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 [7], and developed by a number of other authors since, e.g.. [8], 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. (The IEEE specify finite state machine testing as an important aspect of protocol testing, [18].)

3.1. X-machines.

The model we have chosen, both as a basis for theoretical work in the theory of computability and the theory of testing and as a basis for a formal specification language, is the *X*-machine. Introduced by Eilenberg in 1974, [9], they have received little further study. Holcombe, [10], proposed the model as a possible specification language and since then a number of further investigations have demonstrated that this idea is of great potential value to software engineers. In its essence an *X*-machine is like a finite state machine but with one important difference. A basic data set, *X*, is identified together with a set of basic processing functions, Φ, which operate on *X*. Each arrow in the finite state machine diagram is then labelled by a function from Φ, the sequences of state transitions in the machine determine the processing of the data set and thus the function computed. The data set *X* can contain information about the internal memory of a system as well as different sorts of output behaviour so it is possible to model very general systems in a transparent way. It is best to separate the control state of the system from the data state since this allows much more scope for organising the model to ensure a small and manageable state space. This is done easily with this method, the set *X* is often an array consisting of fields that define internal structures such as registers, stacks, database filestores, input information from various devices, models of screen displays and other output mechanisms. The functions will read inputs, datafiles, internal memory, write to all of these, refresh displays etc..

Consider a simple, abstract example:



We haven't specified *X* or the functions f_i but this is sufficient to provide a basic idea of the machine. It starts in a given, initial state (control state) and a given state of the system's underlying data type *X*, (the data state), there are a number of paths that can be traced out from that initial state, these paths are labelled by functions f_1, f_2 etc. Sequences of functions from this

space are thus derived from paths in the state space and these may be composed to produce a function that may be defined on the data state. This is then applied to the value x, providing that the composed function is defined on x. This then gives a new value, $x' \in X$ for the data state and a new control state. It is best if the machine is deterministic so that at any moment there is only one possible function defined (that is the domains of the functions emerging from a given state are mutually disjoint).
From the diagram we note that there is a possible sequence of functions from state 1 to state 6, here a specified terminal state, labelled by the functions f_1, f_1, f_2, f_1. Assuming that each value is defined this path then transforms an initial value $x \in X$ into the value $f_1(f_2 (f_1(f_1(x)))) \in X$. So we are assuming that $x \in$ domain f_1;

$f_1(x) \in$ domain f_1 ;
$f_1(f_1(x)) \in$ domain f_2;
and $f_2 (f_1(f_1(x))) \in$ domain f_1 .

The computation carried out by this path is thus a transformation of the data space as well as a transformation of the control space.

This is a very general model of computing and a Turing machine can easily be represented in this way, see [9]. In fact it is slightly too general and we now consider a natural subclass of these machines.

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

Those X-machines in which the input and the output sets behave as orderly streams of symbols are called *stream X*-machines and are defined formally in the appendix. The basic idea is that the machine has some internal memory, M, and the stream of inputs determine, depending on the current state of control and the current state of the memory, the next control state, the next memory state and any output value.

So if Σ is the set of possible inputs and Γ represents the set of possible outputs we put

$$X = \Gamma^* \times M \times \Sigma^*$$

Each processing function $\phi : \Gamma^* \times M \times \Sigma^* \rightarrow \Gamma^* \times M \times \Sigma^*$ is of the form whereby, given a value of the memory and an input value, ϕ can change the memory value and produce an output value, the input value is then discarded. There is an initial state and all states are terminals..

The results that we will discuss are based on this class of machines. They represent a very wide class of systems that can describe all feasible computing systems. We proceed now with some general strategic remarks.

The theory of finite state machines 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 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 note that in the automaton all states are terminals. 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:

Let M and M' be two deterministic stream X-machines, f and f' the functions computed by them and A and A' their associated automata. If A and A' accept the same language then $f = f'$.

Proof. See [13].

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 accept the same language.

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.

A type Φ is called *output-distinguishable* if:
$\forall\ \phi_1, \phi_2 \in \Phi$, if $\exists\ m \in M, \sigma \in \Sigma$ such that $\phi_1(m, \sigma) = (\gamma, m_1')$ and $\phi_1(m, \sigma) = (\gamma, m_2')$ with m_1', $m_2' \in M, \gamma \in \Gamma$, then $\phi_1 = \phi_2$.

What this is saying is that we must be able to distinguish between any two different processing functions 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.

A type Φ, is called *complete* if $\forall\ \phi \in \Phi$ and $\forall\ m \in M, \exists\ \sigma \in \Sigma$ such that $(m, \sigma) \in \text{dom}\ \phi$. This condition prohibits "dead-ends" in the machine.

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.

Now we consider how test sets can be constructed that will show that the two associated automata are isomorphic.

Let $\mathbf{V} \subseteq \Phi^*$ be a set of input sequences and let q and q' two states in A and A' respectively. Then, q and q' are said to be **V** - *equivalent* if $\forall\ v \in \mathbf{V}$ then $A\ q$ accepts v iff $A'q$ accepts v, where $A\ q = (\Phi, Q, F, q)$ and $A'q = (\Phi', Q', F', q')$ are the finite state machines obtained from A and A' respectively by considering q and q' as initial states. Thus q and q' are said to be **V**-equivalent if we can always find defined paths labelled by elements from **V** from both q and q'.

Two states q and q' are said to be **V**- *distinguishable* if they are not **V**- equivalent. Thus some v exists in **V** such that either:
a path labelled v exists from q and no path labelled v exists from q';
or a path labelled v exists from q' and no path labelled v exists from q.

The following definitions are derived from the work of Chow [7].

Let $A = (\Phi, Q, F, q_o)$ be a minimal finite state machine. Then a set of input sequences $\mathbf{W} \subseteq \Phi^*$ is called a *characterisation* set of A if **W** can distinguish between any two pairs of states of A. .

Let $A = (\Phi, Q, F, q_o)$ be a finite state machine. Then a set of input sequences $\mathbf{T} \subseteq \Phi^*$ is called a *transition cover* if, for any state $q \in Q$, there is an input sequence $t \in \Phi^*$ which forces the machine A into q from the initial state q_o such that $t \in \mathbf{T}$ and $t\phi \in \mathbf{T}, \forall\ \phi \in \Phi$.

We can thus construct 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. However, this is not really very convenient, we really want a set of input sequences from Σ^*. We thus need to convert sequences from Φ^* into sequences from Σ^*. We do this by using a fundamental test function, $t : \Phi^* \rightarrow \Sigma^*$, defined recursively with reference to the specification machine. This is defined in the appendix. 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 which we have yet to investigate.

We can now assemble our fundamental result which is the basis for the testing method.

The fundamental theorem of testing:

Let M and M' be two deterministic stream *X*-machines with Φ output-distinguishable and complete which compute f and f' respectively and $t : \Phi^* \rightarrow \Sigma^*$ be a fundamental test function of M.

Let $\mathbf{T}$ and $\mathbf{W}$ be a transition cover and a characterisation set, respectively, of the associated automaton A of M and put $\mathbf{Z} = \Phi^k\mathbf{W} \cup \Phi^{k-1}\mathbf{W} \cup ... \cup \mathbf{W}$.

If A and A' are minimal, $\text{card}(Q') - \text{card}(Q) \leq k$ and $f(s) = f'(s)\ \forall\ s \in t(\mathbf{TZ})$, then the associated automata A and A' are isomorphic. If the automata are not both minimal we can still establish that the functions the two machines compute are equal.

Once all of this mechanism is in place we can apply the fundamental theorem to generate a test set mechanically. Thus we construct explicitly $\mathbf{Z} = \Phi^k\mathbf{W} \cup \Phi^{k-1}\mathbf{W} \cup ... \cup \mathbf{W}$ and form the set of input strings $t(\mathbf{TZ})$. 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.

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 complete;
3. The associated automata of the specification machine is 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 [13]. 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 [14] 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.

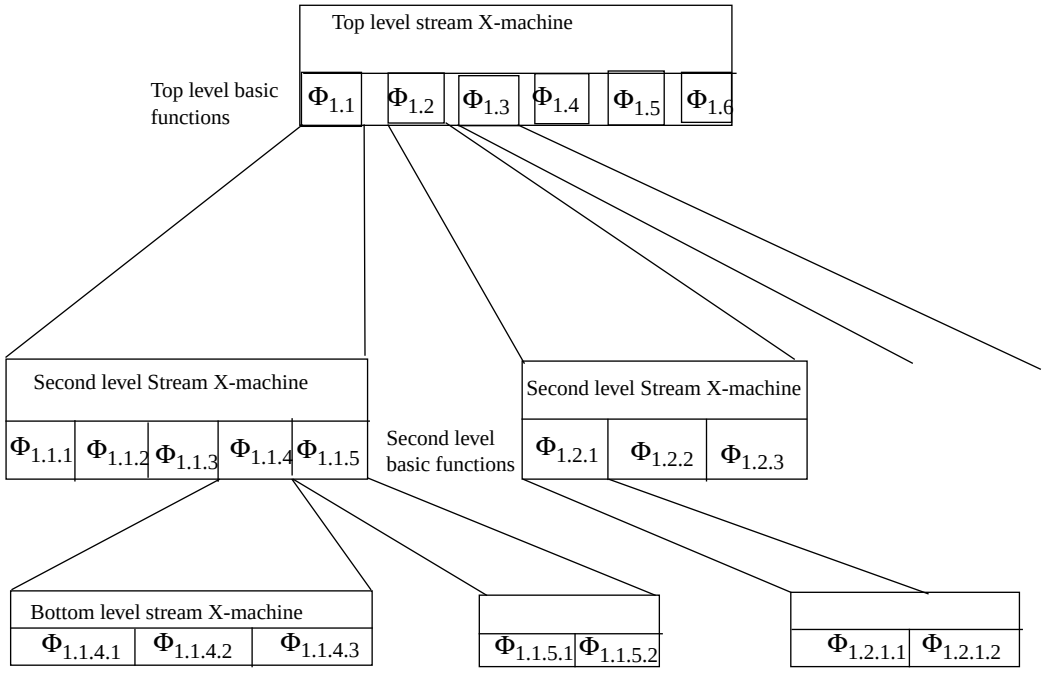


Diagram illustrating how the basic functions can themselves be specified using simpler stream X-machines in a natural heirarchical manner.

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.

4. 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*.

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? A detailed complexity analysis of the algorithm has yet to be carried out and so we cannot provide a theoretical answer to this question yet. However, we can make out the following observations.

Let *A* , the associated automaton of the stream *X*-machine specification, be a minimal finite state machine with n states and card(Φ) = r and let n' be the maximum number of states in the implementation. Then, the method involves generating the values of n test functions t_o, ..., t_{n-1} for domains included in $\mathbf{Z} \cup \Phi\mathbf{Z}$. In the worst case, **W** contains at most n^2 sequences of length n-1 each. Hence, the maximum number of sequences in $\mathbf{Z} \cup \Phi\mathbf{Z}$ is of the order of $n^2 (r+1) r^{n'-n}$, each one is of length no more than n'. Therefore, the maximum size of the set test will be of the order of $n^3 (r+1) r^{n'-n}$, and the maximum length of a test sequence will be n' n. The actual size could be significantly lower, since the domains of the test functions we generate are subsets of $\mathbf{Z} \cup \Phi\mathbf{Z}$ and also since these test functions might not be injective. If the number of the inputs is large (i.e. card(Σ) >> card(Φ)) the size of the test set is considerable lower compared to Chow's method.

Thus test sets generated by the method appear to be manageable as is the test application process. Clearly it has to be supported by automated systems and suitable tools which do not yet exist.

The X-machine specification method also needs further research and evaluation although all those that we know of who have used it have found it to be very intuitive and simple to use. It is also powerful and has been used to construct a detailed specification of a time dependent system [15]. There will clearly be a need to develop the method to deal with large scale systems and a refinement process which allows the development of components and their linking together will be needed. Some of the theoretical work for this has been done [16].

In [11] we propose 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.

Conclusions.

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 claim is made in [3], that "*a state model can provide a conceptual basis for object-oriented testing*". 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.

References.

[1]. G. Myers, "*The art of software testing*.", Wiley, 1979.
[2]. E. Dijkstra, "*A discipline of programming.*" Prentice-Hall, 1976.
[3]. R.V. Binder, "Testing object-oriented systems: a status report." *Proc. STAR94*. 385-392, 1994.
[4]. H. Waeselynck & P. Thévenod-Fosse, "An experimentation with statistical testing." *Proc. EuroSTAR94*, Brussels, 10/1-10/14, 1994.1994.
[5]. R.G. Hamlet, "Probable correctness theory.", *Inf. Proc. Letters*. **25**, 17-25, 1987.
[6]. J.B. Goodenough and S.L. Gerhard. "Toward a theory of test data selection." *IEEE Transactions on Software Engineering*, **1**(2), 156-173, 1975.
[7]. T.S. Chow, "Testing software design modeled by finite state machines." *IEEE Transactions*

on Software Engineering, **4**(3), 178-187, 1978.
[8]. S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou and A. Ghedamsi, "Test selection based on finite state models." *IEEE Transactions on Software Engineering*, **17**(6), 591-603, 1991.
[9]. S. Eilenberg, "*Automata, languages and machines, Vol. A.*", Academic Press, 1974.
[10]. M. Holcombe, "X-machines as a basis for dynamic system specification." *Software Engineering Journal*, **3**(2), 69-76, 1989.
[11]. M. Holcombe, "An Integrated Methodology for the Specification, Verification and Testing of Systems. *Software Testing, Verification and Reliability*, **3**, 149-163, 1993.
[12]. F. Ipate & M. Holcombe, " X-machines with stacks." Departmental Report, Department of Computer Science, University of Sheffield. 1994.
[13]. F. Ipate & M. Holcombe, " X-machine based testing." Departmental Report, Department of Computer Science, University of Sheffield. 1994.
[14]. W. Hetzel, "*The complete guide to software testing*."Wiley, 1984.
[15]. M. Fairtlough, F. Ipate, M. Holcombe, C. Jordan, Z. Duan. "Using an X-machne to specify a video cassette recorder." Departmental Report, Department of Computer Science, University of Sheffield. 1994.
[16]. F. Ipate. "*X-machines - theory and applications in specification and testing*." PhD thesis, University of Sheffield. 1995.
[17].B. Littlewood, "Stochastic reliability growth: a model for fault removal in computer programs and hardware designs." *IEEE Trans. Reliability*, **R-30**, 313-320, 1981.
[18] A.T. Dahbura, K.K. Sabnani & M.U. Uyar, "Formal methods for generating protocol conformance test sequences." *Proc. IEEE*, **78**, 1317-1325, 1990.
[19] R.E.Miller & S.Paul, *IEEE/ACM Trans. Networking*, **2**, 457-470.

Appendix.

A. 1. Stream X-machines.
Let Σ and Γ two finite alphabets (called the input and output alphabet respectively). Then, an *X*-machine $M = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0)$ is called a *stream* X-*machine* if

1. $X = \Gamma^* \times M \times \Sigma^*$, where M is a (possibly infinite) set called the (internal) *memory.*
2. The *type* is
$\Phi = \{\phi: X \leftrightarrow X\}$, where each relation ϕ is defined by:

$$\phi(g, m, s) = \begin{cases} (g\ \rho(m, head(s)), \mu(m, head(s)), tail(s)), & \text{if } s \neq \lambda \\ \varnothing, & \text{otherwise} \end{cases}$$

where $\mu: M \times \Sigma \leftrightarrow M$, $\rho: M \times \Sigma \leftrightarrow \Gamma$ are relations.

3. Q is the finite set of (control) states, I is the set of initial states, T is the set of terminal states. In this paper we will always take $T = Q$.

4. F: Q x $\Phi \rightarrow P(Q)$, where $P(Q)$ denotes the set of subsets of Q, (the power set), defines the next state function.

Note: $\lambda \in \Sigma^*$ is the empty string. For two strings $s, t \in \Sigma^*$, $s::t$ represents the string obtained by concatenation. Also *head*: $\Sigma^* \rightarrow \Sigma^*$ and *tail*: $\Sigma^* \rightarrow \Sigma^*$ are partial functions defined by:
head$(\sigma::s) = \sigma$, $\forall\ \sigma \in \Sigma$, $s \in \Sigma^*$; *head*(λ) is undefined;
tail$(\sigma::s) = s$, $\forall\ \sigma \in \Sigma$, $s \in \Sigma^*$; *tail*(λ) is undefined.

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.

It is clear that each relation ϕ is well determined by ρ and μ. For the sake of simplicity, in what

follows we shall be referring to ϕ as a pair $\phi = (\rho, \mu)$. Then, the type, Φ, will be written as

$\Phi = \{\phi \mid \phi: M \times \Sigma \leftrightarrow \Gamma \times M, \phi = (\rho, \mu)\}$.

For a a stream *X*-machine to be *deterministic*, then *I* must be a single state and $T = Q$, Φ must be a set of partial functions and $\forall\ \phi, \phi' \in \Phi$, if $\exists\ q \in Q, m \in M, \sigma \in \Sigma$ such that $(q, \phi) \in$ dom *F*, $(m, \sigma) \in$ dom ϕ and $(m, \sigma) \in$ dom ϕ', then $\phi = \phi'$. (Here dom *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 $f: \Sigma^* \rightarrow \Gamma^*$.

A. 2. The test functions.
Let $M = (\Sigma, \Gamma, Q, M, \Phi, F, q_o, m_o)$ be a deterministic stream *X*-machine with Φ complete and let $q \in Q, m \in M$. We define recursively a function $t_{q,m} : \Phi^* \rightarrow \Sigma^*$ as follows:

1. $t_{q,m}(\lambda) = \lambda$, where $\lambda \in \Phi^*$ and $\lambda \in \Sigma^*$ are the empty strings.
2. The recursion step depends on the following two cases:
   a. if $\exists$ a path

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

in *M* starting from *q*, then $t_{q,m}(\phi_1 \ldots \phi_{n+1}) = t_{q,m}(\phi_1 \ldots \phi_n)\ \sigma_{n+1}$, where σ_{n+1} is chosen such that $(w(q, m, t_{q,m}(\phi_1 \ldots \phi_n)), \sigma_{n+1}) \in$ dom ϕ_{n+1} and where $w(q, m, t_{q,m}(\phi_1 \ldots \phi_n))$ is the value of the next memory state given the current control state, q, the initial memory value m, and the input string $t_{q,m}(\phi_1 \ldots \phi_n)$. [Note: Since we only apply this construction in the case where Φ is complete, there exists such a σ_{n+1}.]
   b. otherwise, $t_{q,m}(\phi_1 \ldots \phi_{n+1}) = t_{q,m}(\phi_1 \ldots \phi_n)$.

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

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

is a path in *M* , then $s = t_{q,m}(\phi_1 \ldots \phi_n)$ 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 $t_{q,m}(\phi_1 \ldots \phi_n\ \phi_{n+1}) = s::\sigma$, where σ 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 exists).
Also, for all $\phi_{n+2}, \ldots, \phi_{n+k} \in \Phi$, $t_{q,m}(\phi_1 \ldots \phi_n\ \phi_{n+1} \ldots \phi_{n+k}) = t_{q,m}(\phi_1 \ldots \phi_n\ \phi_{n+1})$ (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).

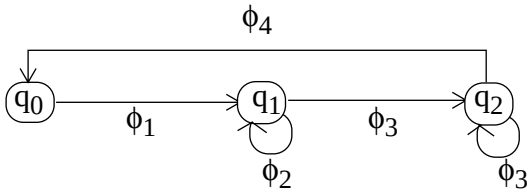
A. 3. The fundamental theorem of X-machine testing:
Let $M = (\Sigma, \Gamma, Q, M, \Phi, F, q_o, m_o)$ and $M' = (\Sigma, \Gamma, Q', M, \Phi, F', q_o', m_o)$ be two deterministic stream *X*-machines with Φ output-distinguishable and complete which compute *f* and *f'* respectively and $t: \Phi^* \rightarrow \Sigma^*$ be a fundamental test function of *M*. .

Let **T** and **W** be a transition cover and a characterisation set, respectively, of the associated automaton A of M and put $\mathbf{Z} = \Phi^k\mathbf{W} \cup \Phi^{k-1}\mathbf{W} \cup ... \cup \mathbf{W}$.
If A and A' are minimal, card(Q') - card(Q) ≤ k and $f(s) = f'(s)\ \forall\ s \in t(\mathbf{TZ})$, then the associated automata A and A' are isomorphic.
Proof. [13].

A. 4. Example.
A simple example of a deterministic stream X-machine satisfying the design for test conditions is:
1. $\Sigma = \{x, y\}$ is the input set,
2. $\Gamma = \{a, b\}$ is the output set,
3. $Q = \{q_o, q_1, q_2\}$ is the set of states; q_o is the initial state and all the states are terminal (i.e. $T = Q$).
4. $M = \{0, 1\}$. The initial memory value is $m_o = 0$.
5. $\Phi = \{\phi_1, \phi_2, \phi_3, \phi_4\}$, where

ϕ_1: $M \times \{y\} \rightarrow \Gamma \times M$ is defined by:
$\phi_1 (m, y) = (a, 1), m \in M$
ϕ_2: $M \times \{x\} \rightarrow \Gamma \times M$ is defined by:
$\phi_2 (m, x) = (a, 0), m \in M$
ϕ_3: $M \times \{y\} \rightarrow \Gamma \times M$ is defined by:
$\phi_3 (m, y) = (b, 1), m \in M$
ϕ_4: $M \times \{x\} \rightarrow \Gamma \times M$ is defined by:
$\phi_4 (m, x) = (b, 0), m \in M$

6. F is represented in the diagram below:



For example, the function, f computed by this machine will, for example, transform the input sequence yxyy into f(yxyy) = aabb and for xy, f(xy) will be undefined etc.
The sequence of state transitions when the input sequence yxyy is applied is:

$(q_0, 0)\ \phi_1 \rightarrow (q_1, 1)\ \phi_2 \rightarrow (q_1, 0)\ \phi_3 \rightarrow (q_2, 1)\ \phi_3 \rightarrow (q_2, 1)$

where

$(q_0, 0)\ \phi_1 \rightarrow (q_1, 1)$ means that in state q_0 with memory contents 0 the function ϕ_1 can be applied giving a next state q_1 and new memory value 1.

We see that, for example, ϕ_1 distinguishes between q_0 and q_1.
A characterisation set for this machine is $\mathbf{W} = \{\phi_1, \phi_2\}$.
A transition cover is :

$\mathbf{T} = \{ \Lambda, \phi_1, \phi_2, \phi_3, \phi_4, \phi_1\phi_1, \phi_1\phi_2, \phi_1\phi_3, \phi_1\phi_4, \phi_1\phi_3\phi_3, \phi_1\phi_3\phi_4, \phi_1\phi_3\phi_1, \phi_1\phi_3\phi_2 \}$

but this can be broken down into more manageable pieces so that for this stream X-machine a transition cover $\mathbf{T} = \mathbf{T_0}$ can be written as follows:

$\mathbf{T_0} = \{1\} \cup \{\phi_1\} \cup \{\phi_2, \phi_3, \phi_4\} \cup \{\phi_1\}\mathbf{T_1}$, where
$\mathbf{T_1} = \{1\} \cup \{\phi_2, \phi_3\} \cup \{\phi_1, \phi_4\} \cup \{\phi_3\}\mathbf{T_2}$ and
$\mathbf{T_2} = \{1\} \cup \{\phi_3, \phi_4\} \cup \{\phi_1, \phi_2\}$

If $m_o = 0$ is the initial memory value and $m_1 = m_2 = 1$, then:
t_o is a test function w.r.t. q_o and m_o with $t_o(\phi_1) = y$;
t_1 is a test function w.r.t. q_1 and m_1 with $t_1(\phi_3) = y$;
t_2 is a test function w.r.t. q_2 and m_2.
Then, a test set $\mathbf{Y_0} = t_o(\mathbf{X_0})$, $\mathbf{X_0} = \mathbf{T_0}\,\mathbf{Z}$, can be written as:
$t_o(\mathbf{X_0}) = t_o(\mathbf{Z}) \cup t_o(\{\phi_1\}\mathbf{Z}) \cup t_o(\{\phi_2, \phi_3, \phi_4\}) \cup \{t_o(\phi_1)\}\, t_1(\mathbf{X_1})$, where
$t_1(\mathbf{X_1}) = t_1(\mathbf{Z}) \cup t_1(\{\phi_2, \phi_3\}\mathbf{Z}) \cup t_1(\{\phi_1, \phi_4\}) \cup \{t_1(\phi_3)\}\, t_2(\mathbf{X_2})$ and
$t_2(\mathbf{X_2}) = t_2(\mathbf{Z}) \cup t_2(\{\phi_3, \phi_4\}\mathbf{Z}) \cup t_2(\{\phi_1, \phi_2\})$.

For $\mathbf{W} = \{\phi_1, \phi_2\}$ and m = n, we have $\mathbf{Z} = \{\phi_1, \phi_2\}$. Hence, by choosing appropriate values for the test functions, the test set $\mathbf{Y_0}$ is:
$\mathbf{Y_0} = \{y, x\} \cup \{yy, yx\} \cup \{x, y, x\} \cup \{y\}\mathbf{Y_1}$, where
$\mathbf{Y_1} = \{y, x\} \cup \{xy, xx, yy, yx\} \cup \{y, x\} \cup \{y\}\mathbf{Y_2}$ and
$\mathbf{Y_2} = \{y, x\} \cup \{yy, yx, xy, xx\} \cup \{y, x\}$
The full set of test inputs is thus:

$\mathbf{Y_0} = \{y, x, yy, yx, xy, xx, yyy, yyx, yxy, yxx, yyyy, yyyx, yyxy, yyxx\}$.

Alternatively we can construct a fundamental test function, *t*, directly, which satisfies the following values $t(\phi_1) = y$, $t(\phi_1\phi_2) = yx$, $t(\phi_1\phi_2\phi_4) = yxx$, $t(\phi_1\phi_2\phi_4\phi_1) = yxx$ etc. and proceed from there.