

An integration testing method that is proved to find all faults

Florentin Ipate & Mike Holcombe

Formal Methods and Software Engineering (FORMSOFT) Group

Department of Computer Science

University of Sheffield,

Regent Court, 211, Portobello Street

Sheffield S1 4DP, UK.

Correspondence to:

Professor Mike Holcombe

Department of Computer Science

University of Sheffield,

Regent Court, 211, Portobello Street

Sheffield S1 4DP, UK.

Email : M.Holcombe@dcs.shef.ac.uk

Abstract. Although a great deal of research has been done in the area of formal methods and their practical use for the specification and verification of software systems, testing issues are very seldom mentioned by those within the formal methods community. Almost all the methods currently used for testing software are experience based rather than theoretically founded methods. In particular, very few methods allow us to make any statement about the type or number of faults that remain undetected after testing is completed. This paper describes a method for using the existence of a formal specification as the basis for the development of a detailed functional testing strategy. By considering testing from a theoretical point of view we demonstrate that this method can provide a more convincing approach to the problem of detecting *all* faults. The formal method used, X-machines, is a blend of finite state machines, data structures and processing functions and provides a simple and intuitive way for specifying computer systems.

Key words: Testing, test set, formal specification, finite state machines, X-machines.

1. Introduction

Two current areas of emphasis in the development of higher quality software is the use of formal methods for the specification and verification of software and the development of sophisticated methods of software and system testing. In general, these areas have been regarded as mutual exclusive and testing issues are very seldom mentioned by those within formal methods community. Indeed, the belief of 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 to system correctness. However, formal verification has not been able to demonstrate yet that it is "scalable" and practical in the context of the massive complexity of today's applications.

On the other hand, although a number of techniques for carrying out testing, and in particular for the generation of test sets exist and many sophisticated (and expensive) tools are available on the market, very few of the current testing methods are based on a firm scientific approach or theory, they tend to be experience based methods that have evolved, informal 'ad hoc' approaches rather than carefully constructed engineering methods based on a defensible well-founded strategy. These techniques are aimed at finding faults, but in most cases they fail to give any guarantee that the system is fault-free after testing has been completed. This is clearly a major drawback and justifies to some extent the claim that "all a test can tell us is that a system has failed - it cannot tell us that a system is correct" (statement attributed to Dijkstra).

The claim "the system is fault-free after testing has been completed" can only be made if the (finite) test set generated by the method is proved to reveal *all* the faults of the implementation. Therefore, if the specification S and the implementation I are assumed to be (partial) functions $S, I: D \rightarrow R$ (i.e. where D is the input domain and R the result domain) and $X \subseteq D$ is the (finite) test

set, then the implication " $S(x) = I(x) \forall x \in X \Rightarrow S(x) = I(x) \forall x \in D$ " must be shown to be correct.

The way in which we shall address this problem will be to consider testing based on an algebraic approach to computational modelling. Obviously, when we are constructing a software (or hardware) system, we are attempting to construct something that will carry out some computable function. Therefore, both the specification and the implementation are computable functions, hence they can be both represented as some computational models or machines. A testing method would then try to ascertain if these two machines compute the same function. However, if S and I are assumed to be arbitrary Turing machines, this is clearly impossible.

One way to get around this problem is to look at more restrictive models (e.g. finite state machines). Some finite testing methods have been developed first by Chow [2], and Fujiwara et al. [5]. But the finite state machine is too restrictive for many common applications. The solution to this lack of generality suggested by Chow, was to separate the control structure of a program from the data structure and to represent the former as a finite state machine. In this way the method could be used to test the control structure of a program. However, the assumption that the control structure of the system can be modelled separately from the data variables is not realistic in many cases. This would mean that the next state depends solely on the current state and the input. This is not usually the case. The variables that affect the program control could be replaced by a number of additional states, but in many cases this number will be large and then the method would become impractical.

So, it appears that what we need is a test method based on a model much more general than the finite state machine. Such a model will also have a control structure, represented by a finite number of states and a state transition diagram, but also a (possibly infinite) data or memory structure. Furthermore, the model has to be based at a sufficiently high level of abstraction (i.e.

unlike, for example models such as pushdown automata and Turing Machines) to make it a convenient tool for specifying real systems. Such a model is Eilenberg's X-machine.

2. X-machines.

Introduced by Eilenberg [3] in 1974, the X-machines have received little further study. Holcombe [6] proposed the model as a basis for a possible specification language and since then a number of further investigations have demonstrated that this idea is of great potential value for software engineers (see Fairtlough et al. [4], Chiu [1] and Laycock [9]). 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 finite set of basic processing (partial) functions, Φ , which operate on X . Φ is called the *type* of the machine and represents the elementary operations that the machine is capable of performing. The set Φ is finite since the machine has a finite number of edges. 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.

This method allows the control state of the system to be easily separated from the data set, 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.

Consider a simple, abstract example (see figure 1):

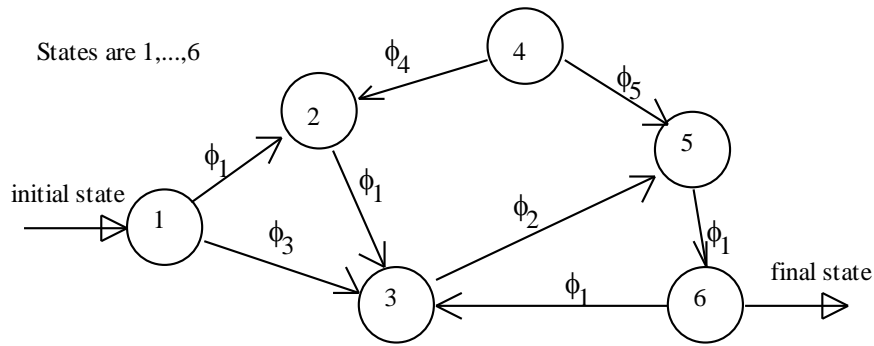


Figure 1. The state transition diagram of an X-machine.

We haven't specified X or the functions ϕ_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 ϕ_1, ϕ_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. Usually, 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 $\phi_1, \phi_1, \phi_2, \phi_1$. Assuming that each value is defined this path then transforms an initial value $x \in X$ into the value $\phi_1(\phi_2(\phi_1(\phi_1(x)))) \in X$. So we are assuming that $x \in \text{dom } \phi_1; \phi_1(x) \in \text{dom } \phi_1; \phi_1(\phi_1(x)) \in \text{dom } \phi_2; \text{ and } \phi_2(\phi_1(\phi_1(x))) \in \text{dom } \phi_1$. 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 [3]. In fact it is slightly too general and we now consider a natural subclass of these machines.

3. Stream X-machines.

Those X-machines in which the input and the output sets are 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

$$\phi: \Gamma^* \times M \times \Sigma^* \rightarrow \Gamma^* \times M \times \Sigma^*$$

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.

Definition 3.1.

Let Σ and Γ two finite alphabets (called the *input* and *output alphabet* respectively). Then, a *stream X-machine* is a tuple $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0)$, where:

1. Q is the (finite) *set of states*.
2. M is a (possibly infinite) set called *memory*.
3. Φ is the *type* of \mathcal{M} , a finite set of partial functions

$$\phi: \Gamma^* \times M \times \Sigma^* \rightarrow \Gamma^* \times M \times \Sigma^*$$

of the form

$$\phi(g, m, s) = \begin{cases} (g \rho(m, \text{head}(s)), \mu(m, \text{head}(s)), \text{tail}(s)), & \text{if } s \neq 1 \\ \emptyset, & \text{otherwise} \end{cases}$$

where $\mu: M \times \Sigma \rightarrow M$, $\rho: M \times \Sigma \rightarrow \Gamma$ are partial functions.

[**Note:** 1 is the empty string. For two strings $s, t \in \Sigma^*$, st represents the string obtained by concatenation. $\text{head}: \Sigma^* \rightarrow \Sigma^*$ and $\text{tail}: \Sigma^* \rightarrow \Sigma^*$ are partial functions defined by:

$$\text{head}(\sigma s) = \sigma, \forall \sigma \in \Sigma, s \in \Sigma^*; \text{head}(1) = 1;$$

$$\text{tail}(\sigma s) = s, \forall \sigma \in \Sigma, s \in \Sigma^*; \text{tail}(1) = 1].$$

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.

4. F is the '*next state*' partial function.

$$F: Q \times \Phi \rightarrow \mathcal{P}Q$$

F is often described by means of a *state-transition diagram*.

5. I and T are the sets of *initial* and *terminal states* respectively.

$$I \subseteq Q, T \subseteq Q.$$

6. m_0 is the *initially memory value*.

As for finite state machines, if $q, q' \in Q$, $\phi \in \Phi$ and $q' \in F(q, \phi)$, then $q \xrightarrow{\phi} q'$ is called the *arc* from q to q' . A sequence of arcs $q \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} q_2 \dots q_n \xrightarrow{\phi_{n+1}} q'$ is called a *path*. A *successful path* is one that starts in an initial state and ends in a final one. For a path p , $|p|$ denotes the (partial) function computed along that path.

Definition 3.2.

The *behaviour* of \mathcal{M} is the relation $|\mathcal{M}|: \Gamma^* \times M \times \Sigma^* \leftrightarrow \Gamma^* \times M \times \Sigma^*$ defined as

$$|\mathcal{M}| = \cup |p|,$$

with the union extending over all the successful paths p in \mathcal{M} .

Definition 3.3.

Let \mathcal{M} be a stream X-machine and $\alpha: \Sigma^* \rightarrow \Gamma^* \times M \times \Sigma^*$, $\beta: \Gamma^* \times M \times \Sigma^* \rightarrow \Gamma^*$ defined by :

$$\alpha(s) = (1, m_0, s) \quad \forall s \in \Sigma^*.$$

$$\beta(g, m, s) = \begin{cases} g, & \text{if } s = 1 \\ \emptyset, & \text{otherwise} \end{cases}$$

Then the composite relation

$$f = \Sigma^* \xrightarrow{\alpha} \Sigma^* \times M \times \Gamma^* \xleftarrow{\mathcal{M}} \Sigma^* \times M \times \Gamma^* \xrightarrow{\beta} \Gamma^*$$

is called the *relation computed* by \mathcal{M} .

In other words, for an input sequence s , $f(s)$ will represent the set of outputs produced by the machine when s takes the machine from an initial state and the initial memory value to a terminal state.

Now, it is clear that any processing function ϕ of a stream X-machine is completely determined by ρ and μ (see definition 3.1). In order to simplify the notation, in what follows such a function will be referred to as a pair $\phi = (\rho, \mu)$. That is, instead of saying

$$\phi(g, m, \sigma s) = (g\gamma, m', s),$$

with $m, m' \in M$, $\gamma \in \Gamma$, $\sigma \in \Sigma$, $\forall g \in \Gamma^*$, $s \in \Sigma^*$ we say

$$\phi(m, \sigma) = (\gamma, m'), \text{ the rest being understood implicitly.}$$

Then the type Φ of a stream X-machine will be referred to as a set of partial functions

$$\phi: M \times \Sigma \rightarrow \Gamma \times M$$

A deterministic stream X-machine is one in which there is at most one possible transition for any state triplet $q \in Q, m \in M, \sigma \in \Sigma$. This will almost always be the case in practical applications.

Definition 3.4.

A X-machine \mathcal{M} is called *deterministic* if:

1. F maps each pair $(q, \phi) \in Q \times \Phi$ onto at most a single next state:

$$F: Q \times \Phi \rightarrow Q$$

2. I contains only one element (i.e. $I = \{q_0\}$, where $q_0 \in Q$).

3. If $q \xrightarrow{\phi} p$ and $q \xrightarrow{\phi'} p'$ are distinct arcs emerging from the same state q , then

$$\text{dom } \phi \cap \text{dom } \phi' = \emptyset.$$

Clearly, a deterministic stream X-machine computes a (partial) function $f: \Sigma^* \rightarrow \Gamma^*$ rather than a relation.

Example 3.5. *A deterministic stream X-machine*

1. $\Sigma = \{x, y\}$

2. $\Gamma = \{a, b\}$

3. $Q = \{q_0, q_1, q_2\}$; q_0 is the initial state and all the states are terminal ($T = Q$).

4. $M = \{0, 1\}$. The initial memory value is $m_0 = 0$.

5. $\Phi = \{\phi_1, \phi_2, \phi_3, \phi_4\}$, where

$$\phi_1: M \times \{y\} \rightarrow \Gamma \times M \text{ is defined by } \phi_1(m, y) = (a, 1), m \in M;$$

$$\phi_2: M \times \{x\} \rightarrow \Gamma \times M \text{ is defined by } \phi_2(m, x) = (a, 0), m \in M;$$

$$\phi_3: M \times \{y\} \rightarrow \Gamma \times M \text{ is defined by } \phi_3(m, y) = (b, 1), m \in M;$$

$$\phi_4: M \times \{x\} \rightarrow \Gamma \times M \text{ is defined by } \phi_4(m, x) = (b, 0), m \in M.$$

6. F is represented in the figure 2.

For example, for the input sequence $xyxy$ will have $f(xyxy) = aabb$ and for xy , $f(xy)$ will be undefined .

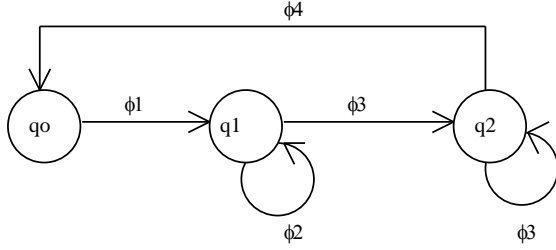


Figure 2.

Before we proceed any further, we introduce some preparatory concepts and notations.

Definition 3.6.

Given a deterministic stream X-machine \mathcal{M} , we define the partial functions

$$v: Q \times M \times \Sigma \rightarrow Q, w: Q \times M \times \Sigma \rightarrow M, \lambda: Q \times M \times \Sigma \rightarrow \Gamma,$$

with

$$\text{dom } v = \text{dom } w = \text{dom } \lambda,$$

such that :

1. $(q, m, \sigma) \in \text{dom } v$ iff $\exists \phi \in \Phi$ such that $(q, \phi) \in \text{dom } F$ and $(m, \sigma) \in \text{dom } \phi$.
2. if $F(q, \phi) = q', \phi(m, \sigma) = (\gamma, m')$, then

$$v(q, m, \sigma) = q', w(q, m, \sigma) = m' \text{ and } \lambda(q, m, \sigma) = \gamma,$$

where ϕ is the one above.

Obviously, since \mathcal{M} is deterministic, such ϕ is unique, hence v, w and λ are well defined. $v(q, m, \sigma)$ and $w(q, m, \sigma)$ indicate the next state and the next memory value respectively produced by the machine when σ is received in q and m ; $\lambda(q, m, \sigma)$ indicates the output symbol produced by the machine.

3. We also define

$$u: Q \times M \times \Sigma \rightarrow Q \times M,$$

by

$$u(q, m, \sigma) = (v(q, m, \sigma), w(q, m, \sigma)) \quad \forall q \in Q, m \in M, \sigma \in \Sigma$$

(i.e. u indicates both the next state and next memory).

Then u is called the *transition function* and λ the *output function*.

Definition 3.7.

We extend u and λ to

$$u_e: Q \times M \times \Sigma^* \rightarrow Q \times M \text{ and } \lambda_e: Q \times M \times \Sigma^* \rightarrow \Gamma^*,$$

where u_e , and λ_e are defined recursively by:

$$u_e(q, m, 1) = (q, m), \text{ where } 1 \text{ is the empty string}$$

$$u_e(q, m, s\sigma) = \begin{cases} u(u_e(q, m, s), \sigma) & \forall \sigma \in \Sigma, s \in \Sigma^* \text{ such that} \\ & u_e(q, m, s) \neq \emptyset \text{ and } u(u_e(q, m, s), \sigma) \neq \emptyset \\ \emptyset, & \text{otherwise} \end{cases}$$

$$\lambda_e(q, m, 1) = 1$$

$$\lambda_e(q, m, s\sigma) = \begin{cases} \lambda_e(q, m, s) \lambda(u_e(q, m, s), \sigma) & \forall \sigma \in \Sigma, s \in \Sigma^* \text{ such that} \\ & \lambda_e(q, m, s) \neq \emptyset \text{ and } \lambda(u_e(q, m, s), \sigma) \neq \emptyset \\ \emptyset, & \text{otherwise} \end{cases}$$

Then u_e is called the *extended transition function* and λ_e the *extended output function*. It is clear that $\text{dom } u_e = \text{dom } \lambda_e$. We also define two partial functions

$$v_e: Q \times M \times \Sigma^* \rightarrow Q \text{ and } w_e: Q \times M \times \Sigma^* \rightarrow M$$

by:

$$\text{dom } v_e = \text{dom } w_e = \text{dom } u_e$$

and

$$v_e(q, m, s) = q' \text{ and } w_e(q, m, s) = m',$$

where $(q', m') = u_e(q, m, s) \forall (q, m, s) \in \text{dom } u_e$.

It is obvious that v_e and w_e are extensions of v and w respectively.

If a machine \mathcal{M} is in the state q with the memory value m , then an input sequence s takes the machine to the state $q' = v_e(q, m, s)$ and the memory value $m' = w_e(q, m, s)$, while adding the sequence $\lambda_e(q, m, s)$ to the output string.

Lemma 3.8.

Let \mathcal{M} be a deterministic stream X-machine, $(q, m) \in Q \times M$ and $s, s' \in \Sigma^*$. Then:

$$\lambda_e(q, m, ss') = \lambda_e(q, m, s) \lambda_e(u_e(q, m, s), s'),$$

$$u_e(q, m, ss') = u_e(u_e(q, m, s), s').$$

Proof:

It follows by induction on the length of s' . ⑥

We can now express the (partial) function computed by a deterministic stream X-machine in terms of the extended output and transition functions.

Lemma 3.9.

Let \mathcal{M} be a deterministic stream X-machine. Then we say that \mathcal{M} computes a (partial) function $f: \Sigma$

$^* \rightarrow \Gamma^*$ defined by:

$$\begin{cases} \lambda_e(q_0, m_0, s), & \text{if } v_e(q_0, m_0, s) \in T \end{cases}$$

$$f(s) = \begin{cases} \lambda_e(q_0, m_0, s) & \text{if } v_e(q_0, m_0, s) \in T \\ \emptyset, & \text{otherwise} \end{cases}$$

Proof:

Let $s \in \Sigma^*$ and $x_0 = (1, m_0, s)$. Let also $x \in \Gamma^* \times M \times \Sigma^*$ be the final value computed by the machine following the (unique) path (if any) emerging from initial state q_0 having x_0 as the initial value. Then

$$x = (\lambda_e(q_0, m_0, s), w_e(q_0, m_0, s), 1).$$

Hence

$$\beta(x) = \lambda_e(q_0, m_0, s).$$

Now, the result above follows since $f(s) = \beta(x)$ if $v_e(q_0, m_0, s) \in T$ and undefined otherwise. ©

In what follows we shall consider that all the states of the machine are terminal. Therefore all the outputs produced by the machine will be considered, even though the machine is not in a state where its computation can be considered terminated. In practical terms this means that, for testing purposes, the intermediary results produced by the system will be considered, as well as the final ones. In what follows we shall be referring to deterministic stream X-machines with all the states terminal (thus the set of final states will not be mentioned, the implicit assumption being that $T = Q$). Obviously, in this case, the (partial) function computed by the machine will satisfy

$$f(s) = \lambda_e(q_0, m_0, s) \quad \forall s \in \Sigma^*.$$

4. Stream X-machine testing.

Let us get back to testing and see how test sets that find all faults in the implementation can be derived from a stream X-machine specification. Unlike the finite state machine case, a testing method for stream X-machines is not straightforward. Indeed, since the machine memory can be

infinite, it is fairly clear that there is no way of finding a finite set of input sequences that would guarantee that two arbitrary stream X-machines (one representing the specification, the other the implementation) compute the same function.

The approach we shall use to get around this problem will be a reductionist one. This entails the reduction of a problem to the solution of simpler ones. In such a reductionist approach we would consider a system and produce a testing regime that results 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, this approach will work only if we are able to make the following statement:

"the system S is composed of the parts P_1, \dots, 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, \dots, P_n are fault-free".

If the system is a stream X-machine \mathcal{M} , then the basic components or parts of the system are the ϕ 's. Then, what we are looking for is a testing method that ensures that \mathcal{M} is fault-free provided that Φ is fault-free.

4.1. Theoretical basis for stream X-machine testing.

Given an X-machine $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$, we can convert it into a finite state machine $\mathcal{A} = (\Phi, Q, F, q_0)$ 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 . We call this *the associated automaton* of \mathcal{M} . If \mathcal{M} and \mathcal{M}' are two deterministic stream X-machines with the same type Φ and their associated automata \mathcal{A} and \mathcal{A}' accept the same language $L \subseteq \Phi^*$, then it is fairly clear that \mathcal{M} and \mathcal{M}' compute the same function (see also Ipaté [8]).

The strategy we employ is to reduce testing that the two machines (i.e. one representing the specification, the other the implementation) compute the same function to testing that their associated automata accept the same language. For this idea to work, we require that the type Φ of the two machines has to satisfy two conditions, completeness and output-distinguishability, as defined in what follows.

Definition 4.1.1.

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 $\phi_2(m, \sigma) = (\gamma, m_2')$ for some $m_1', m_2' \in M, \gamma \in \Gamma$, then $\phi_1 = \phi_2$.

What this is saying is that we must be able to distinguish between any two different processing functions (the ϕ 's) by examining outputs. If we cannot then we will not be able to tell them apart.

Definition 4.1.2.

A Φ is called *complete* if:

$\forall \phi \in \Phi$ and $\forall m \in M, \exists \sigma \in \Sigma$ such that $(m, \sigma) \in \text{dom } \phi$.

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 input symbols and augmenting the output alphabet (see Ipate [8]). The extended versions of the ϕ 's will only be used in testing and the extra features can be removed after testing has been completed.

Now we consider how test sets can be constructed that will show that the two associated automata accept the same language. The following definitions and concepts refer to finite state machines and are largely from Chow [2].

Definition 4.1.3.

Let $Y \subseteq \Phi^*$ be a set of input sequences and let q and q' two states in \mathcal{A} and \mathcal{A}' respectively. Then, q and q' are said to be *Y-equivalent* if $\forall y \in Y$ an input sequence then \mathcal{A}_q accepts y iff \mathcal{A}'_q accepts y , where $\mathcal{A}_q = (\Phi, Q, F, q)$ and $\mathcal{A}'_q = (\Phi', Q', F', q')$ are the finite state machines obtained from \mathcal{A} and \mathcal{A}' respectively by considering q and q' as initial states.

Thus q and q' are said to be *Y-equivalent* if whenever we can find a path labelled by $y \in Y$ from q we can find a path labelled by y from q' and vice versa.

Definition 4.1.4.

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

Thus some y exists in Y such that either:

- a path labelled y exists from q and no path labelled y exists from q' ;
- or a path labelled y exists from q' and no path labelled y exists from q .

Definition 4.1.5.

Let $\mathcal{A} = (\Phi, Q, F, q_0)$ be a minimal finite state machine. Then a set of input sequences $W \subseteq \Phi^*$ is called a *characterisation set* of \mathcal{A} if any two different states in \mathcal{A} are *W-distinguishable*.

Definition 4.1.6.

Let $\mathcal{A} = (\Phi, Q, F, q_0)$ be a finite state machine. Then a set of input sequences $T \subseteq \Phi^*$ is called a *transition cover* if for any state $q \in Q$, there is an input sequence $y \in \Phi^*$ which forces the machine \mathcal{A} into q from the initial state q_0 such that $y \in T$ and $y\phi \in T, \forall \phi \in \Phi$.

For the associated automaton of the X-machine presented in example 3.5,

$W = \{\phi_1, \phi_2\}$ is a characterisation set and

$T = \{1, \phi_1, \phi_2, \phi_3, \phi_4, \phi_1\phi_1, \phi_1\phi_2, \phi_1\phi_3, \phi_1\phi_4, \phi_1\phi_3\phi_1, \phi_1\phi_3\phi_2, \phi_1\phi_3\phi_3, \phi_1\phi_3\phi_4\}$ is a transition cover.

Now, using Chow's method [2] we can construct a set of sequences of elements from Φ^* that will establish whether the two associated automata accept the same language. 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 as discussed next.

Definition. 4.1.7.

Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ 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. $t_{q,m}(1) = 1$, where 1 is the empty string.

2. For $n \geq 0$, the recursion step that defines $t_{q,m}(\phi_1 \dots \phi_n \phi_{n+1})$ as a function of $t_{q,m}(\phi_1 \dots \phi_n)$ depends on the following two cases:

a. if \exists a path $q \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} q_2 \dots q_{n-1} \xrightarrow{\phi_n} q_n$ in \mathcal{M} starting from q , then

$$t_{q,m}(\phi_1 \dots \phi_n \phi_{n+1}) = t_{q,m}(\phi_1 \dots \phi_n) \sigma_{n+1},$$

where σ_{n+1} is chosen such that

$$(w_e(q, m, t_{q,m}(\phi_1 \dots \phi_n)), \sigma_{n+1}) \in \text{dom } \phi_{n+1}.$$

[**Note:** Since Φ is complete, there exists such σ_{n+1} .]

In other words, if m_n is the final value computed by the machine along the above path on the input sequence $t_{q,m}(\phi_1 \dots \phi_n)$, then (m_n, σ_{n+1}) will exercise ϕ_{n+1} .

b. otherwise,

$$t_{q,m}(\phi_1 \dots \phi_n \phi_{n+1}) = t_{q,m}(\phi_1 \dots \phi_n).$$

Then $t_{q,m}$ is called a *test function* of \mathcal{M} w.r.t. q and m .

If $q = q_0$ and $m = m_0$, $t_{q,m}$ is denoted by t and is called a *fundamental test function* of \mathcal{M} .

In other words if

$$q \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} q_2 \dots q_{n-1} \xrightarrow{\phi_n} q_n$$

is a path in \mathcal{M} , then

$$s = t_{q,m}(\phi_1 \dots \phi_n)$$

will be an input string which, when applied in q and m , will cause the computation of the machine to follow this path (i.e. $s = \sigma_1 \dots \sigma_n$ such that σ_1 exercises ϕ_1 , ..., σ_n exercises ϕ_n).

If there is no arc labelled ϕ_{n+1} from q_n , then

$$t_{q,m}(\phi_1 \dots \phi_n \phi_{n+1}) = s\sigma_{n+1},$$

where σ_{n+1} 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, $\forall \phi_{n+2}, \dots, \phi_{n+k} \in \Phi$,

$$t_{q,m}(\phi_1 \dots \phi_n \phi_{n+1} \dots \phi_{n+k}) = t_{q,m}(\phi_1 \dots \phi_n \phi_{n+1})$$

(i.e. therefore only the first non-existing arc in the path is exercised by the value of the test function).

Note that a test function is not uniquely determined, many different possible test functions exist and it is up to the designer to construct it.

For the X-machine presented in example 3.5, we can construct a fundamental test function which satisfies

$$t(\phi_1) = y, t(\phi_1\phi_2) = yx, t(\phi_1\phi_2\phi_4) = yxx, t(\phi_1\phi_2\phi_4\phi_1) = yxx, t(\phi_1\phi_2\phi_4\phi_1\phi_2) = yxx.$$

The scope of a test function is to test whether a certain path exists or not in \mathcal{M} using appropriate input symbols (hence the name). This idea is formalised in the following lemma.

Lemma 4.1.8.

Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ and $\mathcal{M}' = (\Sigma, \Gamma, Q', M, \Phi, F', q_0', m_0')$ be two deterministic stream X-machines with Φ output-distinguishable and complete, λ_e and λ_e' their extended output functions and \mathcal{A} and \mathcal{A}' their associated automata. Let $q \in Q, q' \in Q', m \in M, Y \subseteq \Phi^+$, and let $t_{q,m}: \Phi^* \rightarrow \Sigma^*$ be a test function of \mathcal{M} w.r.t q and m . If $\lambda_e(q, m, s) = \lambda_e'(q', m, s) \forall s \in t_{q,m}(Y)$, then q and q' are Y -equivalent as states in \mathcal{A} and \mathcal{A}' respectively.

Proof:

Let $\phi_1 \dots \phi_n \in Y$ and $s = t_{q,m}(\phi_1 \dots \phi_n)$. We prove that $\lambda_e(q, m, s) = \lambda_e'(q', m, s)$ implies:

there exists a path in \mathcal{M} starting from q labelled $\phi_1 \dots \phi_n$ iff there exists a path in \mathcal{M}' starting from q' labelled $\phi_1 \dots \phi_n$.

Let us assume that there exists a path

$$q \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} q_2 \dots q_{n-1} \xrightarrow{\phi_n} q_n$$

in \mathcal{M} . In this case

$$t_{q,m}(\phi_1 \dots \phi_n) = \sigma_1 \dots \sigma_n \text{ with } \sigma_1, \dots, \sigma_n \in \Sigma.$$

Thus there exist $\gamma_1, \gamma_2, \dots, \gamma_n \in \Gamma$ and $m_1, \dots, m_n \in M$ such that

$$\phi_1(m, \sigma_1) = (\gamma_1, m_1) \text{ and } \phi_i(m_{i-1}, \sigma_i) = (\gamma_i, m_i), i = 2, \dots, n.$$

Also, we have

$$\lambda_e(q, m, \sigma_1 \dots \sigma_n) = \gamma_1 \gamma_2 \dots \gamma_n.$$

Since $\lambda_e(q, m, \sigma_1 \dots \sigma_n) = \lambda_e'(q', m, \sigma_1 \dots \sigma_n)$, it follows that there exists a path

$$q' \xrightarrow{\phi_1'} q_1' \xrightarrow{\phi_2'} q_2' \dots q_{n-1}' \xrightarrow{\phi_n'} q_n'$$

in \mathcal{M}' and there exist m_1', \dots, m_n' such that

$$\phi_1'(m, \sigma_1) = (\gamma_1, m_1') \text{ and } \phi_i'(m_{i-1}', \sigma_i) = (\gamma_i, m_i'), i = 2, \dots, n.$$

Using a simple induction, it follows that

$$\phi_i = \phi_i' \text{ and } m_i = m_i', i = 1, \dots, n.$$

Indeed, $\phi_1 = \phi_1'$ follows since Φ is output-distinguishable. Hence $m_1 = m_1'$. Similarly, if $m_i = m_i'$, it follows that $\phi_{i+1} = \phi_{i+1}'$ and $m_{i+1} = m_{i+1}'$. Therefore, there exists a path in \mathcal{M}' starting from q' labelled $\phi_1 \dots \phi_n$.

Let us assume there is no path in \mathcal{M} starting from q labelled $\phi_1 \dots \phi_n$. Let $k \in \{0, \dots, n-1\}$ be the maximum number such that there exists a path in \mathcal{M} starting from q labelled $\phi_1 \dots \phi_k$. Let

$$q \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} q_2 \dots q_{k-1} \xrightarrow{\phi_k} q_k$$

this path. Then

$$t_{q,m}(\phi_1 \dots \phi_n) = \sigma_1 \dots \sigma_{k+1} \text{ with } \sigma_1, \dots, \sigma_{k+1} \in \Sigma.$$

Thus there exist $\gamma_1, \gamma_2, \dots, \gamma_{k+1} \in \Gamma$ and $m_1, \dots, m_{k+1} \in M$ such that

$$\phi_1(m, \sigma_1) = (\gamma_1, m_1) \text{ and } \phi_i(m_{i-1}, \sigma_i) = (\gamma_i, m_i), i = 2, \dots, k+1.$$

Now, we prove that there is no path \mathcal{M}' starting from q' labelled $\phi_1 \dots \phi_{k+1}$. Let us assume otherwise. Then there exists a path

$$q' \xrightarrow{\phi_1'} q_1' \xrightarrow{\phi_2'} q_2' \dots q_k' \xrightarrow{\phi_{k+1}'} q_{k+1}'$$

in \mathcal{M}' . Hence

$$\lambda_e'(q', m, \sigma_1 \dots \sigma_{k+1}) = \gamma_1 \gamma_2 \dots \gamma_{k+1}.$$

Since $\lambda_e(q, m, \sigma_1 \dots \sigma_{k+1}) = \lambda_e'(q', m, \sigma_1 \dots \sigma_{k+1})$, it follows that there exists $\phi_{k+1}' \in \Phi$, $q_{k+1}' \in Q$ and $m_{k+1}' \in M$ such that

$$q_k \xrightarrow{\phi_{k+1}'} q_{k+1}' \text{ is an arc in } \mathcal{M} \text{ and } \phi_{k+1}'(m_k, \sigma_{k+1}) = (\gamma_{k+1}, m_{k+1}').$$

Since Φ is output-distinguishable, it follows that $\phi_{k+1} = \phi_{k+1}'$. This contradicts our initial assumption. Hence, there is no path \mathcal{M}' starting from q' labelled $\phi_1 \dots \phi_{k+1}$.

Therefore, we have proved that $\lambda_e(q, m, s) = \lambda_e'(q', m, s)$ implies q and q' are $\{\phi_1 \dots \phi_n\}$ -equivalent as states in \mathcal{A} and \mathcal{A}' respectively. Hence q and q' are Y -equivalent. ©

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

Theorem 4.1.9 (fundamental theorem of testing).

Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ and $\mathcal{M}' = (\Sigma, \Gamma, Q', M, \Phi, F', q_0', m_0)$ be two deterministic stream X-machines with Φ output-distinguishable and complete which compute f and f' respectively, \mathcal{A} and \mathcal{A}' their associated automata and $t: \Phi^* \rightarrow \Sigma^*$ a fundamental test function of \mathcal{M} .

We assume that \mathcal{A} and \mathcal{A}' are minimal. Then let T and W be a transition cover and a characterisation set of \mathcal{A} and $Z = \Phi^k W \cup \Phi^{k-1} W \cup \dots \cup W$, where k is a positive integer. If $\text{card}(Q') - \text{card}(Q) \leq k$ and $f(s) = f'(s) \forall s \in t(TZ)$, then \mathcal{A} and \mathcal{A}' are isomorphic.

[**Note:** For two sets $A, B \subseteq \Phi^*$, $AB = \{ab \mid a \in A, b \in B\}$]

Proof:

From lemma 4.1.8 it follows that q_0 and q_0' are TZ -equivalent. Since q_0 and q_0' are TZ -equivalent and $\text{card}(Q') - \text{card}(Q) \leq k$, from Chow [2] it follows that \mathcal{A} and \mathcal{A}' are isomorphic. ©

If our aim is to ensure that the two machines compute the same function the minimality of \mathcal{A}' is not really necessary. Then we have the following corollary.

Corollary 4.1.10.

Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ and $\mathcal{M}' = (\Sigma, \Gamma, Q', M, \Phi, F', q_0', m_0)$ be two deterministic stream X-machines with Φ output-distinguishable and complete, \mathcal{A} and \mathcal{A}' the associated automata

of \mathcal{M} and \mathcal{M}' respectively and t , T and W as above. Let $\mathcal{A}'' = (\Phi, Q'', F'', q_0'')$ be the minimal automaton of \mathcal{A}' . If \mathcal{A} is minimal, $\text{card}(Q'') - \text{card}(Q) \leq k$ and $f(s) = f'(s) \forall s \in t(TZ)$, then $f(s) = f'(s) \forall s \in \Sigma^*$.

Proof:

Let $\mathcal{M}'' = (\Sigma, \Gamma, Q'', M, \Phi, F'', q_0'', m_0)$ the stream X-machine whose associated automaton \mathcal{A}'' is the minimal automaton of \mathcal{A}' (i.e. $\mathcal{A}'' = \text{Min}(\mathcal{A}')$). Then \mathcal{M}'' and \mathcal{M}' compute the same function. From the theorem above, it follows that the associated automata \mathcal{A} and \mathcal{A}'' are isomorphic. Hence $f = f'$.

Ⓞ

4.2. The stream X-machine testing method.

Once all of this mechanism is in place we can apply the fundamental theorem to generate a test set mechanically. Thus we construct explicitly

$$Z = \Phi^k W \cup \Phi^{k-1} W \cup \dots \cup W$$

and form the set of input strings

$$X = t(TZ).$$

This is the test set we are seeking. The value of k is

$$k = n' - n$$

where n represents the number of states of the specification and n' is the (estimated) maximum number of states of the minimal associated automaton of the stream X-machine model of the implementation. In practice k is not usually large (unless there is a considerable degree of misunderstanding from the part of the developer), for especially sensitive applications one can make very pessimistic assumptions about k at the cost of a larger 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 can be modelled as a deterministic stream X-machine with the same set of basic functions Φ .

Of these assumptions the first 3 lie within the capability of the designer. An algorithm for ensuring that a stream X-machine satisfies condition 2 is given in Ipate [8]. Any stream X-machine can be transformed into a machine satisfying condition 2 by extending the definitions of the basic functions using extra input and output symbols that can be removed after testing is completed. We refer to these conditions as "design for test" conditions, without them it is going to be extremely difficult to test a system properly. The output-distinguishability condition ensures that each ϕ can be identified by examining outputs. The completeness condition ensures that each path in the associated automata of the two machines can be exercised by an appropriate input sequence, hence a path in the associated automaton of the specification can be tested against the corresponding path in the implementation. It is also clear that the designer can arrange for the associated automaton of the X-machine specification to be minimal, standard techniques from finite state machine theory are available.

The 4'th 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. In practice this will be done with a separate testing process, depending on the nature of the ϕ 's. The method explained above can be applied to test the basic processing functions if they are expressible as the computations of other, simpler X-machines. Alternatively, other testing approaches (e.g. the category partition method or a variant, see Laycock [9]) can be used, if the ϕ 's are functions that carry out simple tasks on data structures (i.e. inserting and removing items from registers, stacks, files, i.e.). If the basic processing functions are tried and tested with a long history of successful use (i.e. standard procedures, modules or objects from a library) then their individual testing could perhaps be assumed done.

The clear advantage of the method is that it *guarantees* that the system is *fault-free provided* that the basic components are *fault-free*.

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? It is clear that the complexity of the algorithm depends on the complexity of the basic functions in Φ . In Ipaté [8] it is shown that if the complexity of each ϕ is at most C , then the complexity of the algorithm that generates the test set will be proportional to

$$C \cdot p \cdot r^{k+1} \cdot n^2 \cdot (2 \cdot n' - n),$$

where $p = \text{card}(\Sigma)$ and $r = \text{card}(\Phi)$.

The maximum number of test sequences required is less than

$$n^2 \cdot \frac{r^{k+2}}{r-1} \approx n^2 \cdot r^{k+1}$$

and the total length of the test set is less than

$$n^2 \cdot n' \cdot \frac{r^{k+2}}{r-1} \approx n^2 \cdot n' \cdot r^{k+1} \text{ (see Ipaté [8]).}$$

5. Conclusions.

X-machines combine the ability to model data structures, functions and relations of languages such as Z with the dynamic features of finite state machines. It is quite straightforward to use it as the basis of a specification language which allows the designer to define suitable data types and functions which will provide the fundamental processing capabilities of the system. The control structure is clearly separated from the data structures and this has real benefits when it comes to testing. Essentially the method involves testing the two aspects separately.

Stream X-machines are a natural class of X-machines in which the inputs and outputs are sequences of characters and an input/output pair is read/produced whenever a transition is performed. Clearly, they are much more general than the finite state machine model. In fact, several case studies have shown that the model can be used to model a much wider range of systems, ranging from modelling interactive systems (see Laycock [9]) and user interfaces (Holcombe & Duan [7]) to fairly complicated hardware devices (see Chiu [1]).

Stream X-machines are the basis for our theoretical testing method. Clearly, if the implementation is not restricted in some way (i.e. it can be *any* stream X-machine), then the problem of constructing a *finite* test set that finds *all* faults is unsolvable (this is because an *infinite* memory cannot be tested using a finite set of inputs). The way to get around this problem is to use a *reductionist* approach. The stream X-machine method is neither a 'black box' nor a 'white box' one. The implementation is considered to be a 'black box' containing known elements. Therefore, we assume that the basic processing functions are implemented correctly and we test whether the *control structure* of the system (i.e. the state transition diagram) is correct or not. We also assume that some "design for testing conditions" are met.

The benefits that accrue if the method is applied is that the entire control structure of the system is tested and *all* faults detected *modulo* the correct implementation of the basic processing functions. Since the subtle interplay between the control structure of the system and the critical variables within the system is modelled explicitly this is a major advance on the highly restrictive finite state machine testing methods.

The reductionist approach can be continued further and the ϕ 's can be tested using our approach if they are represented as stream X-machines. At the bottom level of the reduction, we shall have simple functions that the developer is confident are fault-free or can be tested using alternative

methods (e.g. category-partition). This hierarchical approach suits the increasing modularisation we see in the development of software and provides a potential way of dealing with large scale systems.

A theoretical questions which remains to be answered is how complex (in terms of computational power) are the systems that our testing method can deal with ? This can be formalised as follows. Let Φ_0 be a set of elementary functions (e.g. if the memory is chosen to be a stack of characters, these could be the 'push' and 'pop' operations). Then, for $n > 1$, we define by M_n the set of all stream X-machines whose basic functions are in Φ_{n-1} or can be obtained from these using very simple transformations such as projections or parallel composition. If Φ_n is the set of functions that are computed by all the machines in M_n then, using the reductionist approach, it follows that our testing method can be used to test all the functions in $\Phi_\infty = \bigcup_{n \in \mathbb{N}} \Phi_n$

. Of course this issue requires further investigation, different Φ_∞ could be obtained depending on the way in which the memory and the set Φ_0 are chosen. However, in Ipaté [8] it is shown that, if the memory is a stack of characters and Φ_0 contains only the 'push' and 'pop' operations, then the stream X-machine acceptors (i.e. machines where $\Gamma = \emptyset$) in M_2 accept a class of languages which includes strictly the class of deterministic context-free languages.

The test sets generated by the method are of manageable size as is the application process. If the processing functions are computable by some algorithms, then the process of generating the test set can be automated. The method has been applied to several case studies and the results are encouraging. Clearly, the method has to be supported by automated systems and suitable tools that do not yet exist.

We are currently looking at a number of related issues:

can outline test sets be refined as the specification is refined?

can a general design environment be constructed which embodies tools for constructing X-machine specifications, verifying their behaviour mathematically, generating test sets and managing the refinement process?

Some results have been achieved in these areas.

References.

- [1] F. P. Chiu, Formal specification of VLSI, MPhil Thesis, University of Sheffield, U. K, 1994.
- [2] T. S. Chow, Testing software design modelled by finite state machines, *IEEE Transactions on Software Engineering*, **4**(3), 1978, pp. 178-187.
- [3] S. Eilenberg, Automata, languages and machines, Vol. A., Academic Press, 1974.
- [4] M. Fairtlough, M. Holcombe, F. Ipaté, C. Jordan, G. Laycock, Z. Duan, Using an X-machine to Model a Video Cassette Recorder, *Current issues in electronic modelling*, **3**, 1995, pp. 141-161.
- [5] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, A. Ghedamsi, Test selection based on finite state models, *IEEE Transactions on Software Engineering*, **17**(6), 1991, pp. 591-603.
- [6] M. Holcombe, X-machines as a basis for dynamic system specification, *Software Engineering Journal*, **3**(2), 1988, pp. 69-76.

- [7] M. Holcombe, Z. Duan, Traceable X-machines as models for describing User Interfaces, Departmental Report, Department of Computer Science, University of Sheffield, 1990.
- [8] F. Ipaté, Theory of X-machines and Applications in Specification and Testing, PhD thesis, University of Sheffield, UK, 1995.
- [9] G. T. Laycock, The theory and practice of specification based testing, Ph.D. thesis, University of Sheffield, UK, 1995.