COMPLETE FUNCTIONAL TESTING OF SAFETY CRITICAL SYSTEMS

Mike Holcombe, Florentin Ipate and Andreas Grondoudis

Department of Computer Science, University of Sheffield, Sheffield, S1 4DP, UK

Abstract. No existing methods of testing allow us to make any statement about the type or precise number of faults that remain undetected after testing is completed. In particular we are unable to state that specific components of the system are free from fault after testing has been concluded. We demonstrate that a new method for generating test cases allows us to make sensible claims about the level and type of faults remaining after the testing process is complete. The method is illustrated by a case study. **Acknowledgement**. The case study was carried out in collaboration with Daimler-Benz AG., Research and Technology, Berlin.

Keywords: Verification, testing, safety-critical systems, finite state machines.

1. INTRODUCTION

The purpose of testing is to *detect* faults and the definition of a good test (Myers 1979) is one that *uncovers* faults. For safety-critical systems, however, a test that uncovers a single serious or critical fault might well be considered more successful than a test involving the same amount of effort that uncovered a number of trivial faults. We wish, then, to construct a set of test cases that will thoroughly expose the system to establish that it is fault-free, in other words to conclude that it *behaves precisely as intended*. The 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, 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, the verification process is not guaranteed to be fault-free, itself since it involves a combination of fallible human activity with tools of unknown reliability. Also, most safety-critical systems involve hybrid situations and few verification methods can deal adequately with these, see Duan et al. (1995), however. A final problem is the intricate interplay that exists between a high level program, its compiled object code and the hardware platforms and communication links. There is a case for hoping that building systems from dependable components will overcome some of these problems and we claim that testing and formal verification methods can be used *together* to ensure correctness and safety.

2. A REDUCTIONIST APPROACH

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. 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. We wish to be able to make the following statement:

"the system S is composed of the parts $P_1,..., P_n$; as a result of carrying out a testing process on **S** we can deduce that **S** is fault-free *if* each of $P_1,..., P_n$ 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 components P_i. Ultimately we reduce it to a point where the P_i are *tried and trusted* (verified or well tested).

3. COMPUTATIONAL APPROACH TO TESTING.

The only assumption we will make is that whatever implementation we produce can be considered to behave like a Turing machine. We will also assume that the specification of the system is also of this form, namely a Turing machine. 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 (1978), and developed by a number of other authors since, e.g.. Fujiwara et al. (1991), 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.

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 (1974) they have received little further study. Holcombe (1988) 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

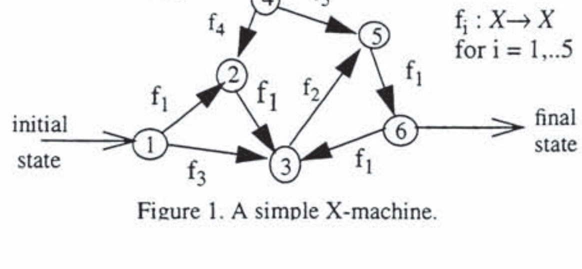

Figure 1. A simple X-machine.

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.. The machine starts in a given, initial state (control state) and a given state, x, 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. Such a sequence 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 Eilenberg (1974). 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 next. 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 $\phi : \Gamma^* \times M \times \Sigma^* \to \Gamma^* \times M \times \Sigma^*$ 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. 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 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. Let M and M' be two deterministic stream X-machines with the same set of basic functions Φ, f and f' the functions computed by them and A and A' their associated automata. If A and A' are isomorphic then $f = f'$. (Proof. See Ipate et al. 1994).

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. We need some further terminology for stream X-machines.

Definition. 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.

Definition. 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.

Definition. 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'.

Definition. Two states q and q' are said to be **V**- *distinguishable* if they are not **V**- equivalent.

Definition. 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. .

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

If A and A' are minimal, card(Q') - card$(Q) \leq$ k and $f(s) = f'(s)\ \forall\ s \in t(\mathbf{TZ})$, then the associated automata A and A' are isomorphic.

Once all of this mechanism is in place we can apply the fundamental theorem to generate a test set mechanically. Thus we construct explicitly $\mathbf{Z} = \Phi^k\mathbf{W} \cup \Phi^{k-1}\mathbf{W} \cup \ldots \cup \mathbf{W}$ 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 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 et al. (1994). 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. 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. 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.

4. CASE STUDY.

The example system that we will use to describe the test set construction is the Two Button Press system. Although this is a simple system it includes a number of essential features which are of interest in a number of safety-critical applications.

4.1. System description.

The system in question is a two-button press. It consists of a pneumatic press controlled by 2 buttons through a Programmable Logic Controller (PLC). The press operates by way of a plunger. The basic requirements for the correct operation of the system are as follows:

A normal cycle of the system starts with the plunger at the top or idle position; when both buttons are pressed and held down the plunger will descend and carry out the press operation on the sample material; the plunger then returns to the idle position.. The system includes two major safety points. The first one is the fact that the plunger will not start descending unless both buttons are pressed and remain pressed. The second is the point of no return (PNR) allowing the operator the capability of suspending a *cycle* once it has been initiated before the plunger has past the PNR. To allow for practical, manual operation the buttons do not have to be pressed at the same instant but can be pressed within 0.5 seconds of each other - and then held down in order for the plunger to continue descending. In the case where a button has been pressed for a time exceeding the 0.5 seconds interval, and the other button has not been pressed in that time, then the pressed button has to be released before a new *cycle* can be initiated. As long as the two buttons are kept pressed, the plunger will continue to descend. In the instance of one or both buttons being released then:

- If the plunger has not yet passed the PNR, then the *cycle* is suspended and the plunger will stop, and return to the open position (*conditional closing*).
- If the plunger has passed the PNR, then the *cycle* cannot be suspended (*unconditional closing*). The plunger will continue descending until it reaches the closed position, it will then close and after the predefined closing time has elapsed, it will then start to ascend on its way to the open position and the completion of the *cycle*. A number of indicators will be included in the design to assist in monitoring the status of the system, these will be reflected in the output sets for the system. Thus the panel of the machine will include some LEDs, some bells, and one more switch, in addition to the left and right buttons (used for input commands).

4.2. Modelling conventions.

We assume that whenever the power of the machine is cut-off, then the power switch of the machine automatically turns to the off setting. It is also assumed that whenever power is cut-off or switched-off, then, the power will not be regained at least for a period of time greater or equal to the time it will take the plunger to be retrieved to the open position (including stopping time and retrieval time). This model uses the one/both button approach. This means that the system is modelled as a combination of either: no button(s), one button, or both buttons being pressed or released at any one time. The input is not left or right press/or release but one or both.

4.3. The fundamental data type X.

The fundamental data type of the SXM is given in the form $X = \Gamma^* \times M \times \Sigma^*$, where Γ^* is the set of possible outputs including the empty element, and Σ^* is the set of possible inputs of the machine also including the empty element, and M is the memory set.

Σ : POWER x SAMPLE x BUTTONS x TIME

where:

POWER = { p_on , p_loss , p_off }

SAMPLE= N

BUTTONS= { both_press, both_release, one_press,

one_release, switch_press}
TIME = N,
The set Σ describes the possible inputs to the system. It consists of a record type comprising four basic sets:
POWER signifies the possible inputs that exist in connection with the power of the system.
SAMPLE is a value that will signify where the plunger is in respect of the open position.
BUTTONS are the possible inputs that the operator can present the machine with by pressing/releasing respective buttons on the panel.
TIME represents a system clock with a regular tick which increments by one the clock reading.

M:=TIME x POSITION x BUTTON x WAIT x MOVE x POWER
TIME = N,
POSITION = N
BUTTON = {one_press, both_press}
WAIT = {both_press, both_release}
MOVE = { up, down, stop, retrieve}
POWER = { power, no_power }
M is the internal memory of the machine.
POSITION is a number that indicates where the plunger is at a given time, the value of POSITION is 0 at the idle position.
BUTTON is the element that will hold information concerning how many buttons are being pressed at any time.
WAIT This element denotes what the system is waiting for with respect to button pressing or releasing.
MOVE signifies the action/move that the plunger is performing at any instance in time.
POWER denotes the existence or lack of power in the machine.

Γ:= LIGHTS x RINGS
LIGHTS= STATUS_L x BUTTON_L
where
STATUS_L = {open, opening, closed, closing, stop, stopping,point_no_return}
BUTTON_L={one_release,both_release,both_pressed one_pressed, none_pressed, locked }
RINGS={power_loss_ring,interruption_ring,default_retrieval_ring}
Γ is the output set which will let the operator know what is happening.

4.4. The transition diagram.

The machine's transition diagram, Figure 2, has been abbreviated slightly, it does not describe the fail safe operational sequences. There a number of annotations to the states (e.g.. +, #) which indicate the existence of fail safe paths which describe the operation during power loss. Lack of space prevents us from discussing these in detail. The symbol $ indicates a loop which models the passage of time, i.e.. $ is a "tick" function that simply increments the system clock. Only one main function is shown, [0;1].

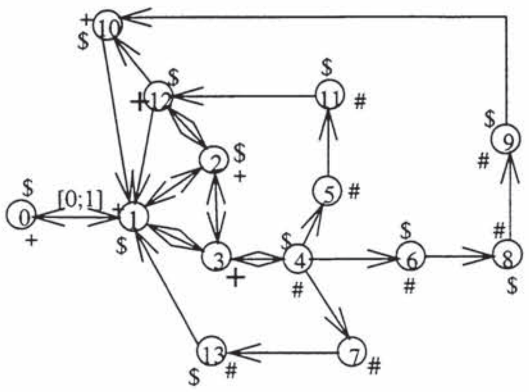

Figure 2. The transition diagram.

4.5. The basic functions of the machine.
Each function has a label of the form [i;j] to indicate that it is the function from state i to state j. The exceptions are the function "tick" which is labelled by $ and the fail safe recovery functions ν, +, #. Functions operate at clock ticks. The format of the function definitions is of the form:

[i;j] : ([memory value], [input value]) $\rightarrow$
([output value], [memory value])

Thus
[0;1] : ([-, -, -, -, -, no_power] , [p_on, -, -, -]) $\rightarrow$
([(open, none_pressed), -], [1, 0, -, -, -, power])

To describe the transition in words; in State 0 the machine is off; there is no output; the memory set is empty apart from the power element which indicates that there is no power in the system; the input indicates the action that switches on the power. The basic function [0;1] defines the new memory state which now has a clock time reading and a flag set corresponding to the fact that the system is switched on. The output indicates that the plunger is open and no buttons have been pressed. So the system is in state 1 (on and idle). The dashes "-" indicate that no values are read or written (changed) in these locations. The last clause means that in all other cases the operation of the function [0;1] preserves the output display except for advancing the clock by 1. So if someone pushes the start button at any other stage of operation it is ignored.

Several further functions are now possible.
[1;2] : ([n, 0, -, -, -, power] , [-, 0, one_press, -]) $\rightarrow$
[(open, one_pressed), -] , [n+1 , 0, one_press, both_press, -, power])
$: ([n, -, -, -, -, -] , [-, -, -, 1]) $\rightarrow$
([(-, -) , -] , [n+1, -, -, -, -, -])
and so on.

4.6. The test set for the machine.
The type Φ of the machine is the following set (some of the basic functions listed are involved in the "fail

safe" part of the system involving states not described here).

$\Phi = \{$ v, \$, +, #, [0;1], [1;0], [1;2], [1;3], [2;1], [2;3], [2;12], [3;1], [3;2], [3;4], [4;5], [4;6], [4;7], [5;11], [6;8], [7;13], [8;9], [9;10], [10;1], [10;12], [11;12], [12;1], [12;2], [12;10], [13;1]$\}$

A question that we ought to consider, now, is whether the "design for test" conditions are met. This example can easily be made to satisfy them although the proof is not given here.

The characterisation set is:

W = {[0;1], [1;2], [2;12], [3;4], [4;5], [5;11], [6;8], [7;13], [8;9], [9;10], [10;1], [11;12], [12;1], [13;1], [14;15] }

For simplicity the test set generation strategy assumes that k is equal to zero, thus defining Z to be equal to W.

The following are the state transition covers for the first two states of the machine. The complete listing of the transition cover can be obtained routinely.

$T=\{\Lambda, \$, [0;1]\} \cup (\Phi - \{\$, [0;1]\}) \cup \{[0;1]\}T_1$

$T_1=\{[1;0], \$, [1;2], [1;3]\} \cup (\Phi - \{[1;0], \$, [1;2], [1;3]\}) \cup \{[1;2]\}T_2 \cup \{[1;3]\}T_3$

There is one test function for each state of the system and it should have as its arguments all possible sequences of transitions. Complete listings of the test function for all the states of the machine can be found routinely The test set can be constructed by following the procedure explained in section 3 and the Appendix. All of the sets and functions required have now been computed. The construction is straightforward.

5. DISCUSSION.

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. Holcombe (1993) 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, 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. The development of the detailed SXM specification for the two button press example was quite straightforward as was the generation of the test sets.

Complexity analysis on the test sets (see Ipate (1995)) shows that they are an order of magnitude smaller than the finite state machine equivalent.

Clearly it is necessary to support these activities with suitable tools and this is now a matter for examination.

It would be interesting to see whether a similar type of testing could be found for hybrid systems.

REFERENCES.

Myers, G. (1979). *The art of software testing*. Wiley, New York.

Duan, Z., Holcombe, M., Linkens, D. (1995). Modelling a soaking pit using hybrid machines. *Systems Analysis, Modelling, and Simulation*. 18, 153-157.

Chow, T.S. (1978). Testing software design modeled by finite state machines. *IEEE Transactions on Software Engineering*, **4**(3), 178-187.

Fujiwara, S., von Bochmann, G., Khendek, F., Amalou, M. and Ghedamsi, A. (1991). Test selection based on finite state models. *IEEE Trans. on Software Engineering*, **17**(6), 591-603.

Eilenberg, S. (1974). *Automata, languages and machines, Vol. A.*, Academic Press.

Holcombe, M. (1989). X-machines as a basis for dynamic system specification. *Software Engineering Journal*, **3**(2), 69-76.

Holcombe, M. (1993). An Integrated Methodology for the Specification, Verification and Testing of Systems. *Software Testing, Verification and Reliability*, **3**, 149-163.

Ipate, F. and Holcombe, M. (1994). X-machine based testing. Departmental Report, Department of Computer Science, University of Sheffield.

Ipate. F. (1995) *X-machines - theory and applications in specification and testing*. PhD thesis, University of Sheffield.

APPENDIX. 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 $t_{q,m} : \Phi^* \to \Sigma^*$ 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 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 $m = m_o$, then $t_{q,m}$ is denoted by *t* and is called a *fundamental test function of M*.