

An Integrated Refinement and Testing Method for Stream X-machines

Florentin Ipate¹, Mike Holcombe²

¹ Faculty of Science, University of Pitesti, Pitesti, Romania
(e-mail: fipate@ifsoft.ro)

² Department of Computer Science, University of Sheffield, Sheffield, UK
(e-mail: m.holcombe@dcs.shef.ac.uk)

Received: March 29, 2000; revised version: December 2, 2001

Abstract. Over the last decade, *stream X-machines* have been used in order to specify a range of systems. One of the strengths of this approach is that, under certain well defined conditions, it is possible to produce a test set that is guaranteed to determine the correctness of the implementation under test. However, if X-machines are to be used in practice as a tool for specification and test generation, there needs to be ways of developing existing specifications into more complex and more detailed versions through a process of *refinement*. Associated with the refinement of the specification, there needs to be methods of *refining the corresponding test sets*, that is to construct the test set in parallel with the specification and to distribute the testing into smaller chunks, with major cost and time savings. A few such specification and testing refinements of X-machine have already been investigated. This paper introduces a new type of X-machine refinement, called *simple covering*, which expands the input-output behaviour of an existing X-machine. Associated with this process, the corresponding refinement of the test set is described and a method of testing X-machines constructed as simple coverings is developed.

Keywords: Stream X-machine, Finite state machine, Refinement, Testing.

1 Introduction

The subject of software quality, of delivering a *correct* implementation of the *correct* system, has occupied the attention of many software engineers and generated a substantial literature. A main area of emphasis, particularly amongst the academic community, is the use of formal methods for the specification and

verification of software. One of the strengths of developing a formal specification of a system is the fact that it can act as a reference point for the project development, it can define, in the form of a quasi-legal statement, the required outcome of the project and it can also be a source of information that can be used to establish that the implementation is correct. In the past, the main way that this second activity proceeded was by formal verification, that is formally proving that the implementation satisfied the specification. This is usually achieved through a process of refinement: a specification at a high level of abstraction is produced first, this is then refined through a series of verified, more concrete, intermediate specifications until a guaranteed correct executable implementation is produced. However, there is a long way to go until formal verification will be able to deal with the massive complexity of today's applications.

In practice, a system will always be subject to *extensive dynamic testing* in order to seek the assurance, that is required, of its compliance with its specification. The generation of test sets is a massive task, in practice. It has to be automated and this means that the test generation system must be given a formal description of the system in some form in order for the test generation to be carried out. Few major projects can boast of a complete formal specification and so most tools in use in industry use the source code as the main basis for information about the system and generate test sets based on analysis of the structure of the code. Since the code may be flawed, this reliance compromises the entire testing and quality assurance process.

Over the last decade, there has been some interest in trying to use the information in a formal specification as a basis for test set generation. This would then be a strong incentive to construct a formal specification as part of the design process. However, the type of specification language used and the test set generation strategy chosen can determine how effective the final test process is. The specification, being a formal description, can be used by a tool to generate test inputs. Laycock [23], Stocks and Carrington [26] consider the issue of generating test cases from Z specifications, here the emphasis is on generating efficient test sets and nothing is said about their effectiveness, that is how many faults may remain after testing is completed. Bernot et al. [1] 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 (i.e. the output of the system has to provide enough information for the evaluation of the testing process). 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.

A considerable amount of work has been done in the area of test generation for the software modelled by *finite state machines* [2], [3], [8], [11], [12], [24]. Here, the assumption is that the control aspects of the software can be somehow separated from the system data and can be modelled as a finite state machine, so that such methods are used to test the control of the program. However, for non-trivial systems, it is usually impossible to describe the system control independent of its data processing. Furthermore, it may be very difficult, or even impossible, to derive the control structure from a system specification, unless appropriate specification languages are used. For example, a finite state machine can be derived from a Z specification only under certain conditions and, even in this case, the resulting machine may be of an unmanageable size [10]. A more useful approach would be to use a specification model that separates the control structure from the outset, while allowing it to be integrated with the data processing.

Such a model is the *X-machine* [16] [17], a blend of finite state machines, data structures and processing functions. Introduced by Eilenberg [6] in 1974, *X-machines* were proposed by Holcombe [13] 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. In its essence, an X-machine is like a finite state machine, but with one important difference: the labels of the transitions are (partial) functions instead of abstract symbols. Of particular practical importance are those X-machines, called *stream X-machines*, where these functions process input and output symbols while changing the value of an internal memory or data set M . M is often an array consisting of fields such as registers, stacks, database filestores, etc., so it is possible to model very general systems in a transparent way. However, stream X-machines are best suited for specifying interactive systems such as user interfaces.

A method of testing systems specified as stream X-machines was developed by Ipate and Holcombe [18], [21]. The *stream X-machine testing method* is proved to detect all faults of the implementation provided that two major conditions are met. Firstly, the system specification has to meet some “*design for testing conditions*” – here the idea is similar to that advocated by Bernot et al. [1]: if the specification does not satisfy some appropriate conditions it may be impossible to test. Secondly, the implementation of the system has to be made of correct components, that is the implementations of the processing functions have to be correct. In practice this second requirement is ensured through a separate process of testing, depending on the nature of the processing functions: complex functions can be themselves expressed as stream X-machines and the same testing method can be applied again; simple functions can be tested using alternative methods such as Ostrand and Balcer’s category-partition [25] or a

variant; standard functions or procedures from a library can be even assumed to be fault-free if they are tried and tested with a long history of successful use. Note that this method does not claim to test an arbitrary system, thus it does not contradict in any way the undecidability of the halting problem for Turing machines [4].

Obviously, an important theoretical question that has to be answered is how complex are the systems that can be specified and tested using the stream X-machine method? This problem 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 X_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 X_n then the 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, Ipate [22] shows that, if the memory is a stack of characters and Φ_0 contains only the “push” and “pop” operations, then the stream X-machines in X_2 will accept a class of languages that strictly includes the class of deterministic context-free languages.

If X-machines are to be used in practice as a tool for specification and test generation, there needs to be ways of developing existing specifications into more complex and more detailed versions through a process of *refinement*; in practice, for large scale systems, attempts to build the specification in one go are rarely successful; instead, a process of refinement is usually used. Associated with the refinement of the specification, there needs to be methods of *refining the corresponding test sets*, that is to construct the test set in parallel with the specification and to distribute the testing into smaller chunks, with major cost and time savings. Such specification and testing refinements of X-machine are investigated in [16] and [20] and used in several case studies [7].

This paper introduces a new type of X-machine refinement, called *simple covering*, that expands the input-output behaviour of an existing X-machine. Associated with this process, the corresponding refinement of the test set is described and a method of testing X-machines constructed as simple coverings is developed. The paper is structured as follows. Sections 2 and 3 provide brief presentations of the stream X-machine model and the stream X-machine testing method, respectively. The simple covering is presented in sections 4, 5 and 6: the transformation between the original and refined machines is presented both in terms of their input-output behaviour (section 4) and of their internal structure (section 5) and the equivalence of these two aspects is then proved (section 6). The corresponding method of refining test sets is presented in sections 7 and 8; section 7 presents the theory while section 8 describes the application of the method. The last section contains conclusions and further work.

Before we go any further, we introduce the notation used in this paper. When considering sequences of inputs or outputs we will use A^* to denote the set of finite sequences with members in A . ϵ will denote the empty sequence. For a sequence $a \in A^*$, $|a|$ will denote the length of a , i.e. the number of elements of a ; $|\epsilon| = 0$. For $a, b \in A^*$, ab will denote the concatenation of sequences a and b . a^n is defined by $a^0 = \epsilon$ and $a^n = a^{n-1}a$ for $n \geq 1$. For $U, V \subseteq A^*$, $UV = \{ab | a \in U, b \in V\}$; U^n is defined by $U^0 = \{\epsilon\}$ and $U^n = U^{n-1}U$ for $n \geq 1$.

For a (partial) function $f : A \longrightarrow B$, $dom f$ denotes the domain of f , i.e. $dom f = \{a \in A | f(a) \text{ is defined}\}$. If $U \subseteq A$ then $f(U) = \{f(a) | a \in U \cap dom f\}$.

For a finite set A , $card(A)$ will denote the number of elements of A .

2 Stream X-machines

This section gives a brief overview of the stream X-machine model. For more details see [16] or [18].

Definition 2.1 *A stream X-machine P is a tuple $(\Sigma, \Gamma, Q, M, \Phi, F, I, m_0)$, as follows:*

- Σ and Γ are finite sets called the input alphabet and the output alphabet, respectively.
- M is a (possibly) infinite set called memory.
- Q is the finite set of states.
- Φ is the type of P , a finite set of basic (partial) processing functions that the machine can use, of the form

$$\phi : M \times \Sigma \longrightarrow \Gamma \times M.$$

- F is the next state (partial) function,

$$F : Q \times \Phi \longrightarrow 2^Q.$$

F is usually described by a state-transition diagram.

- I is the set of initial states

$$I \subseteq Q.$$

- m_0 is the initial memory value

$$m_0 \in M.$$

Thus, stream X-machines are X-machines for which the basic processing functions have the form $\phi : M \times \Sigma \longrightarrow \Gamma \times M$, i.e. each such function will read an input symbol, discard it and produce an output symbol while (possibly) changing the value of the memory.

It is sometimes helpful to think of an X-machine as an automaton with the arcs labelled by functions from the type Φ . The automaton $A = (\Phi, Q, F, I)$ over the alphabet Φ is called *the associated automaton* of P .

Definition 2.2 *If $q, q' \in Q$, $\phi \in \Phi$ and $q' \in F(q, \phi)$ we say that ϕ is an arc from q to q' and write $\phi : q \rightarrow q'$. If $q, q' \in Q$ are such that $\exists q_1, \dots, q_{n+1} \in Q$ with $q_1 = q$ and $q_{n+1} = q'$ so that $\phi_1 : q_1 \rightarrow q_2, \phi_2 : q_2 \rightarrow q_3, \dots, \phi_n : q_n \rightarrow q_{n+1}$ we say that we have a path $p = \phi_1 \cdots \phi_n$ from q to q' and write $p : q \rightarrow q'$. Each path $p = \phi_1 \cdots \phi_n$ gives rise to a (partial) function (the path function) $[p] : M \times \Sigma^* \rightarrow \Gamma^* \times M$ defined by*

$[p](m, s) = (g, m')$ if $\exists n \geq 0, \sigma_1, \dots, \sigma_n \in \Sigma, \gamma_1, \dots, \gamma_n \in \Gamma, m_1, \dots, m_{n+1} \in M$ with $m_1 = m, m_{n+1} = m', s = \sigma_1 \cdots \sigma_n$ and $g = \gamma_1 \cdots \gamma_n$ so that $\phi_i(m_i, \sigma_i) = (\gamma_i, m_{i+1}) \forall 1 \leq i \leq n$. The function corresponding to the empty path ϵ is defined by $[\epsilon](m, \epsilon) = (\epsilon, m), m \in M$.

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 is defined next.

Definition 2.3 *A stream X-machine P is called deterministic if the following are true.*

- *P has only one initial state, i.e.*

$$I = \{q_0\}$$

- *The next state function of P maps each pair state/processing function into at most one single state, i.e. F is a (partial) function*

$$F : Q \times \Sigma \rightarrow Q$$

- *If $\phi_1 : q \rightarrow q_1$ and $\phi_2 : q \rightarrow q_2$ are distinct arcs emerging from the same state q , then*

$$\text{dom } \phi_1 \cap \text{dom } \phi_2 = \emptyset.$$

In what follows we will only refer to deterministic stream X-machines.

A machine computation takes the form of a traversal path in the state space and the application, in turn, of the path labels (which represent basic processing functions). The *behaviour* of the machine identifies the correspondence between the initial state, memory value and input sequence and the final state, memory value and output sequence for all paths. The correspondence between the input sequence applied to the initial state and memory value of the machine and the output sequence produced gives rise to the *function computed* by the machine. These are defined next.

Definition 2.4 The (partial) function $[P] : Q \times M \times \Sigma^* \longrightarrow \Gamma^* \times Q \times M$ defined by

$$[P](q, m, \epsilon) = (g, q', m') \text{ if } \exists \text{ a path } p : q \rightarrow q' \text{ such that } [p](m, s) = (g, m').$$

is called the behaviour of the machine.

Definition 2.5 For $q \in Q$ and $m \in M$ the (partial) function $f_{q,m} : \Sigma^* \longrightarrow \Gamma^*$ defined by:

$$f_{q,m}(s) = g \text{ if } \exists q' \in Q, m' \in M \text{ such that } [P](q, m, s) = (g, q', m')$$

is called the function computed by P in (q, m) . If $q = q_0$ and $m = m_0$ then $f_{q,m}$ is simply called the function computed by P and is denoted by f .

The following definition and theorem provide a characterisation of the functions computed by stream X-machines.

Definition 2.6 Let $f : \Sigma^* \longrightarrow \Gamma^*$ be a (partial) function. Then f is called a (partial) stream function if the following are true:

- $|f(s)| = |s| \forall s \in \text{dom } f$
- if $sx \in \text{dom } f$ then $s \in \text{dom } f$ and $\exists y \in \Gamma^*$ such that $f(sx) = f(s)y \forall s, x \in \Sigma^*$.

Theorem 2.1 f is a (partial) stream function if and only if there exists a stream X-machine P such that P computes f .

Proof. see [16].

Definition 2.7 Let $f : \Sigma^* \longrightarrow \Gamma^*$ be a (partial) stream function and let $x \in \text{dom } f$. Then we define a (partial) function $f_x : \Sigma^* \longrightarrow \Gamma^*$ with $\text{dom } f = \{s \in \Sigma^* \mid xs \in \text{dom } f\}$ by:

$$f_x(s) = y \text{ such that } f(xs) = f(x)y.$$

It can be shown easily that f_x is also a (partial) stream function, see [16].

In terms of stream X-machines, f_x admits the following straightforward interpretation. If f is the function computed by the stream X-machine P having initial state q_0 and initial memory value m_0 and $x \in \text{dom } f$ then let $[P](q_0, m_0, s) = (g, q, m)$, where $q \in Q, m \in M$ and $g \in \Gamma^*$. Then f_x is the (partial) function computed by P in (q, m) , i.e. $f_x = f_{q,m}$.

Example 2.1. Let P be a stream X-machine with $\Sigma = \{a, b, c\}, \Gamma = \{A, B, C\}, M = \{0, 1\}, m_0 = 0$ and the associated automaton represented in Fig. 1 $\phi_1, \phi_2 : M \times \Sigma \longrightarrow \Gamma \times M$ are partial functions defined by:

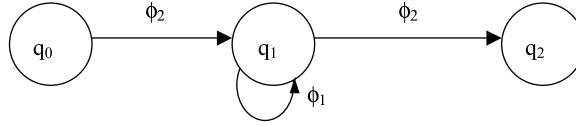


Fig. 1. The associated automaton of P

$$\begin{aligned}\phi_1(m, a) &= (A, 1), \quad m \in \{0, 1\} \\ \phi_2(0, b) &= (B, 0) \\ \phi_2(1, c) &= (C, 0)\end{aligned}$$

The function computed by P , $f : \Sigma^* \longrightarrow \Gamma^*$, is defined by:

$$\begin{aligned}f(\epsilon) &= \epsilon \\ f(ba^j) &= BA^j, \quad j \geq 0 \\ f(bb) &= BB \\ f(ba^j c) &= BA^j C, \quad j \geq 1\end{aligned}$$

If $x = ba$, $q = q_1$ and $m = 1$ then $f_x = f_{q,m} = h$, where $h : \Sigma^* \longrightarrow \Gamma^*$ is defined by:

$$\begin{aligned}h(a^j) &= A^j, \quad j \geq 0 \\ h(a^j c) &= A^j C, \quad j \geq 0\end{aligned}$$

3 Stream X-machine Testing

This section presents briefly the theoretical results that are the basis of the *stream X-machine testing method* [18]. The method uses a stream X-machine specification to generate a set of input sequences that detects all faults of the implementation – i.e. if the application of all the sequences in the test set to the implementation produce the specified outputs then the implementation will produce the specified output for any input sequence – under the following two assumptions. Firstly, the method assumes that the implementation can be modelled as a stream X-machine with the same set of basic processing functions Φ as the specification – the implications of this assumption are outlined in the introduction of the paper and are analysed in detail in [18] [16] and [19]. Furthermore, the method assumes that the set of processing functions meets two “design for testing conditions”, completeness and output-distinguishability, as defined next.

Definition 3.1 Φ is called output-distinguishable if the following is true.

$\forall \phi_1, \phi_2 \in \Phi$, if $\exists m, m_1, m_2 \in M, \sigma \in \Sigma, \gamma \in \Gamma$ such that $\phi_1(m, \sigma) = (\gamma, m_1)$ and $\phi_2(m, \sigma) = (\gamma, m_2)$ then $\phi_1 = \phi_2$.

What this is saying is that we must be able to distinguish between any two different processing functions by examining outputs. In example 2.1. Φ is output-distinguishable.

Definition 3.2 Φ is called complete if the following is true.

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

What this is saying is that any processing function can exercise all memory values using appropriate inputs. In example 2.1., Φ is complete.

These conditions can be imposed on a stream X-machine specification by a suitable enrichment of the input and output alphabets, simple algorithms are available [16], [19], [15]. The extra input or output symbols can be removed after testing has been completed.

The method reduces testing that the two machines $P_1 = (\Sigma, \Gamma, Q_1, M_1, \Phi, F_1, q_{01}, m_0)$ and $P_2 = (\Sigma, \Gamma, Q_2, M_2, \Phi, F_2, q_{02}, m_0)$ – i.e. one representing the specification, the other the implementation – compute the same function to testing that their associated automata accept the same language and generates test sets that ensure this using only the X-machine specification and an estimation of the number of states of the implementation.

Before we go any further, we present briefly the finite state machine model and a few related results. We only present the concepts and results that are absolutely necessary for the construction of our test set and assume that the reader is familiar to basic finite state machine theory, for further details see for example [6] or [9].

In what follows, a *finite state machine (automaton)* will represent a tuple $A = (\Sigma, Q, F, q_0)$, where:

- Σ is a finite set called the *input alphabet*.
- Q is the finite *set of states*.
- F is the (*partial*) *next-state function*, $F : Q \times \Sigma \longrightarrow Q$, usually described as a transition diagram. The machine is assumed to be deterministic, i.e. for each state and input value there is at most one next state.
- $q_0 \in Q$ is the *initial state*.

Given a state $q \in Q$, the *language accepted by A in q*, L_q , is defined by:

$$L_q = \{s \in \Sigma^* | \exists q' \in Q \text{ such that } s : q \rightarrow q' \text{ is a path in } A\}.$$

The language accepted by A in q_0 is simply called the *language accepted by A*.

Given a set $X \subseteq \Sigma^*$, two states $q, q' \in Q$ are called *X-equivalent* if $L_q \cap X = L_{q'} \cap X$. If q and q' are not *X-equivalent* then they are called *X-distinguishable*. Two automata A and A' are called *X-equivalent* or *X-distinguishable* if their initial states are *X-equivalent* or *X-distinguishable*, respectively. Clearly, A and A' accept the same language if and only if A and A' are Σ^* -equivalent.

Two automata A and A' are said to be *isomorphic* if they are identical up to a renaming of the state set.

An automaton is said to be *minimal* if $\forall q \in Q \exists s \in \Sigma^*$ such that $s : q_0 \rightarrow q$ is a path in A and $\forall q_1, q_2 \in Q$ if $q_1 \neq q_2$ then q_1, q_2 are Σ^* -distinguishable. It has been shown that for any automaton A' there exists a minimal automaton A that accepts the same language as A' . A is called the *minimal automaton of A'*.

Standard techniques for deriving the minimal automaton are available, see [6], [9].

Given a minimal automaton A , a set of input sequences $W \subseteq \Sigma^*$ is called a *characterisation set* of A if any two distinct states of A are W -distinguishable. A set of input sequences $S \subseteq \Sigma^*$ is called a *state cover* of A if $\forall q \in Q \exists s \in S$ such that $s : q_0 \rightarrow q$ is a path in A . Note that the minimality of A ensures the existence of a characterisation set and of a state cover. A state cover and a characterisation set of the automaton represented in Fig. 1 are $S = \{\epsilon, \phi_2, \phi_2\phi_2\}$ and $W = \{\phi_1, \phi_2\}$, respectively.

We can now give the formal definition of a test set.

Definition 3.3 *Let $P_1 = (\Sigma, \Gamma, Q_1, M_1, \Phi, F_1, q_{01}, m_0)$ and $P_2 = (\Sigma, \Gamma, Q_2, M_2, \Phi, F_2, q_{02}, m_0)$ be two stream X -machines, A_1 and A_2 their associated automata and f_1 and f_2 , respectively, the functions they compute. Let also k be a positive integer.*

- *A set $Y \subseteq \Sigma^*$ is called a test set of P_1 and P_2 if the following is true.
If $f_1(y) = f_2(y) \forall y \in Y$, then A_1 and A_2 accept the same language.*
- *A set $Y \subseteq \Sigma^*$ is called a k -test set of P_1 if $\forall P_2$ as above with $\text{card}(Q_2) - \text{card}(Q_1) \leq k$, Y is a test set of P_1 and P_2 .*

Note that if A_1 and A_2 accept the same language then f_1 and f_2 are identical – this is easy to prove, see [16] – so a test set ensures that P_1 and P_2 compute identical functions.

The test set is generated in two steps. First, a set of sequences of ϕ 's is constructed from the associated automaton of the specification. Then this is translated into sequences of inputs using a so-called *test function*, as defined next.

Definition 3.4 *Let $P = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ be a stream X -machine with Φ complete, $q \in Q$ and $m \in M$. Then a function $t : \Phi^* \rightarrow \Sigma^*$ is called a test function of P w.r.t. (q, m) if:*

- $t(\epsilon) = \epsilon$
- $\forall \phi_1 \dots \phi_n \in \Phi$ with $n > 0$, let $p_i = \phi_1 \dots \phi_i$, $1 \leq i \leq n$. Then $t(p_n)$ meets the following requirements.
 - If p_n is a path in P starting in q_0 then $t(p_n) = \sigma_1 \dots \sigma_n$, where $\sigma_1, \dots, \sigma_n \in \Sigma$ such that $(m_0, \sigma_1 \dots \sigma_n) \in \text{dom}[p_n]$.
 - Otherwise, $t(p_n) = \sigma_1 \dots \sigma_{k+1}$, where $\sigma_1, \dots, \sigma_{k+1} \in \Sigma$ such that $(m_0, \sigma_1 \dots \sigma_{k+1}) \in \text{dom}[p_{k+1}]$, where k is the largest number $0 \leq k \leq n - 1$ for which p_k is a path in P starting in q_0 .

Note that since Φ is complete there exists at least an input sequence $t(p_n)$ that meets the above requirements. Also note that a test function w.r.t. (q, m) is not uniquely determined, in general many different possible test functions exist. If $q = q_0$ and $m = m_0$ then t is simply called a *test function of P* .

The following values illustrate the construction of a test function for the X-machine in example 2.1.: $t(\epsilon) = \epsilon$, $t(\phi_2) = b$, $t(\phi_2\phi_1) = ba$, $t(\phi_2\phi_1\phi_2) = bac$, $t(\phi_2\phi_1\phi_2\phi_1) = baca$, $t(\phi_2\phi_1\phi_2\phi_1\phi_1) = baca$, $t(\phi_2\phi_1\phi_2\phi_1\phi_1\phi_2) = baca$, etc.

The following theorem is the theoretical basis of the stream X-machine testing method.

Theorem 3.1 *Let $P_1 = (\Sigma, \Gamma, Q_1, M_1, \Phi, F_1, q_{01}, m_0)$ and $P_2 = (\Sigma, \Gamma, Q_2, M_2, \Phi, F_2, q_{02}, m_0)$ be two stream X-machines with Φ output-distinguishable and A_1 the associated automaton of P_1 such that A_1 is minimal. Let S be a state cover of A_1 , W a characterisation set of A_1 and $t : \Phi^* \rightarrow \Sigma^*$ a test function of P_1 . If $\text{card}(Q_2) - \text{card}(Q_1) \leq k$, where k is a positive integer, then $Y = t(X)$ is a test set of P_1 and P_2 , where $X = S(\Phi^{k+1} \cup \Phi^k \cup \dots \cup \Phi \cup \{\epsilon\})W$.*

Proof. See [18] or [16].

Corollary 3.1 *Given the notation and the assumptions of the above theorem, Y is a k -test of P_1 .*

4 Simple Covering

In broad terms, we say that a machine P' is a *simple covering* of another machine P when the new machine P' does everything that the original did plus something extra. So

- The input and output alphabets of P' will be extensions of the input and output alphabets of P , respectively.
- The function computed by P' will be an extension of the function computed by P .

Before we define this formally, we establish some notation.

Definition 4.1 *Let $\Sigma \subseteq \Sigma'$ be two alphabets. Then we denote by $\text{Filter}_\Sigma : \Sigma'^* \rightarrow \Sigma^*$ the function that extracts a subsequence containing all elements in Σ from a sequence of elements in Σ' , i.e.*

$$\text{Filter}_\Sigma(\epsilon) = \epsilon$$

$$\text{Filter}_\Sigma(s\sigma) = \text{Filter}_\Sigma(s)\sigma, \quad \forall s \in \Sigma'^*, \sigma \in \Sigma$$

$$\text{Filter}_\Sigma(s\sigma) = \text{Filter}_\Sigma(s), \quad \forall s \in \Sigma'^*, \sigma \in \Sigma' - \Sigma.$$

Definition 4.2 *Let $P = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ and $P' = (\Sigma', \Gamma', Q', M', \Phi', F', q'_0, m'_0)$ be two stream X-machines such that $\Sigma \subseteq \Sigma'$. Given $q \in Q$, $m \in M$, $q' \in Q'$, $m' \in M'$, we say that (q, m) is Σ -equivalent to (q', m') and write*

$(q, m) \sim_{\Sigma} (q', m')$ if $\forall s \in \Sigma^* f_{q,m}(s) = f_{q',m'}(s)$. If $q = q_0, m = m_0, q' = q'_0$ and $m' = m'_0$ then we say that P and P' are Σ -equivalent.

We can now give a formal definition of the simple covering.

Definition 4.3 Let $P = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ and $P' = (\Sigma', \Gamma', Q', M', \Phi', F', q'_0, m'_0)$ be two stream X -machines such that $\Sigma \subseteq \Sigma'$ and $\Gamma \subseteq \Gamma'$. Then we say that P' is a simple covering of P if the following is true $\forall x' \in \Sigma'^*$ and $x = \text{Filter}_{\Sigma}(x')$.

If $[P](q_0, m_0, x) = (g, q, m)$ and $[P'](q'_0, m'_0, x') = (g', q', m')$, with $q \in Q, m \in M, g \in \Gamma, q' \in Q', m' \in M', g' \in \Gamma'$, then $(q, m) \sim_{\Sigma} (q', m')$.

Thus when x' and $\text{Filter}_{\Sigma}(x')$ are applied to P' and P , respectively, they produce state/memory pairs that are Σ -equivalent. In particular (i.e. for $x' = \epsilon$), P' and P are Σ -equivalent.

The following proposition provides a characterisation of the simple covering in terms of the functions computed by the two machines.

Proposition 4.1 Let P and P' be two stream X -machines as above and $f : \Sigma^* \rightarrow \Gamma^*$ and $f' : \Sigma'^* \rightarrow \Gamma'^*$ the (partial) functions computed by them. Then P' is a simple covering of P if and only if the following is true $\forall x' \in \Sigma'^*$ and $x = \text{Filter}_{\Sigma}(x')$.

If $x' \in \text{dom } f'$ and $x \in \text{dom } f$ then $f'_{x'}(s) = f_x(s) \forall s \in \Sigma^*$.

Proof. Follows from definitions 2.5 and 4.3.

Corollary 4.1 If P' is a simple covering of P then $f'(s) = f(s) \forall s \in \Sigma^*$.

5 Construction of Simple Coverings

Once the transformation has been defined in terms of the input-output behaviour of the two machines, we turn our attention to the construction of the new machine P' . The internal structure of P' may differ from that of P in various ways. Obviously, we will be interested in methods of constructing P' starting from the architecture of P , so that the internal structure of P is preserved in some sense and the construction of the new model will not start anew with each modification.

In general, given a stream X -machine P , it is possible to construct more than one simple covering, so our construction will generate a set of machines, we will call this $R(P)$. The process will involve an expansion of the internal structure of the machine, i.e. of the state and memory sets. Each processing function $\phi \in \Phi$ will be replaced by an *extended function* ϕ_e that operates on the expanded memory. The transitions that are triggered by inputs that were not in

the original alphabet Σ will be performed by some new processing functions, these will be called the *auxiliary functions*.

So let $P = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ be the original machine and let Σ', Γ' and M' the input alphabet, output alphabet and the memory set of the new machine such that $\Sigma \subseteq \Sigma', \Gamma \subseteq \Gamma'$ and there exists $H = \{H_m\}_{m \in M}$ a partition of M' indexed by M . Let also $\Sigma_0 = \Sigma' - \Sigma$. Then we have the following definitions.

Definition 5.1 Let $\phi : M \times \Sigma \longrightarrow \Gamma \times M$ be a (partial) processing function of P . Then a (partial) function $\phi_e : M' \times \Sigma \longrightarrow \Gamma \times M'$ is called an extension of ϕ if $\forall \sigma \in \Sigma, \gamma \in \Gamma, m, n \in M, m' \in H_m,$

$\exists n' \in H_n$ such that $\phi_e(m', \sigma) = (\gamma, n')$ if and only if $\phi(m, \sigma) = (\gamma, n)$.

Definition 5.2 A (partial) function $\psi : M' \times \Sigma_0 \longrightarrow \Gamma' \times M'$ is called an auxiliary function if $\forall m', n' \in M', \sigma \in \Sigma_0, \gamma \in \Gamma',$

if $\psi(m', \sigma) = (n', \gamma)$ then $\exists m \in M$ such that $m', n' \in H_m$.

That is, an auxiliary function will operate only on inputs that are not in the original machine and any new memory value produced will be in the same subset H_m as the original memory value.

Definition 5.3 Let P be a stream X-machine and let $\Sigma', \Gamma', M', \Sigma_0$ and $H = \{H_m\}_{m \in M}$ as above. Then $R(P)$ is the set of all stream X-machines $P' = (\Sigma', \Gamma', Q', M', \Phi', F', q_0, m'_0)$ for which the following are true.

- The state set includes the state set of P , i.e. $Q \subseteq Q'$.
- The initial state is q_0 , the initial state of P .
- The initial memory value $m'_0 \in H_{m_0}$, where m_0 is the initial memory value of P .
- The type is $\Phi' = \Phi_e \cup \Psi$, where $\Phi_e = \{\phi_e | \phi \in \Phi\}$ is a set of extensions of the original processing functions and Ψ is a set of auxiliary functions. Φ_e will be called the extended type and Ψ the auxiliary type.
- There is a partition of Q' indexed by Q , $E = \{E_q\}_{q \in Q}$, with $q \in E_q \forall q \in Q$ such that F' meets the following conditions.
 - $\forall q' \in Q', \phi \in \Phi, F'(q', \phi_e) = F(q, \phi)$, where $q \in Q$ is such that $q' \in E_q$.
 - $\forall q' \in Q', \psi \in \Psi$ if $F'(q', \psi) = \theta'$ then $\theta' \in E_q$, where $q \in Q$ is such that $q' \in E_q$.

Note that for a given q', q is uniquely determined since $E = \{E_q\}_{q \in Q}$ is a partition of Q .

Thus, the state-transition diagram of the new machine is as though the states in the original machines are replaced by sub-machines that contain only transitions performed by auxiliary functions - these are called the *auxiliary machines* and are defined next. The construction is illustrated in Fig. 2.

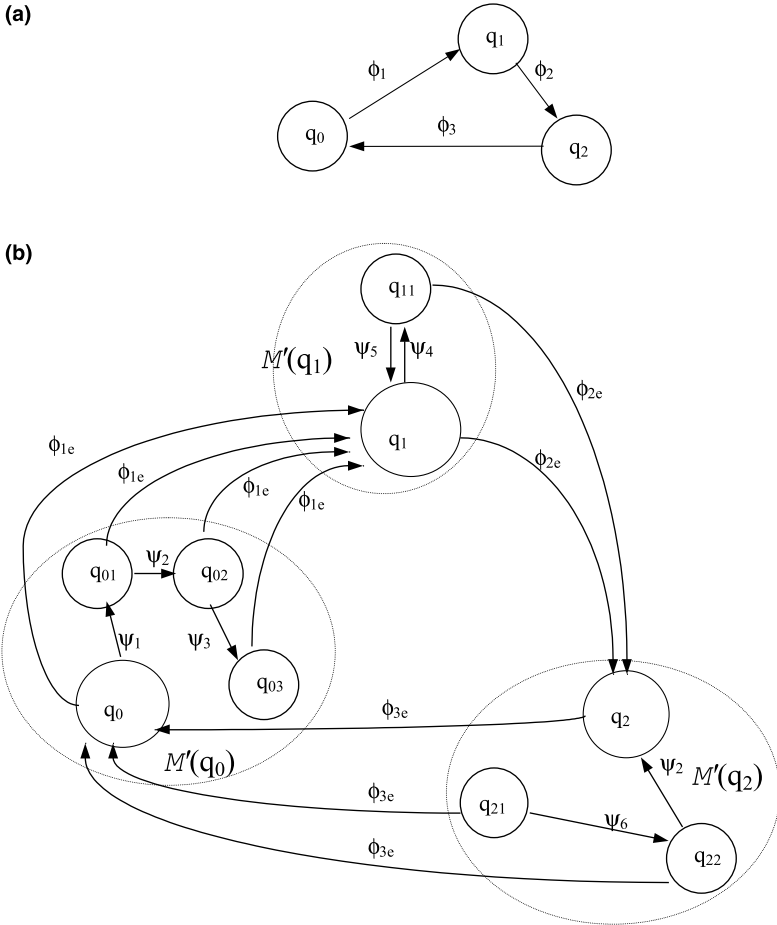


Fig. 2. The associated automata of P (a) and $P' \in R(P)$ (b)

Given Φ and Φ_e as above, we denote by $e : \Phi^* \longrightarrow \Phi_e^*$ the free semigroup isomorphism induced by $e(\phi) = \phi_e \forall \phi \in \Phi$.

Definition 5.4 Given the construction above and a state $q \in Q$, the auxiliary machine in q , denoted $P'(q)$, is a stream X -machine $(\Sigma_0, \Gamma', E_q, M', \Psi, F_q, q, m'_q)$ with state set E_q , initial state q , type Ψ and next-state function F_q defined by:

$$F_q(q', \psi) = F'(q', \psi) \forall q' \in E_q, \psi \in \Phi.$$

The initial memory value m'_q is not relevant, so it can be arbitrarily chosen.

Example 5.1. We illustrate the above construction for the stream X -machine P of example 2.1.. We take $\Sigma' = \{a, b, c, x, y, z\}$, $\Gamma' = \{A, B, C, X, Y, Z\}$, $M' = M \times \{0, 1\} = \{0, 1\} \times \{0, 1\}$, $m_0 = (0, 0)$, $H = \{H_0, H_1\}$, where $H_0 = \{0\} \times \{0, 1\}$ and $H_1 = \{1\} \times \{0, 1\}$. The associated automata of P' and of the auxiliary machines are represented in Fig. 3.

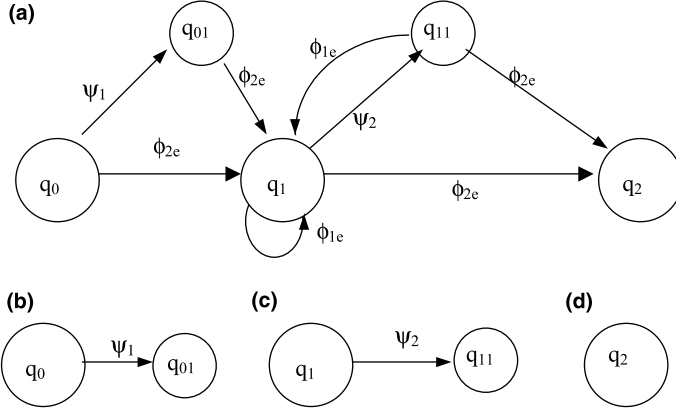


Fig. 3. The associated automata of P' (a), $P'(q_0)$ (b), $P'(q_1)$ (c) and $P'(q_2)$ (d)

$$\phi_{1e}((m, m'), a) = (A, (1, m')), \quad m \in \{0, 1\}, \quad m' \in \{0, 1\}$$

$$\phi_{2e}((0, m'), b) = (B, (0, m')), \quad m' \in \{0, 1\}$$

$$\phi_{2e}((1, m'), c) = (C, (0, m')), \quad m' \in \{0, 1\}$$

$$\psi_1((m, m'), x) = (X, (m, 1)), \quad m \in \{0, 1\}, \quad m' \in \{0, 1\}$$

$$\psi_2((m, 0), y) = (Y, (m, 0)), \quad m \in \{0, 1\}$$

$$\psi_2((m, 1), z) = (Z, (m, 0)), \quad m \in \{0, 1\}$$

The function computed by P' is $f' : \Sigma'^* \longrightarrow \Gamma'^*$ defined by:

$$f'(\epsilon) = \epsilon$$

$$f'(x) = X$$

$$f'(ba^{k_0}ya^{k_1}y \dots a^{k_n}) = BA^{k_0}YA^{k_1}Y \dots A^{k_n}, \quad n \geq 0, \quad k_0 \geq 0, \quad k_1, \dots, k_{n-1} \geq 1, \quad k_n \geq 0$$

$$f'(xba^{k_0}za^{k_1}z \dots a^{k_n}) = XBA^{k_0}ZA^{k_1}Z \dots A^{k_n}, \quad n \geq 0, \quad k_0 \geq 0, \quad k_1, \dots, k_{n-1} \geq 1, \quad k_n \geq 0$$

$$f'(bb) = BB$$

$$f'(xbb) = XBB$$

$$f'(byb) = BYB$$

$$f'(xbzb) = XBZB$$

$$f'(ba^{k_0}ya^{k_1}y \dots a^{k_n}c) = BA^{k_0}YA^{k_1}Y \dots A^{k_n}C, \quad n \geq 0, \quad k_0 \geq 0, \quad k_1, \dots, k_{n-1} \geq 1, \quad k_n \geq 0 \text{ with } k_0 + \dots + k_n \geq 1$$

$$f'(xba^{k_0}za^{k_1}z \dots a^{k_n}c) = XBA^{k_0}ZA^{k_1}Z \dots A^{k_n}C, \quad n \geq 0, \quad k_0 \geq 0, \quad k_1, \dots, k_{n-1} \geq 1, \quad k_n \geq 0 \text{ with } k_0 + \dots + k_n \geq 1$$

6 Equivalence Between Simple Covering and $R(P)$

This section shows that all elements of $R(P)$ are simple coverings of P (theorem 6.1) and, conversely, for any simple covering of P there exists an element

of $R(P)$ that is functionally equivalent to it (theorem 6.2). A couple of intermediary results are proved first.

Lemma 6.1 *Let P and $P' \in R(P)$ as above and let $q_1, q_2 \in Q$, $m_1, m_2 \in M$, $q'_1 \in E_{q_1}$, $m'_1 \in H_{m_1}$, $m'_2 \in H_{m_2}$, $s \in \Sigma^*$, $g \in \Gamma^*$, $p \in \Phi^*$. Then the following are true.*

- a) $p : q_1 \rightarrow q_2$ is a path in P if and only if $e(p) : q'_1 \rightarrow q_2$ is a path in P'
- b) $[P](q_1, m_1, s) = (g, q_2, m_2)$ if and only if $[P'](q'_1, m'_1, s) = (g, q_2, m'_2)$

Proof. From definition 5.3 it follows that $\forall \phi \in \Phi$, $\phi : q_1 \rightarrow q_2$ is an arc in P if and only if $e(\phi) : q'_1 \rightarrow q_2$ is an arc in P' . Then a) follows by induction on the length of p . b) follows from a).

Lemma 6.2 *Let P , P' as above and let $q_1, q_2 \in Q$, $m_1, m_2 \in M$, $q'_1 \in E_{q_1}$, $q'_2 \in E_{q_2}$, $m'_1 \in H_{m_1}$, $m'_2 \in H_{m_2}$, $s' \in \Sigma'^*$, $s = \text{Filter}_\Sigma(s')$, $p' \in \Phi'^*$, $p = e^{-1}(\text{Filter}_{\Phi_e}(p'))$. Then the following are true.*

- a) If $p' : q'_1 \rightarrow q'_2$ is a path in P' then $p : q_1 \rightarrow q_2$ is a path in P .
- b) If $\exists g' \in \Gamma'^*$ such that $[P'](q'_1, m'_1, s') = (g', q'_2, m'_2)$ then $\exists g \in \Gamma^*$ such that $[P](q_1, m_1, s) = (g, q_2, m_2)$.

Proof. Since $p = e^{-1}(\text{Filter}_{\Phi_e}(p'))$, there exist $k \geq 0$, $\phi_1, \dots, \phi_k \in \Phi$ and $p'_0, \dots, p'_k \in \Psi^*$ such that $p = \phi_1 \cdots \phi_k$ and $p' = p'_0 e(\phi_1) p'_1 \cdots p'_{k-1} e(\phi_k) p'_k$. Then a) follows from definition 5.3 by induction on k . b) follows from a).

Theorem 6.1 *Let P be a stream X -machine and $P' \in R(P)$. Then P' is a simple covering of P .*

Proof. Follows from lemmas 6.1 b) and 6.2 b).

Theorem 6.2 *Let $P = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ be a stream X -machine, P'' a simple covering of P and $f'' : \Sigma'^* \rightarrow \Gamma'^*$ the (partial) function computed by P'' . Then there exists $P' \in R(P)$ such that P' computes f'' .*

Proof. $P' = (\Sigma', \Gamma', Q', M', \Phi', F', q_0, m'_0)$ is constructed as in definition 5.3 with the following additional properties:

- $Q' = Q$, $M' = M \times \Sigma'$, $m'_0 = (m_0, \epsilon)$.
- $\forall \phi \in \Phi$, ϕ_e is defined by: $\phi_e((m, x), \sigma) = ((n, x\sigma), \gamma)$, if $\phi(m, \sigma) = (n, \gamma)$, where $m, n \in M$, $x \in \Sigma'^*$, $\sigma \in \Sigma$, $\gamma \in \Gamma$.

- $\Psi = \{\psi_q\}_{q \in Q}$, where $\forall q \in Q$, ψ_q is defined by: $\psi_q((m, x), \sigma) = ((m, x\sigma), \gamma)$, where $m \in M$, $x \in \Sigma'^*$, $\sigma \in \Sigma' - \Sigma$, and $\gamma = f_x''(\sigma)$.

We also define $H = \{H_m\}_{m \in M}$ by $H_m = \{m\} \times \Sigma'^* \forall m \in M$. Then $P' \in R(P)$. Note that, since $Q' = Q$, we have $E_q = \{q\}$ and $F_q(q, \psi_q) = q \forall q \in Q$.

We have to prove now that $f' = f''$, where $f' : \Sigma'^* \rightarrow \Gamma'^*$ is the (partial) function computed by P' . We prove that $f'(x) = f''(x)$, $x \in \Sigma'^*$, by induction on the length of x . $f'(\epsilon) = f''(\epsilon) = \epsilon$. We assume that $f'(x) = f''(x)$ and we have to prove that $f'(x\sigma) = f''(x\sigma)$, where $\sigma \in \Sigma'$. Clearly, if $x \notin \text{dom } f'$ and $x \notin \text{dom } f''$ then $x\sigma \notin \text{dom } f'$ and $x\sigma \notin \text{dom } f''$. Otherwise, since $f'(x\sigma) = f'(x)f'_x(\sigma)$ and $f''(x\sigma) = f''(x)f''_x(\sigma)$, it follows that $f''(x\sigma) = f'(x\sigma)$ if and only if $f''_x(\sigma) = f'_x(\sigma)$. Now, we have two cases:

- $\sigma \in \Sigma$. Since P' and P'' are both simple coverings of P , from proposition 4.1 it follows that $f'_x(\sigma) = f_s(\sigma)$ and $f''_x(\sigma) = f_s(\sigma)$, where f is the function computed by P and $s = \text{Filter}_\Sigma(x)$. Hence $f'_x(\sigma) = f''_x(\sigma)$.
- $\sigma \in \Sigma' - \Sigma$. Since $x \in \text{dom } f'$, by induction on the length of x it follows that $\exists q \in Q, m \in M, \gamma \in \Gamma'^*$ such that $[P'](q_0, (m_0, \epsilon), x) = (y, q, (m, x))$, and thus $f'_x = f'_{q, (m, x)}$. Now, from the above definition of ψ_q and from definition 2.7 it follows that $f'_{q, (m, x)}(\sigma) = f''_x(\sigma)$. Hence $f'_x(\sigma) = f''_x(\sigma)$.

7 Testing Simple Coverings – Theory

Once we know how to construct simple coverings, we can turn our attention to testing. The problem we seek to address is, given an initial stream X-machine specification, P_1 , and $P'_1 \in R(P_1)$, the final specification, how can we test P'_1 ? Obviously P'_1 is itself a stream X-machine, so the method described in section 3 can be applied. However, this is not always convenient, for large scale systems the state set can be extremely large and this will result in an unmanageable test set. Furthermore, since P'_1 is constructed through a process of refinement, one would hope that the test set of the original machine P_1 could be reused in the construction of the new test set and the testing process would be distributed into smaller chunks, thus diminishing the effort devoted to the testing of the final specification. This issue is addressed in this section, where a method for testing simple coverings is developed. Similarly to the stream X-machine testing method, this new method guarantees the detection of all faults of the specification provided that the system is made of fault-free components (i.e. the basic processing functions Φ' are implemented correctly) and some design for testing requirements (completeness and output-distinguishability) are met. Furthermore, we will assume that the implementation can be modelled as a stream X-machine P'_2 and there exists another stream X-machine P_2 having the same components (i.e. the same basic processing functions Φ) as P_1 , so that $P'_2 \in R(P_2)$, see Fig. 4. The implications of this extra assumption will be discussed in the following section, that deals with the practical application of the method.

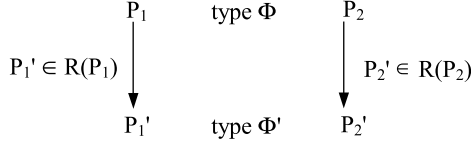


Fig. 4. Testing strategy

So, let $P_1 = (\Sigma, \Gamma, Q_1, M_1, \Phi, F_1, q_{01}, m_0)$, $P_2 = (\Sigma, \Gamma, Q_2, M_2, \Phi, F_2, q_{02}, m_0)$, $P_1' = (\Sigma', \Gamma', Q_1', M', \Phi', F_1', q_{01}, m_0')$ and $P_2' = (\Sigma', \Gamma', Q_2', M', \Phi', F_2', q_{02}, m_0')$ be four stream X-machines such that $P_1' \in R(P_1)$ and $P_2' \in R(P_2)$ and let $A_1 = (\Phi, Q_1, F_1, q_{01})$, $A_2 = (\Phi, Q_2, F_2, q_{02})$, $A_1' = (\Phi', Q_1', F_1', q_{01})$ and $A_2' = (\Phi', Q_2', F_2', q_{02})$ be their associated automata. Let also f_1, f_2, f_1' and f_2' be the functions computed by P_1, P_2, P_1' and P_2' , respectively.

Let $\Phi' = \Phi_e \cup \Psi$, where Φ_e is the extended type of P_1' and P_2' and Ψ the auxiliary type of P_1' and P_2' – note that P_1' and P_2' have the same extended type and the same auxiliary type, this is because P_1' and P_2' have the same type and $\Phi_e \cap \Psi = \emptyset$.

For $q \in Q_1$, let $P_1'(q) = (\Sigma_0, \Gamma', E_q^1, M', \Psi, F_q^1, q, m')$ be the auxiliary machine in q used in the construction of P_1' and $A_1'(q) = (\Psi, E_q^1, F_q^1, q)$ its associated automaton. Similarly, for $\theta \in Q_2$, let $P_2'(\theta) = (\Sigma_0, \Gamma', E_\theta^2, M', \Psi, F_\theta^2, \theta, m')$ be the auxiliary machine in θ used in construction of P_2' and $A_2'(\theta) = (\Psi, E_\theta^2, F_\theta^2, \theta)$ its associated automaton. Obviously, $Q_1' = \bigcup_{q \in Q_1} E_q^1$ and $Q_2' = \bigcup_{\theta \in Q_2} E_\theta^2$.

Our aim is to construct Y' , a test set of P_1' and P_2' , and to reuse Y , a test set of P_1 and P_2 , in the construction of Y' . Before we can do this, we prove a few preparatory results.

Lemma 7.1 *If $c : Q_2 \longrightarrow Q_1$ is a surjective function and $d_\theta : E_\theta^2 \longrightarrow E_{c(\theta)}^1$, $\theta \in Q_2$, a set of surjective functions such that:*

$$\begin{aligned}
 c(q_{02}) &= q_{01}, \\
 c(F_2(\theta, \phi)) &= F_1(c(\theta), \phi) \quad \forall \theta \in Q_2, \phi \in \Phi, \\
 d_\theta(\theta) &= c(\theta) \quad \forall \theta \in Q_2 \text{ and} \\
 d_\theta(F_2'(\theta', \psi)) &= F_1'(d_\theta(\theta'), \psi) \quad \forall \theta \in Q_2, \theta' \in E_\theta^2, \psi \in \Psi
 \end{aligned}$$

then there exists a surjective function $d : Q_2' \longrightarrow Q_1'$ such that

$$\begin{aligned}
 d(q_{02}) &= q_{01} \text{ and} \\
 d(F_2'(\theta', \phi')) &= F_1'(d(\theta'), \phi') \quad \forall \theta' \in Q_2', \phi' \in \Phi'.
 \end{aligned}$$

Proof. d is defined by $d = \bigcup_{\theta \in Q_2} d_\theta$.

Aside: The proofs of the following lemmas and theorems will use the following results from finite state machine theory: Let $A = (\Sigma, Q, F, q_0)$ and $A' = (\Sigma, Q', F', q_0')$ be two finite state machines over the same input alphabet.

1. If there exists a surjective function $d : Q \longrightarrow Q'$ such that

$$d(q_0) = q'_0 \text{ and} \\ d(F(q, \sigma)) = F'(d(q), \sigma) \forall q \in Q, \sigma \in \Sigma.$$

then A and A' accept the same language.

2. If A' is the minimal automaton of A then there exists a surjective function $d : Q \longrightarrow Q'$ such that

$$d(q_0) = q'_0 \text{ and} \\ d(F(q, \sigma)) = F'(d(q), \sigma) \forall q \in Q, \sigma \in \Sigma.$$

Proof.

1. By induction on the length of $s \in \Sigma^*$ it follows that $s : q_0 \rightarrow q$ is a path of A if and only if $s : q'_0 \rightarrow d(q)$ is a path of A' . Hence A and A' accept the same language.

2. d is defined by

$$d(q) = q' \text{ if } q \text{ and } q' \text{ are } \Sigma^*\text{-equivalent states}$$

For more detailed proofs of these results see [6].

Lemma 7.2 *If $c : Q_2 \longrightarrow Q_1$ is surjective function such that*

$$c(q_{02}) = q_{01}, \\ c(F_2(\theta, \phi)) = F_1(c(\theta), \phi) \forall \theta \in Q_2, \phi \in \Phi \text{ and} \\ A'_1(c(\theta)) \text{ is the minimal automaton of } A'_2(\theta) \forall \theta \in Q_2$$

then A'_1 and A'_2 accept the same language.

Proof. Since $A'_1(c(\theta))$ is the minimal automaton of $A'_2(\theta)$ we can define a surjective function $d_\theta : E_\theta^2 \longrightarrow E_{c(\theta)}^1$ such that $d_\theta(\theta) = c(\theta)$ and $d_\theta(F'_2(\theta', \psi)) = F'_1(d_\theta(\theta'), \psi) \forall \theta' \in E_\theta^2, \psi \in \Psi$. Then, using lemma 7.1, there exists a surjective function $d : Q'_2 \longrightarrow Q'_1$ such that $d(q_{02}) = q_{01}$ and $d(F'_2(\theta', \phi')) = F'_1(d(\theta'), \phi') \forall \theta' \in Q'_2, \phi' \in \Phi'$. Thus A'_1 and A'_2 accept the same language.

Lemma 7.3 *If A_1 is minimal, Φ is complete and output-distinguishable, $Y \subseteq \Sigma^*$ is a test set of P_1 and P_2 and $f'_1(s) = f'_2(s) \forall s \in Y$ then A_1 and A_2 accept the same language.*

Proof. Since $P'_1 \in R(P_1)$ and $P'_2 \in R(P_2)$, we have $f'_1(s) = f_1(s)$ and $f'_2(s) = f_2(s) \forall s \in \Sigma^*$, hence $f_1(s) = f_2(s) \forall s \in Y$. Since Y is a test set of P_1 and P_2 , A_1 and A_2 accept the same language.

We can now generate the test set we are looking for. We assume that A_1 is minimal and $\forall q \in Q_1$ $A'_1(q)$ is minimal. We also assume that Φ and Ψ are both complete and output-distinguishable. First, let us establish some notation.

Let S and W be a state cover and a characterisation set of A_1 , respectively. For $q \in Q_1$ let S'_q and W'_q be a state cover and a characterisation set of $A'_1(q)$, respectively. Let $t : \Phi^* \rightarrow \Sigma^*$ be a test function of P_1 and for $q \in Q_1, m' \in M'$ let $t'_{q,m'} : \Psi^* \rightarrow (\Sigma' - \Sigma)^*$ be a test function of $P'_1(q)$ w.r.t. (q, m') .

Let $n = \text{card}(Q_1)$, $v = \text{card}(Q_2)$, for $q \in Q_1$ let $n'_q = \text{card}(E_q^1)$, for $\theta \in Q_2$ let $v'_\theta = \text{card}(E_\theta^2)$ and let $v' = \max_{\theta \in Q_2} v'_\theta$. Let $k = v - n$ and for $q \in Q_1$ let $k'_q = v' - n'_q$.

Let $X = S(\Phi^{k+1} \cup \Phi^k \cup \dots \cup \Phi \cup \{\epsilon\})W$ and $Y = t(X)$ and for $q \in Q_1, m' \in M'$ let $X'_q = S'_q(\Psi^{k_q+1} \cup \Psi^{k_q} \cup \dots \cup \Psi \cup \{\epsilon\})W'_q$ and $Y'_{q,m'} = t'_{q,m'}(X'_q)$. From theorem 3.1 it follows that Y is a test set of P_1 and P_2 and that $Y'_{q,m'}$ is a test set of $P'_1(q)$ and $P'_2(\theta) \forall \theta \in Q_2$, where the initial memory of $P'_1(q)$ and $P'_2(\theta)$ is m' .

Let $Z = \{p \in S(\Phi^k \cup \Phi^{k-1} \cup \dots \cup \{\epsilon\}) \mid \exists q \in Q_1 \text{ such that } p : q_{01} \rightarrow q \text{ is a path in } P_1\}$ the set of elements of $S(\Phi^k \cup \Phi^{k-1} \cup \dots \cup \{\epsilon\})$ that are also paths of P_1 that start in the initial state q_{01} and let $V = t(Z)$.

Finally, let $U = \bigcup_{y \in V} \bigcup_{q \in Q_1, m' \in M'} \{y\} \otimes Y'_{q,m'}$, where for $y \in V, q \in Q_1$ and $m' \in M'$, $\{y\} \otimes Y'_{q,m'}$ is defined as follows:

- If $\exists g \in \Gamma^*$ such that $[P'_1](q_{01}, m'_0, y) = (g, q, m')$ then $\{y\} \otimes Y'_{q,m'} = \{y\}Y'_{q,m'}$;
- Otherwise, $\{y\} \otimes Y'_{q,m'} = \emptyset$.

In other words, U is obtained by concatenating each sequence $y \in V$ with all sequences of $Y'_{q,m'}$, where $q \in Q_1$ and $m' \in M'$ are such that y takes P'_1 from the initial state q_{01} and initial memory value m'_0 to the state q and memory value m' .

Then $Y' = Y \cup U$ is the test set we are after.

The following examples illustrate the construction of Y' for P and P' in examples 2.1. and 5.1. and $k = 1, k'_{q_0} = k'_{q_1} = 0$ and $k'_{q_2} = 1$.

$$S = \{\epsilon, \phi_2, \phi_2\phi_2\}, \quad W = \{\phi_1, \phi_2\},$$

$$S_0 = \{\epsilon, \psi_1\}, \quad W_0 = \{\psi_1\},$$

$$S_1 = \{\epsilon, \psi_2\}, \quad W_1 = \{\psi_2\},$$

$$S_2 = \{\epsilon\}, \quad W_2 = \{\epsilon\},$$

$$X = \{\epsilon, \phi_2, \phi_2\phi_2\}\{\epsilon, \phi_1, \phi_2, \phi_1\phi_1, \phi_1\phi_2, \phi_2\phi_1, \phi_2\phi_2\}\{\phi_1, \phi_2\},$$

$$X_0 = \{\epsilon, \psi_1\}\{\epsilon, \psi_1, \psi_2\}\{\psi_1\},$$

$$X_1 = \{\epsilon, \psi_2\}\{\epsilon, \psi_1, \psi_2\}\{\psi_2\},$$

$$X_2 = \{\epsilon, \psi_1, \psi_2, \psi_1\psi_1, \psi_1\psi_2, \psi_2\psi_1, \psi_2\psi_2\},$$

$$Z = \{\epsilon, \phi_2, \phi_2\phi_1, \phi_2\phi_2\},$$

$$U = t'_0(X_0) \cup \{b\}t'_1(X_1) \cup \{ba\}t'_1(X_1) \cup \{bb\}t'_2(X_2),$$

$$Y' = t(X) \cup U$$

where t'_0 is a test function of $P'(q_0)$ w.r.t. q_0 and $(0, 0)$, t'_1 is a test function of $P'(q_1)$ w.r.t. q_1 and $(0, 0)$, t''_1 is a test function of $P'(q_1)$ w.r.t. q_1 and $(1, 0)$, t'_2 is a test function of $P'(q_2)$ w.r.t. q_2 and $(0, 0)$ and t is a test function of P .

Before we prove that Y' is a test set of P'_1 and P'_2 , we need the following intermediary result.

Lemma 7.4 $\forall \theta \in Q_2 \exists p \in Z$ such that $p : q_{02} \rightarrow \theta$ is a path in A_2 .

Proof. We define the sets

$Z_j = \{p \in S(\Phi^j \cup \Phi^{j-1} \cup \dots \cup \{\epsilon\}) \mid \exists q \in Q_1 \text{ such that } p : q_{01} \rightarrow q \text{ is a path in } P_1\}$

and we prove by induction on $1 \leq j \leq k$ that Z_j reaches at least $n + j$ distinct states of A_2 . Since S is a state cover of A_1 and A_1 is the minimal automaton of A_2 , $Z_0 = S$ will reach n distinct states of A_2 . Let $1 \leq j \leq k - 1$, and let us assume that Z_j reaches at least $n + j$ distinct states of A_2 . Then either Z_j reaches all the states of A_2 (i.e. $v = n + k$ distinct states) or Z_{j+1} reaches at least one more state than Z_j does – this is because

$Z_{j+1} = \{p \in (Z_j \cup Z_j \Phi) \mid \exists q \in Q_1 \text{ such that } p : q_{01} \rightarrow q \text{ is a path in } P_1\}$.
So, $Z = Z_k$ reaches at least $n + k = v$ states of A_2 .

Theorem 7.1 If A_1 is minimal and $\forall q \in Q_1 A'_1(q)$ is minimal and Φ and Ψ are complete and output-distinguishable then the set $Y' = Y \cup U$ constructed as above is a test set of P'_1 and P'_2 .

Proof. We assume that $f'_1(s) = f'_2(s) \forall s \in Y'$ and we have to prove that A'_1 and A'_2 accept the same language. From lemma 7.3 it follows that A_1 and A_2 accept the same language. Thus A_1 is the minimal automaton of A_2 . Then there exists a surjective function $c : Q_2 \rightarrow Q_1$ such that $c(q_{02}) = q_{01}$ and $c(F_2(\theta, \phi)) = F_1(c(\theta), \phi) \forall \theta \in Q_2, \phi \in \Phi$.

Now, let $\theta \in Q_2$ and let $q = c(\theta)$. We want to prove that $A'_1(q)$ is the minimal automaton of $A'_2(\theta)$. Using lemma 7.4, $\exists p \in Z$ such that $p : q_{02} \rightarrow \theta$ is a path in A_2 . Hence $p : q_{01} \rightarrow q$ is a path in A_1 . From lemma 6.1 a) it follows that $e(p) : q_{02} \rightarrow \theta$ is a path in P'_2 and $e(p) : q_{01} \rightarrow q$ is a path in P'_1 . Thus $\exists m' \in M', g \in \Gamma^*$ such that $[P'_2](q_{02}, m'_0, y) = (g, \theta, m')$ and $[P'_1](q_{01}, m'_0, y) = (g, q, m')$, where $y = t(p)$. From definition 2.7 we have $f'_1(ys) = f'_1(y)f'_{1_y}(s)$, $f'_2(ys) = f'_2(y)f'_{2_y}(s)$ and $f'_{1_y}(s) = f'_{1_{q,m'}}(s)$ and $f'_{2_y}(s) = f'_{2_{\theta,m'}}(s) \forall s \in \Sigma'^*$. Since $f'_1(s) = f'_2(s) \forall s \in \{y\} \otimes Y'_{q,m'}$, we have $f'_{1_{q,m'}}(s) = f'_{2_{\theta,m'}}(s) \forall s \in Y'_{q,m'}$. Now, the function computed by $P'_1(q)$ is the restriction of $f'_{1_{q,m'}}$ to $\Sigma_0 = \Sigma' - \Sigma$. Similarly, the function computed by $P'_2(\theta)$ is the restriction of $f'_{2_{\theta,m'}}$ to Σ_0 . Thus, since $Y'_{q,m'}$ is a test set of $P'_1(q)$ and $P'_2(\theta)$, $A'_1(q)$ and $A'_2(\theta)$ accept the same language. Hence $A'_1(q)$ is the minimal automaton of $A'_2(\theta)$.

From lemma 7.2 it follows that Y' is a test set of P'_1 and P'_2 .

8 A Method of Testing Simple Coverings

Our method for testing X-machine specifications constructed as simple coverings is based on the theorem above. It assumes that the following conditions are met.

1. The specification P'_1 is constructed as a simple covering of another stream X-machine, i.e. $P'_1 \in R(P_1)$; the type of P_1 (i.e. Φ) and the auxiliary type of P'_1 (i.e. Ψ) are both complete and output-distinguishable; the associated automaton of P_1 (i.e. A_1) is minimal and the associated automata of the auxiliary machines (i.e. $A'_1(q), q \in Q_1$) used in the construction of P'_1 are also minimal.
2. The implementation can be modelled as a stream X-machine P'_2 with the same type (i.e. Φ') as P'_1 .
3. P'_2 is such that $P'_2 \in R(P_2)$, where P_2 is a stream X-machine with the same type (i.e. Φ) as P_1 .
4. The maximum number of states of P_2 (i.e. ν) and the maximum number of states of the auxiliary machines used in the construction of P'_2 (i.e. $\nu'_\theta, \theta \in Q_2$) can be estimated.

Under these circumstances, the set Y' constructed as in the previous section finds *all faults* of the implementation.

The first assumption lies within the capability of the designer. An algorithm for enforcing the completeness and output-distinguishability properties on a set of basic processing functions is given in [16]. Essentially, this extends the input and output alphabet, and thus the processing functions, in a suitable manner; the extra functionality can be removed after the testing is done. The designer can also arrange for the associated automaton of an X-machine specification to be minimal, standard techniques are available [6], [9].

The second and third requirements are the most problematic. They can be ensured by a gradual implementation and testing process as follows.

- First, the set of basic functions $\Phi' = \Phi_e \cup \Psi$ are implemented and the correctness of these implementations is ensured. As discussed in [16] and [18], this can be done with a separate testing process. The stream X-machine testing method 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 [25]) can be used if these are functions that carry out simple tasks on data structures.
- The next step is implementing the auxiliary machines using the correct implementation of the basic functions in Ψ – we will call these *machine implementations*.
- Finally, the implementation of the entire system is constructed using the correct implementations of the processing functions in Φ_e and the machine

implementations. Thus, the system implementation can be modelled as a stream X-machine P'_2 with type $\Phi' = \Phi_e \cup \Psi$ so that the associated automaton of P'_2 (i.e. A'_2) will be made of finite automata (i.e. $A'_2(\theta)$, $\theta \in Q_2$) over the alphabet Ψ that communicate between them through elements of Φ_e as follows:

- given $\theta \in Q_2$, all the states of $A'_2(\theta)$ will behave identically on any transition labelled by an element of Φ_e – this is because the states of $A'_2(\theta)$ correspond to the internal control states of the machine implementation and cannot be viewed by the overall system implementation.
- given $\theta \in Q_2$, there are no arcs labelled by elements of Φ_e that enter any state of $A'_2(\theta)$ other than the initial state θ – this is because θ corresponds to the entry point of the machine implementation.

Thus, there exists a stream X-machine P_2 with the same type Φ as P_1 such that $P'_2 \in R(P_2)$.

The final question that needs to be addressed is concerned with the practicality of the method, that is how complex is the test generation algorithm? Obviously, the complexity of the algorithm will depend on the complexity of the basic functions in Φ and Ψ . Using the results in [16], if the complexity of each $\phi \in \Phi$ is at most C_1 and the complexity of each $\psi \in \Psi$ is at most C_2 , then the complexity of the algorithm that generates the test set will be proportional to

$$C_1 \cdot p \cdot k \cdot r^k \cdot n + C_2 \cdot r^k \cdot n \cdot p' \cdot \max_{q \in Q_1} (r'^{k'_q+1} \cdot n_q'^2 \cdot (2 \cdot n'_q + k'_q)),$$

where $p = \text{card}(\Sigma)$, $r = \text{card}(\Phi)$, $p' = \text{card}(\Sigma') - p$, $r' = \text{card}(\Psi)$. As defined earlier, $n = \text{card}(Q_1)$, $v = \text{card}(Q_2)$, $n'_q = \text{card}(E_q^1)$ for $q \in Q_1$, $v'_\theta = \text{card}(E_\theta^2)$ for $\theta \in Q_2$, $v' = \max_{\theta \in Q_2} v'_\theta$, $k = v - n$ and $k'_q = v' - n'_q$ for $q \in Q_1$.

The complexity of the algorithm depends critically on r^k and $r'^{k'_q+1}$. In practice k is usually low (it is reasonable to assume that the estimated cardinality of Q_2 is very close to the cardinality of Q_1 unless there is a considerable degree of misunderstanding from the part of the developer). On the other hand, if k'_q is to be reasonably low, then the number of states of the auxiliary machines used in the construction of the refinement has to be small. In order to achieve this, a sequence of incremental refinements can be used instead of one single, more complex, process. For critical applications one can make very pessimistic assumptions about k and k'_q at the expense of a large set. In any case, the complexity is lower than that of the algorithm that constructs test sets of P'_1 and P'_2 using only the stream X-machine testing method.

9 Conclusions

The simple covering provides a simple way of developing stream X-machine specifications in an intuitive manner. It has been tried on several case stud-

ies based on real systems and appears to fit in very well with the designers' approach of building the system specification gradually, by expanding its input-output behaviour. However, the approach is in general best suited for the construction of small or medium size software products whose correctness is essential. Furthermore, the functionality of the systems modelled using this approach is limited by the architecture of the refined system. Different types of refinement are obviously needed in order to model a more complex functionality.

One of the strengths of this type of refinement is its associated testing method, that allows test sets to be constructed in parallel with the specification, thus allowing testing to be distributed into smaller chunks, with major cost and time savings. Similar to the stream X-machine testing method, the test set generated is guaranteed to determine the correctness of the implementation under test if certain well defined requirements are met. Basically, these requirements fall into two categories: "design for test" conditions, that can be enforced on the specification at the design stage, and the correctness of the basic components of the system, that can be ensured through a gradual implementation and testing process. This gradual construction of the implementation fits in well with the modular approach used in software development.

Other types of stream X-machine refinement and their associated testing methods are currently investigated.

References

1. Bernot, G., Gaudel, M., Marre, B.: Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, **6**, 387–405 (1991)
2. Bhattacharrya, A.: *Checking Experiments in Sequential Machines*, New Delhi: Wiley Eastern 1989
3. Chow, T. S.: Testing software design modelled by finite state machines. *IEEE Trans. Software Engineering*, **4**(3), 178–187 (1978)
4. Cohen, D. I. A.: *Introduction to Computer Theory*, John Wiley & Sons, Inc., 1991
5. Dauchy, P., Gaudel, M., Marre, B.: Using algebraic specifications in software testing: a case study on the software of an automatic subway. *Journal of Systems Software*, **21**, 229–244 (1993)
6. Eilenberg, S.: *Automata, languages and machines*, Vol. A, Academic Press, 1974
7. Fairtlough, M., Holcombe, M., Ipate, F., Jordan, C., Laycock, G., Duan, Z.: Using an X-machine to Model a Video Cassette Recorder. *Current issues in electronic modelling*, **3**, 141–161 (1995)
8. Fujiwara, S., von Bochmann, G., Khendek, F., Amalou, M., Ghedamsi, A.: Test Selection Based on Finite State Models. *IEEE Transactions on Software Engineering*, **17**(6), 591–603 (1991)
9. Gill, A.: *Introduction to the Theory of Finite-State Machines*, McGraw-Hill, 1962
10. Hierons, R. M.: Testing from a Z specifications. *Journal of Software Testing, Verification and Reliability*, **7**, 19–33 (1997)
11. Hierons, R. M.: Testing from a finite state machine: Extending invertibility to sequences. *The Computer Journal*, **40**, 220–230 (1997)

12. Hierons, R. M.: Adaptive testing of a deterministic implementation against a nondeterministic finite state machine. *The Computer Journal*, **41**, 349–355 (1998)
13. Holcombe, M.: X-machines as a basis for dynamic system specification. *Software Engineering Journal*, **3**(2), 69–76 (1988)
14. Holcombe, M.: An Integrated Methodology for the Specification, Verification and Testing of Systems. *Software Testing, Verification and Reliability*, **3**, 149–163 (1993)
15. Holcombe, M., Ipate, F., Grondoudis, A.: Complete Functional Testing of Safety-Critical Systems, Safety and Reliability in Emerging Control Technologies: A Postprint volume from the IFAC Workshop on Safety and Reliability in Emerging Control Technologies, Daytona Beach, Florida, USA, 199–204, 1–3 November 1995
16. Holcombe, M., Ipate, F.: *Correct Systems: Building a Business Process Solution*. Berlin: Springer Verlag 1998
17. Ipate, F., Holcombe, M.: Another look at computability. *Informatica*, **20**, 359–372 (1996)
18. Ipate, F., Holcombe, M.: An Integration Testing Method That is Proved to Find all Faults. *Intern. J. Computer Math.* **63**(3/4), 159–178 (1997)
19. Ipate, F., Holcombe, M.: Specification and Testing using Generalized Machines: a Presentation and a Case Study. *Software Testing, Verification and Reliability*, **8**, 61–81 (1998)
20. Ipate, F., Holcombe, M.: A method for refining and testing generalised machine specifications. *Inter. J. Computer Math.* **68**, 197–219 (1998)
21. Ipate, F., Holcombe, M.: Generating Test Sequences from Non-deterministic Generalised Stream X-machines, accepted. *Formal Aspects of Computing* **12**, 443–458 (2000)
22. Ipate, F.: *Theory of X-machines and Applications in Specification and Testing*, PhD thesis, University of Sheffield, UK, 1995
23. Laycock, G. T.: Formal specification and testing. *Journal of Software Testing, Verification and Reliability*, **2**, 7–23 (1992)
24. Luo, G., Bochman, G. v., Petrenko, A.: Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalised Wp-Method. *IEEE Transaction on Software Engineering*, **20**(2), 149–161 (1994)
25. Ostrand, T. J., Balcer, M. J.: The Category-Partition Method for Specifying and Generating Functional Tests. *Communication of the ACM*, **31**(6), 667–686 (1989)
26. Stocks, P., Carrington, D.: Test template framework: a specification-based test case study. *SIGSOFT Software Engineering Notes*, **18**(3), 11–18 (1993)