

Specification and testing using generalised machines: a presentation and a case study

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.

Address for correspondence:

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 testing is a major part of software development, it rarely gets the attention it deserves from researchers, partly because its foundations are weak and ill-understood. The principal purpose of testing is to detect (and then remove) faults in a software system. However, very few of the existing methods allow the tester to make any precise statement about the type or number of faults that remain undetected after testing is completed. In particular, none of the main techniques used by the software industry can give serious guarantees that a system is fault-free after testing has been completed. This paper advocates the use of a formal method both as a specification language and as a basis of a test data selection strategy. It presents a new method for generating test cases from this type of formal specification that provides a more convincing answer to the problem of detecting *all faults* in a software system. The method is *reductionist* in the sense that it guarantees that a system is fault-free provided that its components are fault-free; in turn, the same method could be used to test the resulting sub-systems, so the reduction will continue until the components considered are either known to be correct or are fairly simple pieces of code that can be successfully tested using traditional methods. 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 of specifying computer systems. The use of X-machines as a specification tool and the testing method are illustrated with a case study.

Index terms : Functional testing, test set, correctness, formal specification, finite state machines, X-machines.

1. Introduction.

It seems a paradox that the drive for better quality systems, that has encouraged formalists to develop sophisticated and powerful techniques for system specification and verification and engineers to develop even more powerful design based test generation algorithms, has proceeded without any evidence of any interaction between the two approaches. Many researchers, especially those within the formal methods community, pour scorn on the whole concept of testing and insist that all a test can tell is that the system has failed - it cannot tell that the system is correct. Unfortunately, this is true of most of the testing methods currently in use and is clearly a major drawback. On the other hand, even the strongest proponents of program proving do not suggest that formal verification theories are practical in the context of the massive complexity of today's applications. It has been found that correctness proofs, even for small applications, are at least as difficult to construct without errors as the programs themselves.

Moreover, in practice even the systems that are proven to be correct are very seldom released without being tested. Testing attempts to detect faults that are present in the implementation so they can be removed. A substantial number of techniques for carrying out testing, and in particular for the generation of test sets exist - *functional*, where the test set is designed to verify if the system has the required functionality (Myers, 1979), *structural* (Myers, 1979), where the test set is aimed at exercising as much as the program's code as possible, or *statistical* (Waeselynck, 1994), where input data is generated based on a statistical model - and some are used with some success by the software industry. However, the major problem with these techniques is that in general they fail to give any guarantees that no faults are left in the implementation after testing has been completed - this is also true for mutation testing (Howden, 1982) ; in this case, each test is aimed at detecting a certain fault of the system, but there is no guarantee that all faults will be detected.

So, when can testing be considered done? Usually, developers decide to stop testing when no faults have been uncovered over a certain period of time or when the test data has not highlighted any faults in the most recent version of the software. In other words, the testing process ends when the tester's confidence in the system is sufficiently high. This approach is effective in many cases, but it is far from being satisfactory in some areas such as safety-critical systems. In such critical areas, the tester's confidence should not be a sufficient argument for releasing the software and some proof or assurance would be needed. Although it might appear that the achievements of software testing are belittled here, 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.

A principal reason for which the main existing testing methods fail to give any rigorous proof as to the correctness of the system is that they are not based on a firm theoretical ground. This paper proposes the use of a *formal method*, namely the *X-machine*, both as a specification language and as a basis of a test data selection strategy. It presents a method for generating test cases from this type of formal specification, that provides a more convincing approach to the problem of detecting *all faults* in a software system. The testing method is based on the theoretical results of

Ipate & Holcombe (1997). A significant part of the paper is devoted to a case study that illustrates the use of X-machines as a specification tool and the associated testing method.

2. Testing based on formal specification.

Such an integrated formal specification and testing approach would have some clear advantages. Since testing is based on a formal specification model, it can be considered at the start of the development process and the evolving system design can be continuously evaluated in the light of the needs of testing. If there are, for example, some design principles - we will call these *design for test conditions* - that can ease the testing process, then it will surely be more effective to know these principles from the outset and to follow them throughout the specification, design and implementation phases of the project than to modify an existing implementation in order to test it. Testing must be a creative activity like design rather than a destructive activity. Conversely, by developing an integrated formal specification and testing method, testing can be placed on a theoretical basis, thereby providing a more convincing approach to the problem of detecting *all faults*, so that more sensible and theoretically defensible claims about the faults that remain in the implementation after testing is completed can be made. In saying that, the problems associated with the use of formal methods in the industry are not ignored. Despite claims to the contrary, formal methods require extensive training and experience, so any formal method that is likely to become popular with the software industry has to be as user-friendly and intuitive as possible.

Previous attempts at deriving test sets from a formal specification have been made and some of them will be mentioned briefly. Bernot et al. (1991) develop a theory of testing that is used to construct test data from an algebraic specification using logical programming. The centre of their testing theory is the concept of *testing context*, defined as a triple (*hypotheses, test data set, oracle*). The testing hypotheses are conditions that the system has to meet if the test set is to detect all the faults of the system. Obviously, stronger hypothesis result in a smaller test set; conversely, a too weak hypothesis may result in an exhaustive test data set. The oracle determines if the program execution returns a correct result in response to given input data; without it, the tester would not be able to determine whether the test has been successful or not. In most cases, the solution to the oracle problem consists in a set of observability requirements that the system has to meet (an observability requirement of a program could be, for instance, to decide correctly the equality of two integers). The bottom line is that if the specification does not satisfy some appropriate conditions (testing hypotheses and observability requirements) it may be very difficult, even impossible to test.

Laycock's (1992) case study shows how to generate test cases using Ostrand & Balcer (1989) category-partition method, but based on a Z specification. Although the test cases were comprehensive and were precisely defined for the individual functions concerned, some limitations were revealed. The method does not give any guidance on combining the test sets of individual functions into higher, system wide tests, that ensure that each function is tested in all the different circumstances that are possible. Furthermore, even using a formal specification, it is

by no means clear how to formally describe the concept of category. For example, some categories are functions where others are particular parts of a parameter, there does appear to be a general rule. In conclusion, it is fair to say that category-partition testing is suitable for fairly simple systems but is not straightforward in the case of more complex applications.

Chow (1978), Fujiwara et al. (1991), Luo et al. (1994), Bhattacharria (1989) generate test cases from a finite state machine and the test produced is proved to find all faults of a system providing that it can be modelled as a finite state machine. However, not every systems is suited to finite state machines, in fact it is very difficult to represent any non-trivial data using the finite state machine model. Chow's suggestion was to separate the control of the program from its data structure and to represent the former as a finite state machine. Thus the control structure could be tested using existing finite state machine methods. However, there is still a problem: in many cases, the control of a system cannot be completely separated form its data, i.e. the next state in the transition diagram of the program depends not only on the current state and the input, but also on the values of the program variables. One could argue that the input alphabet of the finite state machine that models the system control does not represent the actual system inputs, but the operations that the system performs. If so, how do these operations map onto the real inputs that will be used for testing and who guarantees that this mapping exists for any values of the variables. Furthermore, the problem of testing the system data still remains, alternative testing techniques have to be used for this. So, it appears that a model that integrates these two aspects - control and data - of a software system and their testing is needed; such a model is the X-machine.

On the other hand, the general problem of deciding if an arbitrary system is fault-free once it has passed a testing process is not theoretically solvable. Indeed, it is impossible to establish that an arbitrary Turing machine (the model of any computer system) has a required functionality using a finite test set. If this was possible, then at least the halting problem for Turing machines (see Cohen, 1991) would have to be decidable.

The testing method presented here attempts to get around this problem by using the following two ideas. Firstly, the testing process will be approached in a reductionist manner. Such a reductionist approach would consider a system and produce a testing regime that results in the complete reduction of the test problems for the system to one of looking at the test problems for the components or reduced parts. That is,

“if the system S is composed of the parts P_1, \dots, P_n then
as a result of carrying out a testing process on S
it can deduced that S is fault-free if each of P_1, \dots, P_n are fault-free”.

A corollary is that the approach could be applied to each component P_i thus enabling the reductionist method to be continued downwards for as far as is appropriate and feasible. Since the current popular design methods in software engineering tend to emphasise the construction of systems from modules, objects and other simpler components a similar reductionist approach to testing appears to be quite natural.

Secondly, like Bernot et al. (1991), the method defines some “design for test conditions” that the system has to meet if the testing method is to guarantee system correctness.

3. Dynamic system modelling and the language of X-machines.

The formal method chosen as the basis of the integrated specification and testing method presented here is the X-machine. The model was proposed by Holcombe (1988) as a basis for a 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. Instead of using abstract symbols, the labels of the transitions are (*partial*) *functions* that operate on a *basic data set* X . The set of these (*partial*) functions, Φ , is called the *type* of the machine and represents the elementary operations that the machine is capable of performing.

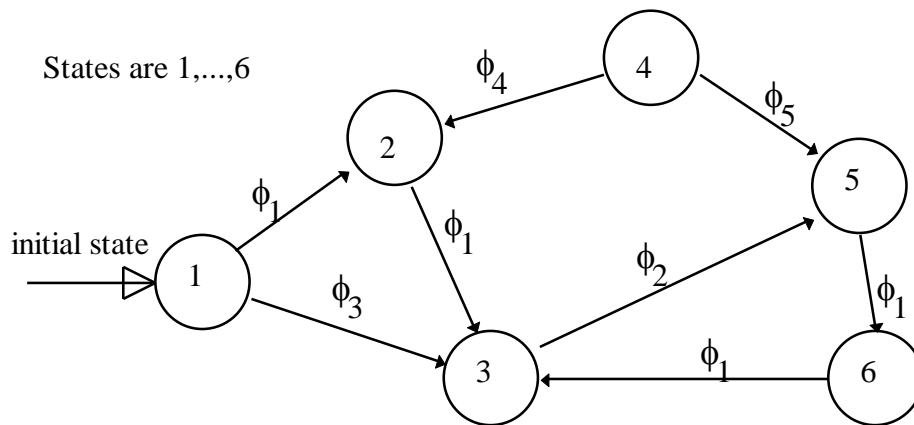


Figure 1. The state transition diagram of an X-machine.

The computation of the machine 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 and each edge is labelled by a function: ϕ_1 , ϕ_2 , etc. Sequences of functions are thus derived from each path 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 that emerge from any state are mutually disjoint).

Since the data set X can contain information about the internal memory of a system as well as different sorts of output behaviour, it is possible to model very general systems in a transparent way. The control state of the system is 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.

The functions will read inputs, datafiles, internal memory, write to all of these, refresh displays etc.

The features of the model which make it an ideal choice as the basis of a specification and testing method include:

- It is intuitive and easy to use.
- It is built on current knowledge and does not involve any revolutionary concept. Indeed, the model is a blend of state diagrams and formal descriptions of data types and functions that can be expressed easily in a language such as Z or as functions in ML or a similar functional language or using traditional mathematical notations.
- It allows for the capture of the dynamic system information in a very intuitive manner. The use of state diagram helps a great deal in this sense.
- The main existing computational models (i.e. Turing machines, pushdown automata, etc.) can be easily represented as X-machines (Ipate, 1995). This is a great advantage when it comes to testing. Indeed, a software system is something that carries out some computable function. Since the specification and the implementation are computable functions, they can be both represented as some machines. A testing method would then try to verify if these two machines are equivalent in some sense. As it has already been pointed out, the machine equivalence is an undecidable problem and the method proposed here will try to get around it by using a reductionist strategy.

It is quite straightforward to use the X-machine model 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 (i.e. the state transition diagram) 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.

4. 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.

Definition 4.1.

Let Σ and Γ two finite alphabets (called the *input* and *output alphabet* respectively). Then, a *stream X-machine* is a tuple $\mathfrak{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$, where:

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

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

Therefore, a basic processing function ϕ transforms the current memory value and the input received by the machine into an output and the next memory value.

4. F is the 'next state' (partial) function

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

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

5. $q_0 \in Q$ is the *initial state*.

6. $m_0 \in M$ is the *initial memory value*.

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 4.2.

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

if $q \xrightarrow{\phi} p$ and $q \xrightarrow{\phi'} p'$ are distinct arcs emerging from the same state q ,
then $\text{dom } \phi \cap \text{dom } \phi' = \emptyset$.

[**Note:** $\text{dom } \phi$ denotes the domain of the (partial) function ϕ].

The machine starts computing from its initial state q_0 and memory value m_0 . The (partial) function $f: \Sigma^* \rightarrow \Gamma^*$ computed by a deterministic stream X-machine associates an input sequence $\sigma_1 \dots \sigma_k$ to the output sequence $\gamma_1 \dots \gamma_k$ if and only if there exists a path

$$q_0 \xrightarrow{\phi_1} q_1 \xrightarrow{\phi_2} q_2 \dots q_{k-1} \xrightarrow{\phi_k} q_k$$

starting from q_0 and k memory values $m_1, \dots, m_k \in M$ such that

$$\phi_1(\sigma_1, m_0) = (m_1, \gamma_1), \phi_2(\sigma_2, m_1) = (m_2, \gamma_2), \dots, \phi_k(\sigma_k, m_{k-1}) = (m_k, \gamma_k).$$

5. Case study : a stream X-machine specification of a cash machine.

The following case study will use a stream X-machine to specify a very simple cash machine. For the sake of simplicity, the following assumptions are made:

- The customer is allowed to enter his/her personal identification number only once. If the attempt fails, then the card will be retained.
- Only two fixed sums of money (say £10, £20) can be withdrawn and only one attempt at withdrawing money can be made. If the amount required exceeds the balance of the account, the machine gives an appropriate warning.
- The balance of the account is also available but only if the user hasn't already attempted to withdraw money.
- The system does not update the account balance after a transaction has been made. Instead, the new transactions are recorded in a separate data structure and the main data structure is updated at certain time intervals by another system.

Stream X-machine specification.

- The input alphabet is

$$\Sigma = \text{CARDS} \cup \text{ID_NOS} \cup \{m_1, m_2, b\} \cup \{yes, no\},$$

where:

- CARDS represents the set of all the valid cash cards.
- ID_NOS represents the set of all possible user identification numbers.
- $\{m_1, m_2, b, yes, no\}$ are distinct inputs that correspond to the options available to the customer; m_1 and m_2 correspond to the two amounts of money available and b to the balance of the account. yes will be used by the customer to request a second service, and no to quit the system.

- The output alphabet is

$$\Gamma = \text{MESSAGES} \times (\text{MONEY} \cup \{null_m\}) \times (\text{BALANCES} \cup \{null_b\}) \times \{card_out, card_retained, card_unch\},$$

where:

- MESSAGES is a set of messages or sequences of messages displayed by the machine. There will be seven messages plus an empty message. Thus

$$\text{MESSAGES} = \{msg1, \dots, msg7, null_msg\}$$

where

$msg1 = \text{"Enter your personal identification number, please."}$

$msg2 = \text{"Would you like: £10, £20, Balance"}$

$msg3 = \text{"The card has been retained. } \square \text{ Insert your card, please."}$

$msg4 = \text{"Would you like another service? yes, no"}$

$msg5 = \text{"Take your card. } \square \text{ Insert your card, please."}$

$msg6 = \text{"The amount requested is not available in your account. } \square \text{ Take your card. } \square \text{ Insert your card, please."}$

$msg7 = \text{"Would you like: £10, £20."}$

- MONEY represents the amounts of money that can be withdrawn; $null_m$ denotes that no money has been output by the machine;
- BALANCES represents the set of all balances; $null_b$ denotes that the machine does not output the balance.
- $card_out$ indicates that the card has been returned to the customer; $card_retained$ denotes that the card has been retained; $card_unch$ indicates that the card state remains unchanged.

- The memory is

$$M = \text{ACC_INFO} \times \text{NEW_INFO} \times \text{CARD_NOS},$$

where:

- an element of ACC_INFO is a data structure that holds information about each account.
- an element of NEW_INFO is a data structure containing information about the transactions that have been made since the last update.
- CARD_NOS is the set of all valid card numbers.

- If the initial values of ACC_INFO, NEW_INFO and CARD_NOS are denoted by in_acc , in_n_info and in_c_no respectively, then the initial memory value of the machine will be:

$$m_0 = (in_acc, in_n_info, in_c_no),$$

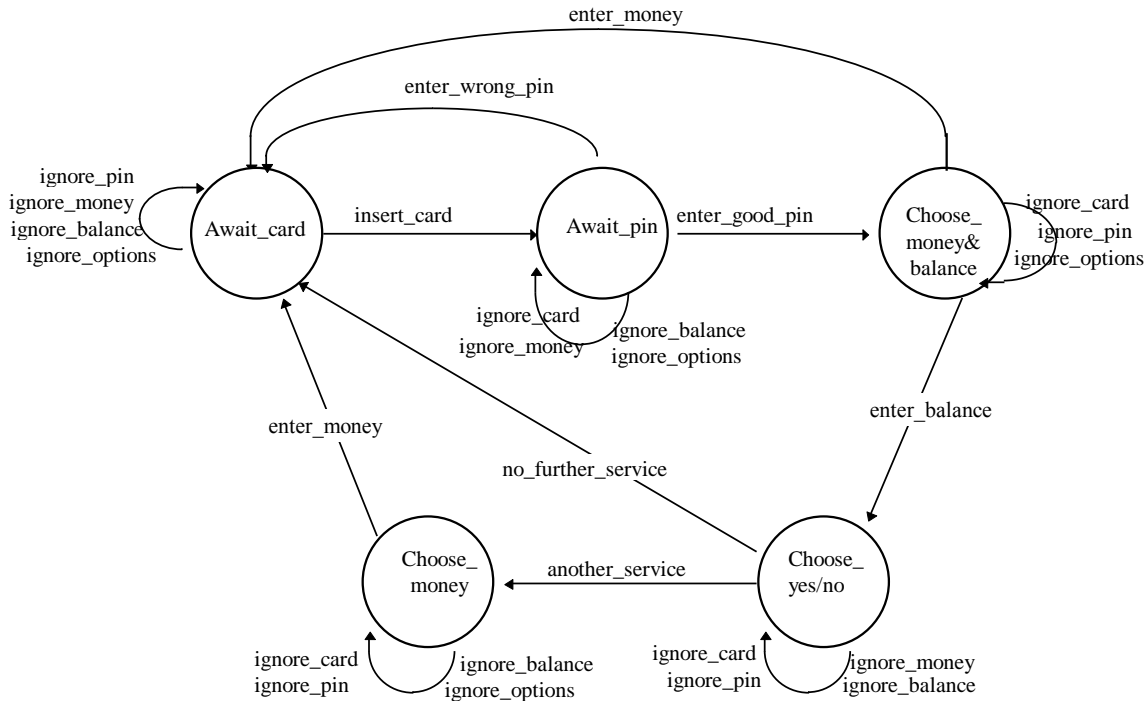


Figure 2. The state transition diagram of the stream X-machine specification.

- The representation of ACC_INFO, NEW_INFO, CARDS, ID_NOS, CARD_NOS, MONEY and BALANCES in the software modelling of the system is ignored. Instead, it is assumed that these sets are manipulated by the following functions.

[Note: B is the set of Booleans]

amount: $\{m_1, m_2\} \rightarrow \text{MONEY}$

This is an injective function that retrieves the appropriate amount of money for each of the two options.

check_account: $\text{ACC_INFO} \times \text{NEW_INFO} \times \text{CARD_NOS} \times \{m_1, m_2\} \rightarrow \text{B}$

Checks if the amount requested is less then the current balance of the account.

update_account: $\text{ACC_INFO} \times \text{CARD_NOS} \times \{m_1, m_2\} \rightarrow \text{NEW_INFO}$

Records the amount of money withdrawn.

get_balance: ACC_INFO \times CARD_NOS \rightarrow BALANCES

Retrieves the balance of the account.

get_card_no: CARDS \rightarrow CARD_NOS

This is a bijective function that retrieves the card number

get_pin: ACC_INFO \times CARD_NOS \rightarrow ID_NOS

Retrieves the personal identification number corresponding to a certain card number.

- The state set, the type Φ and the next state function F result from the state-transition diagram represented in figure 2. The initial state is `Await_card`.
- The basic processing functions are partial functions from $M \times \Sigma$ to $\Gamma \times M$ defined as follows.

[**Note:** $acc \in ACC_INFO$, $n_info \in NEW_INFO$, $c_no \in CARD_NOS$.]

insert_card((acc, n_info, c_no), x) =

if $x \in CARDS$ then

$((msg1, null_m, null_b, card_unch), (acc, n_info, get_card_no(x)))$

When the card is inserted, the system reads the card number and the customer is asked to enter his/her personal identification number.

enter_good_pin((acc, n_info, c_no), x) =

if $x \in ID_NOS$ and $get_pin(acc, c_no) = x$ then

$((msg2, null_m, null_b, card_unch), (acc, n_info, c_no))$

If the personal identification number is correct, then the customer will be allowed to choose one of the following options: two amounts of money and balance.

enter_wrong_pin((acc, n_info, c_no), x) =

if $x \in ID_NOS$ and $\neg(get_pin(acc, c_no) = x)$ then

$((msg3, null_m, null_b, card_retained), (acc, n_info, in_c_no))$.

If the customer enters an incorrect personal identification number, then the card will be retained.

enter_money((acc, n_info, c_no), x) =

if $x \in \{m_1, m_2\}$ then

if $check_account(acc, n_info, c_no, x)$ then

$((msg5, amount(x), null_b, card_out),$

$(acc, update_account(acc, c_no, x), in_c_no))$,

else

$((msg6, null_m, null_b, card_out), (acc, n_info, in_c_no))$,

The required amount of money is output if it is available in the customer's account and a warning message is given otherwise. The customer's card is released, the system returns to the initial state and the message "Insert your card, please" is displayed.

```
enter_balance((acc, n_info, c_no), b) =
  ((msg4, null_m, get_balance(acc, c_no), card_unch), (acc, n_info, c_no))
```

The balance of the account is output. The customer is asked if he/she wants another option.

```
another_service((acc, n_info, c_no), yes) =
  ((msg7, null_m, null_b, card_unch), (acc, n_info, c_no))
```

If the “yes” option is chosen, then the two “money” options will be displayed.

```
no_further_service((acc, n_info, c_no), no) =
  ((msg5, null_m, null_b, card_out), (acc, n_info, in_c_no))
```

If the “no” option is chosen, then the system returns to the initial state and the customer's card is released. The message “Insert your card, please” is then displayed.

The following functions basically “ignore” a certain input (or number of inputs).

```
ignore_card(((acc, n_info, c_no), x) =
  if x ∈ CARDS then
    ((null_msg, null_m, null_b, card_unch), (acc, n_info, c_no))
```

```
ignore_pin(((acc, n_info, c_no), x) =
  if x ∈ ID_NOS then
    ((null_msg, null_m, null_b, card_unch), (acc, n_info, c_no))
```

```
ignore_money(((acc, n_info, c_no), x) =
  if x ∈ {m_1, m_2} then
    ((null_msg, null_m, null_b, card_unch), (acc, n_info, c_no))
```

```
ignore_balance(((acc, n_info, c_no), b) =
  ((null_msg, null_m, null_b, card_unch), (acc, n_info, c_no))
```

```
ignore_options(((acc, n_info, c_no), x) =
  if x ∈ {yes, no} then
    ((null_msg, null_m, null_b, card_unch), (acc, n_info, c_no))
```

6. Stream X-machine testing method - preliminary concepts.

The testing method presented here is based on the theoretical results given of Ipate (1995) and Ipate & Holcombe (1997) that represent a generalisation of Chow’s finite state machine testing. The method works on the assumption that both the system specification and implementation can be represented as stream X-machines and employs a reductionist approach in the sense that it assumes that the basic components of the specification, the processing functions Φ , are

implemented correctly - hence, the two machines (one representing the implementation, the other the specification) are assumed to have the same type Φ . This will be ensured by a separate testing process as discussed later.

Before describing the testing method in detail, some preliminary concepts have to be introduced:

6.1. Associated automata. Characterisation set and transition cover.

Given an X-machine $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$, it can be converted into a finite state machine $\mathcal{A} = (\Phi, Q, F, q_0)$ by treating the elements of Φ as abstract input symbols. In effect, the memory structure and the semantics of the elements of Φ are “forgotten”. This will be called *the associated automaton* of \mathcal{M} .

The following concepts are largely from Chow (1978) and refer to finite state machines.

Definition 6.1.1.

Let Φ be an alphabet, q and q' two states of a finite state machine $\mathcal{A} = (\Phi, Q, F, q_0)$ over the input alphabet Φ and let $Y \subseteq \Phi^*$ be a set of input sequences. Then q and q' are said to be Y -distinguishable if there exists $y \in Y$ 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 .

Also, two finite state machines \mathcal{A} and \mathcal{A}' over the same input alphabet Φ are said to be Y -distinguishable if their initial states are Y -distinguishable.

Definition 6.1.2.

A finite state machine $\mathcal{A} = (\Phi, Q, F, q_0)$ is said to be minimal if:

1. 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 .
2. For any two distinct states $q, q' \in Q$ there exists $Y \subseteq \Phi^*$ such that q and q' are Y -distinguishable.

Definition 6.1.3.

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.

Example 6.1.4.

A characterisation set for the associated automaton of the above bank machine specification is:

$$W = \{\text{insert_card, enter_good_pin, enter_balance, another_service}\}$$

Definition 6.1.5.

Let $\mathcal{M} = (\Phi, Q, F, q_0)$ be a minimal 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{M} into q from the initial state q_0 , such that $y \in T$ and $y\phi \in T, \forall \phi \in \Phi$.

[**Note:** ε denotes the empty string. For two strings $y, z \in \Phi^*$, yz denotes their concatenation. For two sets $Y, Z \subseteq \Phi^*$, YZ is defined by

$$YZ = \{yz / y \in Y, z \in Z\}$$

[**Note:** W and T may contain sequences that are not feasible paths of \mathcal{M}]

Example 6.1.6.

A transition cover set for the associated automaton of the above bank machine specification is:

$$T = \{\varepsilon\} \cup S\Phi,$$

where

$$S = \{\varepsilon, \text{insert_card}, \text{insert_card enter_good_pin}, \\ \text{insert_card enter_good_pin enter_balance}, \\ \text{insert_card enter_good_pin enter_balance}, \\ \text{another_service}\}$$

Note that the minimality of the machine ensures the existence of a characterisation set and of a transition cover.

The following theorem is the basis of Chow's finite state machine testing.

Theorem 6.1.7.

Let \mathcal{M} and \mathcal{M}' be two minimal finite state machines over the input alphabet Φ , n the number of states of \mathcal{M} and n' the number of states of \mathcal{M}' . Let T and W , respectively, be a transition cover and a characterisation set of \mathcal{M} and $Z = \Phi^k W \cup \Phi^{k-1} W \cup \dots \cup W$. If $\text{Card}(Q') - \text{Card}(Q) \leq k$ and \mathcal{M} and \mathcal{M}' are TZ -equivalent, then \mathcal{M} and \mathcal{M}' are isomorphic.

The idea is that the transition cover T ensures that all the states and all the transitions of \mathcal{M} are also present in \mathcal{M}' and Z ensures that \mathcal{M}' is in the same state as \mathcal{M} after each transition is performed. Notice that Z contains W and also all sets $\Phi^i W, i = 1, \dots, k$. This ensures that \mathcal{M}' does not contain extra states. If there were up to k extra states, then each of them would be reached by some input sequence of up to length k from the existing states.

[**Note:** Chow assumes that the two automata have outputs (i.e. each transition will also produce an output symbol) and are complete (i.e. there is a transition from each state on each input symbol) but the above theoretical result is not affected by these assumptions]

6.2. Design for test conditions.

The stream X-machine testing method will require that the type Φ of the two machines satisfies two conditions, completeness w.r.t. M and output-distinguishability, as defined in what follows.

Definition 6.2.1.

A type Φ is called *output-distinguishable* if:

$$\begin{aligned} \forall \phi_1, \phi_2 \in \Phi, \text{ if } \exists m \in M, \sigma \in \Sigma \text{ such that} \\ \phi_1(m, \sigma) = (\gamma, m_1') \text{ and } \phi_2(m, \sigma) = (\gamma, m_2') \\ \text{for some } m_1', m_2' \in M, \gamma \in \Gamma, \text{ then } \phi_1 = \phi_2. \end{aligned}$$

What this is saying is that any two different processing functions will produce different outputs on each memory/input pair.

Definition 6.2.2.

A processing function $\phi \in \Phi$ is called *complete w.r.t. M* if:

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

A Φ is called *complete w.r.t. M* if:

$$\forall \phi \in \Phi, \phi \text{ is complete w.r.t. } M.$$

In other words, any basic function will be able to process all memory values.

These two conditions are required of the specification machine and they will be referred to as “design for test conditions”. Although these might appear as being quite restrictive, they can be easily introduced into a specification by simply extending the definitions of the Φ functions in a suitable manner, introducing extra input symbols and augmenting the output alphabet (Ipate, 1995). In very simple terms, this can be done as follows:

Let $\phi \in \Phi$ be a processing function that is not complete w.r.t. M and let $\alpha \notin \Sigma$ be an extra input. Then ϕ_e defined by

$$\phi_e(m, \sigma) = \begin{cases} \phi(m, \sigma), & \text{if } (m, \sigma) \in \text{dom } \phi \\ (\gamma_0, m), & \text{if } \sigma = \alpha \text{ and } m \in M \end{cases}$$

where $\gamma_0 \in \Gamma$ is arbitrarily chosen, is complete w.r.t. M - obviously, ϕ_e will be a processing function of a stream X-machine whose input alphabet includes $\Sigma \cup \{\alpha\}$ and has the same memory M .

[**Note:** dom ϕ is the domain of ϕ]

The above extension will be performed for all non-complete ϕ 's while the complete ϕ 's will remain unchanged; the resulting type will be named Φ_e . If the resulting machine is to remain deterministic, then any two ϕ 's that are used as labels for arcs with the same initial state will use different extra inputs. Thus the maximum number of extra inputs required will be at most

$$N_1 = \max_{\phi \in \Phi} \text{Card}\{\phi' \in \Phi \mid \exists q \in Q \text{ such that } F(q, \phi) \neq \emptyset \text{ and } F(q, \phi') \neq \emptyset\}.$$

In most cases $N_1 \ll \text{card}(\Phi)$ so the number of extra inputs is usually small.

The output-distinguishability will be achieved by a further augmentation process: the output alphabet of the resulting machine will be $\Gamma \times H$, where the ‘H’ component of the output will be used to distinguish between any two elements of Φ_e . Further details can be found in Ipate (1995).

Clearly, the bank machine specification in section 5 is output-distinguishable. Also, if ID_NOS has at least two elements and for each card number there is a valid identification number, it can be easily verified that the type is also complete w.r.t. M.

6.3. Fundamental test function.

The last theoretical concept introduced is that of *fundamental test function* of a stream X-machine, defined as a means of converting sequences of ϕ 's into sequences of inputs. This will be used to test paths of the machine using appropriate input sequences.

Definition. 6.3.1.

Let $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ be a deterministic stream X-machine with Φ complete w.r.t. M and let $q \in Q, m \in M$. A function $t_{q,m}: \Phi^* \rightarrow \Sigma^*$ will be defined recursively as follows:

1. $t_{q,m}(\varepsilon) = \varepsilon$, where ε 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

$$(m_n, \sigma_{n+1}) \in \text{dom } \phi_{n+1}.$$

where m_n is the final value computed by the machine along the above path on the input sequence $t_{q,m}(\phi_1 \dots \phi_n)$.

[**Note:** Since Φ is complete w.r.t. M, there exists such σ_{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. $s = \sigma_1 \dots \sigma_n$, where σ_1 exercises ϕ_1 , σ_2 exercises ϕ_2 , ..., 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 that will attempt to exercise an arc labelled ϕ_{n+1} (i.e. the pair ' σ_{n+1} /memory value' is in the domain of ϕ_{n+1}), thus making sure that such an arc does not exist.

Also, $\forall \phi_{n+2}, \dots, \phi_{n+k} \in \Phi$,

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

therefore no attempt will be made to exercise the subsequent ϕ 's.

Note that a test function is not uniquely determined, many different possible test functions exist.

Example 6.3.2.

For the stream X-machine specification of the bank machine, let y_1, \dots, y_7 be the following sequences of ϕ 's.

$$y_1 = \text{insert_card}$$

$$y_2 = y_1 \text{enter_good_pin}$$

$$y_3 = y_2 \text{enter_balance}$$

$$y_4 = y_3 \text{another_service}$$

$$y_5 = y_4 \text{enter_balance}$$

$$y_6 = y_5 \text{insert_card}$$

$$y_7 = y_6 \text{enter_wrong_pin}$$

Then, the values of a fundamental test function t can be generated as follows:

- Since there exists a path labelled y_4 from the initial state `Await_card`, $t(y_4)$ will be a sequence of inputs that forces the machine to follow this path.

Let

$$\text{test_card} \in \text{CARDS},$$

$$\text{test_no} = \text{get_card_no}(\text{test_card})$$

and let also

$$\text{test_id} \in \text{ID_NOS}$$

such that

$$\text{get_pin}(\text{in_acc}, \text{test_no}) = \text{test_id}$$

(i.e. $test_id$ is the identification number for $test_card$).

Then the following values of t can be chosen:

$$\begin{aligned} t(y_1) &= test_card \\ t(y_2) &= test_card\ test_id \\ t(y_3) &= test_card\ test_id\ b \\ t(y_4) &= test_card\ test_id\ b\ yes \end{aligned}$$

- Now, there is no path labelled y_5 in the state-transition diagram of the system, thus

$$t(y_5) = t(y_4)\ enter_balance$$

and

$$t(y_7) = t(y_6) = t(y_5)$$

7. The testing method.

The basis of the testing method proposed here is a theoretical result proved by Ipate & Holcombe which shows that

$$Y = t(TZ) = \bigcup_{y \in (TZ)} y$$

is a test set that finds *all* faults of the implementation (i.e. the specification and implementation have the same input/output behaviour) and works on the following assumptions:

- The specification is a deterministic stream X-machine \mathcal{M} .
- The associated automaton \mathcal{A} of \mathcal{M} is minimal.
- The set of basic processing functions Φ of \mathcal{M} is output-distinguishable and complete w.r.t. M .
- The implementation can be modelled as a deterministic stream X-machine \mathcal{M}' with the same set of basic functions Φ .

The following notation has been used:

- t is a fundamental test function of \mathcal{M}
- T is a transition cover of \mathcal{A}
- W is a characterisation set of \mathcal{A}
- $Z = \Phi^k W \cup \Phi^{k-1} W \cup \dots \cup W$
- $k = n' - n$
- n is the number of states of the stream X-machine specification \mathcal{M}
- n' is the (estimated) maximum number of states of \mathcal{M}'

Ipate & Holcombe prove that if the above conditions are met then \mathcal{M} and \mathcal{M}' will compute identical functions. The proof reduces the problem of proving the functional equivalence of the two machines to proving the equivalence of their associated automata. In turn, this latter problem is addressed using Chow's results.

Obviously, the method relies on the specification being a deterministic stream X-machine. Condition 2 and 3 lie within the capability of the designer. The designer can arrange for the associated automaton of the specification X-machine to be minimal, standard techniques from finite state machine theory are available. Also, as already discussed, Φ can be transformed into a type that is complete w.r.t. M and output-distinguishable by using a few extra inputs and augmenting the output alphabet. Obviously, this transformation will also result in an augmentation of the implementation, but this does not constitute a serious problem: the extra inputs can be filtered out and an output filter could be used to map the augmented output alphabet onto the real outputs.

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. In practice this will be done with a separate testing process depending on the nature and complexity of ϕ 's:

1. If these are complex functions then they have to be expressed as the computations of other, presumably much simpler, X-machines, and the same method can be applied.
2. Otherwise, some other testing method for testing simple functions can be used - perhaps the category-partition method (Ostrand & Balcer, 1989) or structural methods (Ntafos, 1989) since a high level of coverage can be achieved for such simple routines.
3. If the basic processing functions are tried and tested with a long history of successful use - e.g. they are standard procedures, modules or objects from a library - then their individual testing can be assumed done.

At first sight, it might appear that what the method really does is shift the burden onto the testing of ϕ 's. This is not so. What is proposed here is a gradual testing process in which at each level the system is assumed to be made of reliable components. In likely applications of the method, this will be successively applied to the hierarchy of stream X-machines that are created when the basic functions Φ are considered 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. Ultimately, at the bottom level, the basic functions need to be tested in some suitable way or assumed to be implemented correctly.

The method also assumes that the maximum number of states of the implementation can be estimated. If the software developer has spent adequate effort in analysing the specification and constructing the design it is reasonable to expect k not to be too large. For critical applications, one can make very pessimistic assumptions about k at the expense of a large set. Also, a hidden assumption of the method is that a reset operation (i.e. an extra input which causes the machine to change back to the initial state q_0 and memory value m_0) is implemented correctly, so that the next test sequence can be applied from q_0 and m_0 . In the worst case, this corresponds to restarting the system.

The final question that needs to be addressed is concerned with the practicality of the method. If the processing functions are computable by some algorithms, then the process of generating the test set can be automated. Clearly, the complexity of the algorithm will depend on the complexity of the basic functions in Φ . Ipate (1995) shows 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 (i.e. the total number of the input sequences of the test set) is less than

$$n^2 \cdot n' \cdot \frac{r^{k+2}}{r-1} \approx n^2 \cdot n' \cdot r^{k+1} \text{ (Ipate, 1995).}$$

7.1. The testing method applied to the case study.

Let us see how the method can be used to test the cash machine specified in section 5. The success of the method relies on the basic functions Φ being correctly implemented, so this has to be ensured first. This is in turn a hierarchical process since the ϕ 's are themselves constructed using other lower-level functions. Thus the following steps are required:

1. Testing the functions that manipulate the inputs and the system data, i.e. `check_account`, `update_account`, `get_card_no`, etc. In practice, these perform fairly standard operations on common data structures (i.e. adding an item to a file, retrieving an item from a file, converting a string into a positive integer, etc.) and are usually standard routines from a library so they can be assumed to be fault-free. Alternatively, a category-partition testing (Ostrand & Balcer, 1989) is adequate for testing such routines.
2. Testing the ϕ 's. These are straightforward constructions ("if/else" statements, functional composition, etc.) that use the above functions. A simple category partition or structural (Ntafos, 1989) testing is sufficient, provided that the lower-level functions (those of type 1) have already been tested.

However, for more general functions the method would require that these can be expressed as the computations of other stream X-machines. Just for the sake of argument, let us assume that `ID_NOS` is the set of all sequences of 4 digits and that no function that checks if two such sequences are identical is available. Then, `enter_good_pin`, for example, could be represented as a stream X-machine which reads one digit at a time and checks if the digit read matches the corresponding digit of the actual personal identification number. For a more complex example where the ϕ 's are expressible as stream X-machines see Ipate (1995).

3. Testing the control structure of the system in order to ensure that the ϕ 's are integrated correctly. This is where the test set given above is used.

For example, if the estimated n' is $n + 2$ then the test set can be obtained by generating the values of a test function t for the set

$$TW \cup T\Phi W \cup T\Phi^2W,$$

where W and T are those given in examples 6.1.4 and 6.1.6 respectively. Further details can be found in Ipate (1995).

8. Conclusions.

The X-machine is quite straightforward to use as a specification method. It combines the ability to model data structures, functions and relations of languages such as Z or VDM with the graphical advantages of finite state machines. The state-transition diagram of the machine represents the control structure of the machine and this is separated from the data set and the definitions of processing functions. On the other hand, the designer is allowed to define suitable data types and functions which will provide the fundamental processing capabilities of the system. These can be expressed using Z or a functional language such as ML or using traditional mathematical notation. Alternatively, the processing functions could be X-machines themselves, so the X-machine model can be used at more than one level in the specification. This combination of state diagrams, data structures and processing functions makes the X-machine a convenient and intuitive specification method in which different features of the system will be communicated at the appropriate level in an intuitive way. Several case studies have shown that the method can be used to model a wide range of systems, from interactive systems (Laycock, 1992) and user interfaces (Holcombe & Duan, 1990) to fairly complicated hardware devices (Chiu, 1994).

The testing method presented here is a significant advance on the highly restrictive finite state machine testing developed by Chow (1978) and Fujiwara et al. (1991). Chow states that his method can be used to test the control structure of a program but, as it has already been pointed out, separating completely the control of a system from its data is often impossible. On the other hand, the X-machine model integrates these two aspects of the system, but allows them to be described and viewed separately.

The reductionist approach used allows the separation of the testing process into two separate tasks: testing the processing functions Φ and testing the control structure of the system (using the method presented here). In turn, a ϕ can fall into one of the following two categories.

- In many of the case studies investigated so far, the basic functions that need to be used are typically straightforward ones that carry out simple tasks on simple structures or “if/else” statements containing such functions. Many of these are standard routines from libraries or can be obtained very easily from such routines. Traditional functional methods such as category-partition testing are sufficient to test such functions.
- Alternatively, more complex ϕ 's have to be represented as the computations of some other stream X-machines, so that they can also be tested using the same method and the reduction will continue. This hierarchical approach suits the increasing modularisation in the development of software and provides a potential way of dealing with large scale systems.

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

The method has been applied to several case studies, including safety critical systems (Holcombe et al., 1995), and the results are encouraging - i.e. all seeded errors were detected. Clearly, some automatic support is needed and suitable tools are currently under development.

An important theoretical questions that remains to be answered to is how complex (in terms of computational power) are the systems that can be tested using the method and the reductionist strategy proposed here. In general terms, this can be formalised as follows. Let Φ_0 be a set of elementary functions. Then, for $n > 0$ let M_n be the set of all stream X-machines whose basic functions are in Φ_{n-1} or can be obtained from these using some elementary transformations (e.g. projections and parallel composition). If Φ_n is the set of all functions that are computed by machines in M_n then, using a reductionist strategy, it follows that the method can cope with all functions in $\Phi_\infty = \bigcup_{n \in \mathbb{N}} \Phi_n$. This issue requires further investigation, different Φ_∞ could be obtained for different memory and Φ_0 chosen. An initial result is given by Ipate (1995) and shows that if the memory is a stack of symbols and Φ_0 contains only the 'push' and 'pop' operations, then the stream X-machine acceptors (i.e. $\Gamma = \emptyset$) in M_2 will accept a class of languages that includes strictly the class of deterministic context-free languages.

References.

- Bernot, G., Gaudel, M.C., Marre, B. (1991) 'Software testing based on a formal specification', *Software Engineering Journal*, 387 - 405.
- Bhattacharrya, A. (1989) *Checking Experiments in Sequential Machines*, Wiley Eastern, New Delhi.
- Chiu F. P. (1994) *Formal specification of VLSI*, MPhil Thesis, University of Sheffield, U. K.
- Chow T. S. (1978), 'Testing software design modelled by finite state machines', *IEEE Transactions on Software Engineering*, **4**(3), 178-187.
- Cohen, D. I. A. (1991) *Introduction to Computer Theory*, John Wiley & Sons.
- Fairtlough, M., Holcombe, M., Ipate, F., Jordan, C., Laycock, G., Duan, Z. (1995) 'Using an X-machine to Model a Video Cassette Recorder', *Current issues in electronic modelling*, **3**, 141-161.
- Fujiwara S., von Bochmann G., Khendek F., Amalou M., Ghedamsi A. (1991) 'Test selection based on finite state models', *IEEE Transactions on Software Engineering*, **17**(6), 591-603.
- Holcombe, M., Ipate, F., Grondoudis, A. (1995) 'Complete Functional Testing of Safety-Critical Systems', *Proceedings of Second IFAC Workshop on Safety and Reliability in Emerging Control Technologies*, Daytona Beach, Florida, USA.

- Holcombe, M. (1988) 'X-machines as a basis for dynamic system specification', *Software Engineering Journal*, **3**(2), 69-76.
- Holcombe, M., Duan, Z. (1990) *Traceable X-machines as models for describing User Interfaces*, Departmental Report, Department of Computer Science, University of Sheffield.
- Holcombe, M. (1993) 'An Integrated Methodology for the Specification, Verification and Testing of Systems', *Software Testing, Verification and Reliability*, **3**(3/4), 149-163.
- Howden, W.E. (1982) 'Week mutation testing and completeness of test sets' *IEEE Transactions on Software Engineering*, **8**(4), 371-379.
- Ipate F. (1995) *Theory of X-machines with Applications in Specification and Testing*, PhD thesis, University of Sheffield, U.K.
- Ipate F., Holcombe M. (1997) 'An integration testing method that is proved to find all faults', *International Journal of Computer Mathematics*, **63**, 159-178.
- Laycock G. T. (1995) *The theory and practice of specification based testing*, Ph.D. thesis, University of Sheffield, U.K..
- Laycock, G. (1992) 'Formal specification and testing: a case study', *Software Testing Verification and Reliability*, **2**(1), 7-23.
- Luo, G., v. Bochmann, G. and Petrenko, A. (1994) 'Test Selection based on Communicating Nondeterministic Finite-State machines Using a Generalised Wp-method' *IEEE Transactions on Software Engineering*, **20**(2), 149-161.
- Myers G. J. (1979), *The art of software testing*, John Wiley and Son.
- Ntafos, S. C (1989) 'A Comparison of Some Structural Testing Strategies' *IEEE Transactions on Software Engineering*, **14**(6), 868-873.
- Ostrand, T.J., Balcer, M.J. (1989) 'The Category-Partition Method for Specifying and Generating Functional Tests', *Communication of the ACM*, **31**(6), 667-686.
- Waeselynck., H. (1994) 'An Experiment with Statistical Testing', *EuroSTAR*, 10/1.