A Tale of Two Tests:
comparing a new test process with an existing industrial process for testing VHDL programs.

Salim Vanak[1], Mike Holcombe[2] and Luke Seed[1].
University of Sheffield.

Abstract. This paper investigates how the Stream X-Machine testing method was used in the context of hardware testing in a leading industrial company. The method was applied to the testing of VHDL alongside the companies own established method of generating test sets. X-machine based tests were generated for a suite of the current designs being produced by the company from both source code and informal design sketches used by the development team. The VHDL source code was then injected with a large number of faults (mutations) and the new test sets applied to these mutations alongside the in-house test sets. The results showed that the X-machine based tests were significantly better (from 50% to 100% improvement) at detecting the injected faults in the VHDL code. They were also more efficient (in one case taking 9 minutes instead of 17 hours to run).

Acknowledgements. This work was carried out while S. Vanak was in receipt of an EPSRC Research Studentship. We would also like to thank ARM plc for collaborating in this research.

1. Introduction.

As semiconductor products become more and more complex, ensuring that these products are error-free is becoming far too time consuming. Today, design verification consumes the largest proportion of system design time. With device complexity on the increase and the pressures upon industry to reduce the time to market, it is necessary to find more efficient and cost effective test methods.

Many approaches to testing are based on *ad hoc* approaches or on the use of tools that generate tests on the basis of information such as the structure of the source code. It has been difficult in the past to generate rigorous tests to determine whether the system is functionally correct, since the representation of the functional behaviour has either been expressed in complex mathematical formulae based on symbolic logic (and thus difficult both to comprehend or to scale up to industrial problems) or on imprecise descriptions written in English and thus unsuitable for using in a test generator.

In the process of detecting faults within a hardware system, there are two basic kinds of faults, Behavioural Faults and Physical Faults. As well as methods used for hardware verification, methods from software are also considered. Some methods used in the behavioural verification of hardware designs are taken from software testing. There are still many methods in software testing that have not been applied to hardware verification. One of the goals of this work is to migrate software testing methods to hardware verification and evaluate their effectiveness [39]. Physical faults are usually present after the fabrication stage of the manufacturing process. These kinds of faults are a result of inaccuracies in the manufacturing process. While there are already well established methods for testing memory [30] we will only be concerned with testing for faults that exist in the logic and its interconnects.

---

1. Department of Electronic and Electrical Engineering.
2. Department of Computer Science.

The next sections show some of the test methods currently in use both in software engineering and hardware design. Our goal is to search for a test method that will speed up the design process by making the verification process less time consuming and more reliable. In addition to providing good error detection, the methods are evaluated in terms of the amount of experience a test engineer would need in order to construct an effective test set. Some existing methods provide a degree of guidance on how to choose the test cases, others leave it totally up to the engineer to decide which test cases will be most effective.

1.1 Behavioural Testing

There are various methods for behavioural testing and these are split into two main categories: Black Box and White Box.

1.1.1. In Black Box Testing (sometimes called data-driven testing) the tester is unconcerned with the internal behaviour of the entity, only with its input-output relationship. Traditionally this means that the construction of a test set capable of finding all the faults is very difficult as the designer may have identified a special case or optimisation that the tester did not consider. This creates extra features in the design that were not present at the specification level and so will not be tested. This is the main weakness of a Black Box Test method and it is the one of the goals of this work to find a black box test method that will not have such a weakness.
White Box Testing (sometimes called Logic-Driven Testing) strategies allow use of the implementation to construct the test set. Unusual or exceptional cases can be seen in the design or program code and test cases can be constructed to look for errors around these points. A Black Box method allows the creation of a test set independent of the design. There is no need to wait until the VHDL version of the implementation has been completed in order to start work on the test set. The construction of the test set can take place at the same time as the design phase, saving valuable time during the design of the device. On the other hand a White Box test method can consider all the extra behaviour the implementer has added.

Examples of Black Box testing are: Equivalence Partitioning and Category Partition testing [32], [34], Cause-Effect graphing [34], Boundary-Value analysis [38] and Random Testing. These all have their strengths and weakness but none are based on any formal theoretical foundation that can provide information about their capability for detecting faults - divergence from the specified behaviour. Some seem better than others in some circumstances but not in all, according to the available empirical evidence.

1.1.2. Measurement or Metric Based testing. This method actually covers a large domain of test set construction methods, most of which are white box test methods. The methods all involve some kind of metric as an adequacy criterion. The test set is then engineered so that it gives as high a performance as possible under the metric. For example, statement coverage is one such metric. A test method that uses this metric would contain test cases that enable the test set to execute as many statements as possible. Usually, multiple metrics are used at the same time in order to get better fault coverage. It is usual to set a threshold score for the metric, once the score for all the metrics is greater than the threshold the test set is complete. There are many other metrics based on circuit structure, FSMs and domain coverage [43]. Ferrandi et al [22] uses an approach similar to stuck at fault testing to introduce error into the design. They consider all the signals as vectors of bits and generate a bit coverage metric. The method assumes that the metric used to evaluate the test set is also a good measure of quality. Of course this may not be the case and can lead to test sets that do not really find many faults.

1.1.3. Manufacturing Testing
By the manufacturing stage, it is expected that all of the behavioural faults have been found and corrected. However, the manufacturing process may itself be inaccurate; faults can be introduced into the hardware causing behaviour that dose not conform to the specification or design. Test sets generated for finding manufacturing faults search for specific types of faults. Traditionally these have been: open or short circuits, faulty transistors and timing faults. However, as chip dimensions decrease and everything becomes closer together, new types of faults become more and more likely. For instance, when two interconnects run next to each other, one of the wires also could induce current in the other wire during a transition. As the chip sizes decrease, the interconnects become closer together, aspect ratios of the wires change and induced currents from adjacent wires become more of a problem. Open circuit faults occur when an interconnect is formed incorrectly and a gap is created in the metal. Short circuits occur between two adjacent interconnects, if one of them is slightly too wide, they will touch, creating a short circuit.

Model based testing There are different types of faults; different types of model are used to model these different types of faults. Usually they are single and multiple stuck-at faults, stuck-open faults, bridging faults and delay faults.

1.1.4. Stuck-at Fault Modelling Open-short circuit faults and faults in transistors can be modelled using a stuck-at fault model. This model suggests that if it can be shown that none of the interconnects (including the inputs and outputs of the circuit) are `stuck at' a particular value (0, 1), then the implementation is reasonably fault free. There are two main types of stuck-at fault models single stuck-at fault and multiple stuck-at fault. The first assumes that there is only one fault in the circuit and attempts to construct test cases to find it. The second assumes that there are multiple faults in the circuit and tries to construct test cases to find all of them. Multiple stuck at fault testing is more difficult to do, but research suggests that, usually, single stuck-at-fault testing is good enough to ensure a reasonably fault free device [2] [33].

Testing Hardware Designs that test for equivalent faults. Two faults are equivalent if they both cause the same failure. Although stuck at fault models are used for generating test sets for most applications there are some cases for which other models like bridging fault models and complex fault models [3] are also used. Other fault models Stuck-at fault models use a circuit model to design test cases that make stuck-at faults visible. Stuck open fault modelling works by constructing a circuit model that make open circuit faults visible. Bridging fault modelling tries to find points in the circuit where short circuit faults will occur. Delay fault models look for faults like slow-to-rise or slow-to-fall. In all these cases, the method used to find test vectors is much the same: Force the circuit in to a state where the error is observable and then check for it (known as Sensitisation and Propagation).

Research in verification is centred in two main areas: Model checking and Behavioural testing. Formal verification techniques are mostly centred around model checking. Formal verification builds models of systems and attempts to show equivalence between the high level design and the original specification. Recent works include: the use of formal verification techniques on microprocessors [29], and FIFOs and token ring controllers [8]. The are tools that support this method which include HISS [4] and CALLAS [24]. Another technique used for formal verification is BDDs; Foster [25] uses BDD to show equivalence between a high level model of a design and its implementation. Ferrandi [23] use BDDs to generate Functional test sets. The Robust Test Method [41] uses a series of orthogonal arrays to reduce

the number of test cases that would otherwise be required by exhaustive testing.

1.1.5. Finite state machine (FSM) testing has been around for many years. Since the FSM model requires explicit representation of the entire state space, a design quickly becomes too large for FSMs to cope with. It is simply not feasible to generate tests for FSMs with thousands or tens of thousands of states. [18] used BDDs [11] to represent the state space symbolically as a series of Boolean functions. Design sizes have already grown to a point that even the above methods cannot handle. For this reason Chein and Jou [15] use a Semantic Finite State Machine as a model for the design. Basically this model collapses states with the similar behaviour in to one semantic state. The state machine is used as the basis for a metric to measure coverage. Cheng [16] tried Extended Finite State Machines (EFSM) to reduce the number of states. Also, [16] develops the idea of an algorithmic functional test set generation method. Instead of an engineer relying upon their own experience to reach a test coverage target, an algorithm is used to guarantee all the statements in the code have been executed at least once. Borrione [9] uses an FSM with data path for the verification of high level VHDL descriptions against the post synthesis version. Chow [17] introduces a test method for testing systems based on finite state machines. Chow provides a test generation method that is proved to find all the behavioural differences between two different FSMs. Since systems state spaces can become unmanageable for FSMs, Ipate [27] took Chow's test generation method for FSMs and applied it to Stream X-Machines. The result was a test generation method that could effectively model any computational process and generate a test set that was proved to find faults. This section has outlined some of the methods used for verification and testing of hardware systems. Although these methods are structured and provide a “rough guide" to generating a test set; some methods even provide an algorithm for generating the test inputs once a test case has been identified.

Table 1 measures each test method against three different criteria:

1. *Experience* shows the amount of input or creativity the designer can input in to the design of the test cases. Low signifies that the test case generation has a strict algorithm to follow, moderate means that there is some guidance provided by the method but no clear algorithm and high means that the test cases are constructed entirely by the engineer.
2. *Dependence on the design phase.* This column show the how much the test method considers the design (VHDL) in order to generate a complete test set. A method with a low score will generate test cases independently of the design. A method with moderate score will require some assistance from the design but will also use the specification. A high dependence will not require a specification and test cases will be constructed entirely from the VHDL design.
3. *Assurance of Quality.* This column measures how easy it is to gauge the quality of a test set once the test set has been constructed. A provable score means the test set provides an assurance that every test set will satisfy a particular goal. A measurable score means that although no formal proof exists of the quality of the test set, this method can provide a measure of how good the test set is. None means that the test method provides no assurance of quality.

Table 1: Comparison of Test Methods

Test Method	Experience	Dependence on Design Phase	Assurance of quality
Equivalence Partition	Moderate	None	None
Boundary-Value analysis	Moderate	None	None

Table 1: Comparison of Test Methods

Test Method	Experience	Dependence on Design Phase	Assurance of quality
Cause-effect graphing	Moderate	None	None
Category-Partition	Moderate	None	None
Metric Based	High	High	Measurable
Model Based Testing	Low	High	Measurable
IDDQ Testing	Low	High	Measurable
Robust Test Method [41]	Moderate	none	Measurable
Chow [17] & Ipate [27]	None	None	Provable
Cheng's EFSM [16]	None	None	Provable

The goal of this survey was to find a test method that will reduce the overall design time of a device and provide a reliable method for verification. Of all the methods shown in this chapter only one fits this criteria. All of the test methods with a measurable degree of quality need a complete design before work can begin on testing. If the method can generate test sets before the completion of the design, the test set can be constructed in parallel and save a large about of time. One of the methods that seems to satisfy out requirements is Cheng's EFSM test method [16]. It is a black box method that gives a proof of quality. However, this method only proves that the test set will give 100% statement coverage of the design. As the later chapters will show, statement coverage alone is not a reliable measure of test set quality. The only current method that fits the requirements for this work is Ipate's [27] X-Machine. The Stream X-Machine model and the X-Machine test method provide a Black Box test method that is capable of finding all of the control faults in a design. Since this method is Black Box, it can be used independently of the design allowing test sets to be constructed as soon as a specification is available. The test generation Algorithm allows a test engineer with little testing knowledge to construct a test set that will still satisfy the proof. Further more, the X-Machine model dose not simply provide a test method. The X-Machine model is a complete specification method that allows any process to be modelled, be it hardware, software of some combination of hardware and software. The following sections demonstrate that testing a hardware design as an X-Machine model is useful and worthwhile.

2. A new type of model.

In 1974 Enabler [20] formulated a new, generalised state machine automaton, an X-Machine, which could be used to model processes. In 1995, Ipate [27] constructed a method for using X-Machines together with ideas borrowed from Chow [17] that allow the construction of test-sets that will allow the designer to guarantee that their design is correct.

An X-Machine [20] is a finite state machine with memory for storing data and a set of data processing functions. The data processing functions allow the consumption of inputs, the production of outputs, reading and writing to the memory and can perform simple data processing tasks. This enable us to model the memory more directly and cuts down on the state space explosion. In 1995, Ipate [27] used a refined version of the X-Machine model called a Stream X-Machine. Ipate showed that if a process is

modelled as a Stream X-Machine a test set can be generated by a method similar to that described in Chow [17]. This test set will be able to find all the behavioural differences between two Stream X-Machines or more usefully in the case of this work a specification and a design.

2.1. The Stream X-machine model. In 1995 Ipate [27] used a modified X-Machine model called a Stream X-Machine. A Stream X-Machine looks, from the outside, like a FSM but retains some of the computational power of an X-Machine. Formally an Stream X-Machine is described as follows:

Definition (A Stream-X-Machine) is a tuple:

$A = (M, \Sigma, \Gamma, S, \Phi, \delta, i_0, m_0, T)$

$X : \Sigma \times M \rightarrow M \times \Gamma$ the fundamental data set the machine operates on.

M is the set called memory.

Σ is the set of input symbols.

Γ is the set of output symbols.

S is a finite set of states

Φ is a set of relations $\phi : P(\Sigma \times M \rightarrow M \times \Gamma)$

δ is the transition function $\delta : S \times \Phi \rightarrow S$

i_0 is the initial state

m_0 is the initial memory

T is the set of terminal states

Notice that a Stream X-Machine is deterministic. If a Stream X-Machine satisfies certain design for test conditions, it is possible to generate test sets to find behavioural differences between two Stream X-Machines, however this model is not suitable for modern devices and needs to be adapted for this type of application.

2.2. Multi Stream X-Machines.

Suppose that the Stream X-Machine model is modified to produce a Multi Stream X-Machine. Multiple inputs are handled in a synchronous manner and modelled as a tuple of values. For example, a Multi Stream X-Machine with two input streams will expect pairs of values as inputs. We assume that the string of symbols on both input streams are the same length. Similarly for the outputs, if the Multi Stream X-Machine had two output streams, then the outputs would be produced as pairs. Both output streams would contain the same number of output symbols. In this respect a Multi Stream X-Machine is no different from a Stream X-Machine. In the case of a Multi Stream X-Machine the input and output alphabets Σ and Γ are tuples of values rather than single values.

When using this model with a synchronous hardware design, it is assumed that at every clock cycle the new input will have arrived, and the old outputs will have been dealt with. This way, at each rising clock edge, the symbols on the output are assumed to be the symbols written by the previous transition. The symbols on the inputs are assumed to be the next symbols to be read from the input streams.

Each of the transitions between each state performs a computation. This computation is defined by a function ϕ and a pre-condition, ϕ_{pre}

Definition 1. (Pre-conditions). A function pre defines a pre-condition for every ϕ in the Machine A.

$$pre: \Phi \rightarrow [\Sigma \times M \rightarrow Bool]$$

where Φ is the set of data processing functions and $[\Sigma \times M \rightarrow Bool]$ is a set of primitive recursive predicates. The application of the function pre to a ϕ is written ϕ_{pre}. ϕ_{pre} denotes pre-condition of the function ϕ.

Definition 2. (Multi Stream X-Machine (MSXM)) The machine A, with i inputs and j outputs, is a 10-tuple $A = (M, \Sigma, \Gamma, S, \delta, S, i_0, m_0, T,$ pre)

M the memory of the machine.
Σ is a set of i-tuples making the inputs to the MSXM
Γ is the set of j-tuples making the outputs to the MSXM.
S is a finite set of states
Φ is a set of Primitive Recursive Functions Φ: $P(\Sigma \times M \rightarrow M \times \Gamma)$
δ is the transition function $\delta : S \times \Phi \rightarrow S$
i_0 is the initial state
m_0 is the initial memory
T is the set of terminal states
pre as for pre in Definition 1

The execution of a MSXM will follow the steps below:

1. "*current state*" is i_0 and "*current memory*" is m_0
2. σ = the next input.
3. Find δ("*current state*"; σ) ="*new state*" such that ϕ_{pre}("*current memory*"; σ) holds.
4. $\Phi(\sigma$; "*current memory*") = ("*new memory*"; γ), when γ is the output.
5. let "*current state*" = "*new state*" and "*current memory*" = "*new memory*"
6. if "*no more inputs*" ^ "*current state*" $\in$ T then END else goto 2

3. Test generation.

The full details of the test generation method is to be found in [46]. We summarise the process here. Take a Multi Stream X-Machine, A. Let

Φ be the set of data processing functions, contained in A
$\Lambda = l(\Phi)$
M be the set of all memory values for A
m_0 is the initial state of memory of A
S is the finite set of states of A

From Definition 3, there exists a language - call it L_A - that will define a subset from Φ^* that will be executable by the MSXM A.

Design-for-test conditions

1. Let there be two MSXM's, A and B, that operate over the same set of functions Φ and the same pre-conditions pre.
2. The functions in Φ are output distinguishable.

$A_i \rightarrow A_j$ indicates that the MSXM A goes from a state A_i to a state A_j using the function ϕ. We can dis-

tinguish between every state in the machine A by an isomorphic function

Definition 5 (The Map Function) map is a bijective function that maps states in A to states in B so that:

if A_i is in A then map(A_i) is in B

Using L_A a test set for A is constructed.

For each (s, λ) ∈ $S \times \Lambda$ try to find a path p such that:

$p: i_0 \rightarrow s$ and $p\,\lambda \in L_A$

Lemma 1 shows that the existence of such a path is decidable, so it is reasonable to define the following function

Definition 6

$\tau : S \times \Lambda \rightarrow (L_A \times L_A) \cup \{$ bottom, bottom $\}$

$\tau(s,\lambda) = (p, p\lambda)$ if $p: i_0 \rightarrow s$ *exists and* $p \in L_A$ *and* $p\,\lambda \in L_A$

(bottom, bottom) otherwise.

Definition 7 (The test set P)

$P = \{a, b \mid (a, b) \in \tau\,(S \times \Lambda) \setminus \{$ bottom, bottom $\}\}$

Definition 8 Two Multi Stream X-Machines A and B are P-equivalent if there exists an initial state i_0 in A and an initial state map(i_0) in B such that

for all $p \in P$, there is $x \in S$ such that p is a path $i_0 \rightarrow x$ iff p is a path map(i_0) $\rightarrow$ map(x)

P is the test set from Definition 7.

4. The method for comparing the new X-machine based testing method with an industrial standard approach.

Practical Work At this point there exists a proof that allows the designer to guarantee in the control part of any device, that the X-Machine test method is capable of finding all the differences between the design and the specification. Unfortunately most devices that exist today also contain a data path. The X-Machine test method suggests that the designer uses a different testing technique to test the data path. It is worth noting that during the course of exercising the control part the data part is also exercised. The rest of this work will try to assess how effectively the data part is tested using the X-Machine test method. Four devices have been investigated: a 6502 microprocessor, an 8051 micro-controller, the APB-to-AHB bridge and the AHBLite-to-AHB wrapper from ARM's AMBA bus.

All four of these devices have their own test sets developed by the engineers that designed the device. However, since the ARM devices are commercial products and the two microprocessors are freely available on the web, the ARM devices have been much more rigorously tested than the microprocessors. In each case the following series of experiments will be performed:

1. An X-Machine Specification will be constructed.
2. A test set will be constructed from the X-Machine specification using the X-Machine Test Method.
3. Measurements will be taken from the specification to assess the nature of the device. This purpose

of this it to give the reader an idea of the type of device being tested.

4. Measurements will be taken from the performance of the two test sets (X-Machine and the designer's original test set). These will include standard metrics such as code coverage and an error seeding metric.

The results from these experiments will be sufficient to give an indication of two things:

1. How tests generated from the X-Machine test method compare with test set generated by experienced designers
2. If the X-Machine test method is an efficient and intuitive way of generating meaningful test sets.

Software

Due to the sheer size of the devices and their test sets, carrying out this series of experiments manually would be both time consuming and difficult. It is far more useful and efficient to design a set of tools for constructing the test sets themselves and to calculate the mutation coverage.

4.5.1 The Test Function and Test Generation

Since the test generation method has been altered significantly from the previous method, [27] there does not exist a program that calculates the test set for a Multi Stream X-Machine. Whilst there is a long standing algorithm for constructing input sequences from sequences of ϕs, until now it has never been implemented. The tool constructed for generating test sets from X-Machines also contains the test function algorithm. Thus, the tool give a series of input/output sequences that can be fed directly in to the device. In addition to the functional requirements of the tool, the non-functional requirements are: This tool is simply a tool to speed up the work carried out in this study. There is no need for fancy user interfaces. The algorithm is written in GNU Prolog. The program will require a model of the X-Machine in executable form. In GNU Prolog this can be accomplished using the following template. The transition functions will be tabulated using the following Prolog clause: delta(state, transition, nextState)

The state is the current state of the X-Machine; transition would be the name of a ϕ function and next-State is the state reached by executing this transition.

The ϕ functions are defined in the following way:

> phi(Name, IN, IN_mem, OUT_mem, OUT) :- some Prolog clauses to model the behaviour of this function.

Here, Name represents the name of the function, IN represents the inputs to a function and is usually a tuple or a list if there are multiple inputs. IN mem and OUT mem are the initial and resultant memory values respectively. The memory can take any form, the case studies all use lists to model the memory. OUT is the outputs for the function, again this is usually a tuple of a list.

In addition to this data additional data is provided in the following clauses: initialstate(state) and initialmemory(M) represent the initial state and memory respectively. The Prolog algorithm also needs the associated automaton see [26]. In order to calculate the associated automaton the Prolog algorithm requires a definition of the abstract memory machine. This machine is defined using the following clause: deltaMem(state, trans, nextState). The deltaMem clause shows the transitions between one abstract memory state to another. The labels state and nextState are not important so long as they are consistent. trans is the name of a ϕ function. Also a clause initialstateMem(m) is required. The term m is the initial abstract memory state.

There are several reasons for choosing GNU Prolog:

1. Prolog will search over the inputs to a function itself. In Prolog it is possible to take a clause and get the Prolog search engine search for values that allow the clause to evaluate to true.
2. The is already a version of the X-Machine Design Language that is written in Prolog [****]. Writing this tool in Prolog makes it easier to integrate the two tools later; if so desired.

Mutation Testing

The mutation testing metric requires many (of the order of 100s or 1000s) of mutated versions of the device to generate a meaningful measure. It is impractical to perform this task manually. A tool has been constructed to create mutated versions of a device written in VHDL.

Tool Requirements

The tool is required to inject one fault into a device written in the 1993 standard of VHDL. Since there is little point in creating thousands of mutated devices that do not compile. The tool injects faults into the design in such away as to guarantee compilation. In order to keep the tool as simple as possible, no attempt will be made to analyse the semantics of the VHDL, nor will any attempt be made to analyse the typing constraints for the VHDL code.

The Coupling Effect

Offutt [35] produced experimental results showing that test sequences capable of detecting simple 1st-order mutants are also capable of detecting more complex nth-order mutants. The survey showed that a test set that has 96% coverage over 3rd-order mutants also has over 99% coverage over 1st-order mutants. For this reason the mutation testing tool concentrates its efforts on introducing very simple static syntactic mutations in to the device under test. In this way, if it can be shown that the test set is finding many of these mutations, it should find more complex mutations that have a greater effect on the behaviour of the device.

Mutation Injection

This section outlines the types of mutations introduced into the device.

If conditionals

This involves mutating the conditionals in an *if* statement. Most conditionals contain logical operators, they are split into classes. The first is the operators that have two arguments of equal types, for example, $=, \neq$. The second class consists of those that have two arguments of boolean type, for example, $=, \neq, \vee, \wedge$ The third are those that operate on real or integer numbers, $<, >, \geq, \leq, =, \neq, \vee, \wedge$ the $\vee$ and $\wedge$ are bit-wise operators.

Each operator in the conditional can be swapped with another operator in its class. Thus guaranteeing the code to be free of syntax errors.

If branches

code that is of the form

```
if cond then
        then-part
else
        else-part
end if
```

will be mutated to this

```
 if cond then
        else-part
 else
```

```
                then-part
        end if
```

VHDL allows the elsif keyword. In this case this mutation will swap two branches in the if statement at random.

Case Labels

```
code that is of the form
        case var of
                A => do-A
                B => do-B
                C => do-C
                others => do-others
        end case
will be mutated to this
        case var of
                A => do-A
                C => do-B
                B => do-C
                others => do-others
        end case
```

Mutation operations on case statements have the same idea as those on an if statement. A label in the case statement is simply swapped for one of the other labels. These mutations are designed to create an error in the code that will always compile. It maybe that sometimes a mutation is introduced that causes no change in behaviour. This can be eliminated at the end of the test runs when the mutation score is calculated.

The construction of the programs to seed mutations in VHDL code requires a VHDL parser. There are several parsers available under the GNU GPL from web site such as http://tech-www.informatik.uni-hamburg.de/vhdl/ Four parsers were tested: VHDL grammar, VHDL lex/yacc grammar; VHDL-93 parser in Prolog, A VHDL-93 compliant parser written in SWI-Prolog and VHDL parser in Java, VHDLParser and VHDLTree: a GNU licensed VHDL 58 Methods and Tools parser and parse-tree viewer written in Java. From these parsers the Java VHDL parser was used. This was primarily because it worked “straight out of the tar file". The Java VHDL parser uses the javacc and jjtree these tools together with a grammar read a VHDL file into a parse tree that the mutation testing program can manipulate. There is a known bug in the parser that was not solved. When parsing if statements, the conditional must be enclosed in brackets. VHDL syntax does not enforce the use of brackets around the conditional; but the tool will not parse properly conditionals that are not enclosed in brackets. The mutation testing program is called with the following command line java SeedFaults <input file> [<random seed>] if the <random seed> parameter is omitted the program will restore the random seed from the end of the last run of the program. Also the program maintains a record of the previously used random numbers to ensure that the same set of random numbers are not generated twice. This is to prevent the same fault occurring twice in the same run. The set of previous random numbers is deleted every time a new random seed in introduced.

Automatic Test Generation

Now there is a method for generating test sets; for complex specifications the MSXM models may become complex and test generation will become more difficult. In the interest of minimising errors in the test generation process an algorithm has been generated. This algorithm takes a high level executable MSXM specification and generates a test set; it is implemented in Prolog and requires the specification be written in Prolog.

The following tools have been developed:
testGen2 An algorithm that will take an executable Prolog MSXM specification and construct a set of input sequences that is guaranteed to find all the faults (of the kind shown in Section 2.3.2) in the implementation.

DocScripts are a set of scripts that are used for documentation purposes. Once a Prolog specification is constructed these scripts allow the easy construction of state space diagrams, translation of the test set into a LATEX format. These tools are not optimised for performance. It is also not known what size of problems the *testGen2* program can handle.

The algorithm for *testGen2* is shown below:

```
Calculate which transitions can never occur.
FOR-every state ∈ S
    FOR-every function ∈ Φ
      IF function can occur from this state
          Find a path to the state, P.
                IF P ∧ function is not executable THEN
                   redo the previous step.
                FI
                add P ∧ function to the test-set.
       FI
    ROF
ROF
FOR-every t in the test-set
    Construct a sequence of inputs, i, so that each function in t is substituted for an input token
      that will make it fire.
          IF i is not executable THEN retry the previous step. FI
    add i to the set of inputs
ROF
```

Effectiveness of the X-Machine Test Method
Unfortunately, searching for the correct values for P and i is quite complex in both time and space (the travelling salesman problem). The complexity is increased by the path to the state begin allowed to revisit the same state multiple times. At the moment, this search process is very `brute force'; it may be possible to reduce the complexity of the problem by using some kind of guided search or another smart way of constructing a test set.

5. Results.
Example 1. The APB-AHB Bridge is a modern, real world example of a device that provides a challenge when it comes to verification. The AMBA devices are not public domain and the APB-AHB bridge devices was obtained from ARM plc covered by a Non-Disclosure Agreement. The AMBA devices come complete with the documentation used to design the device. ARM have their own test sets and test methods which they use to verify these devices and unlike the previous examples, these must

be sufficient to underpin the commercial nature of the device. Even with all this information, an X-Machine model needs to be reverse engineered from the documentation and design. ARM's design process includes the design of a state machine to model the behaviour of the device.
This fact makes constructing the X-Machine model much easier. The design of the X-Machine is much closer to what would be expected if a device had been created from scratch using the X-Machine model. Having said this, the design for test conditions are not fulfilled. The problem is ensuring that the f functions have been tested and are error free. This is because there is no way to test the data path separately from the state transition functions and the design of the X-Machine test set never considers what happens to the data path. This case study will compare the X-Machine test set with the test set developed by ARM. ARM's test set fully tests the device and has been used to verify the APB-to-AHB bridge for commercial use.
The APB-to-AHB bridge is more difficult device to test than standard microprocessors as its transition structure is more complex. ARM's test set actually runs the device in the environment it is intended. Different types of transfers are sent across the bridge, some transfers are seeded with errors, others are error free. The designers know what the device should do and how it should operate. Using the design documentation and their own experience, corner cases in the design are identified as possible sources of erroneous behaviour. These corner cases are then tested thoroughly by the test environment. The test set takes the form of a file of data that is sent across the bus 1 and the bridge transfers the data to bus 2. A device sitting on bus 2 verifies whether the correct data has arrived. Also a series of protocol checkers and other devices supplement the test set in search of other erroneous behaviour. The whole process is reliant on both the quality of the design documentation and the competence of the test engineer.
It is not possible to show a diagram of the bridge at it is commercially confidential.

In contrast the X-Machine test set is much lower level. The test set for this device takes the form of a simple VHDL test-bench that drives the device with the test signals. The X-Machine test set does not require a complete bus architecture to test the APB-to-AHB bridge and so the test bench design time is reduced significantly. The X-Machine test set does not require actual transfers to take place over the bridge, only an exploration of the state transition diagram is required. As a result the test set executes much faster. The observability requirements of the X-Machine test method are met by using the simulator to produce a signal trace of selected signals during simulation.

Mutation Score
The mutation scores for the X-Machine test set were higher than the mutation score for ARM's test set. All the faults missed by the X-Machine test set were also missed by ARM's test set. Of the four faults missed by the X-Machine test set, two were equivalent, leaving two faults that were actually missed. The ARM test set left eight live mutants, most of which were faults that changed the initialisation behaviour of the device. This seemed to be an oversight in their test methods until an ARM engineer pointed out that initialisation tests are carried out manually after the device is completed and are not part of the test suite. Out point of view is that if something can be tested automatically, it should be.

Table 2: Use of Mutation Operators for APB-AHB Bridge and coverage results.

			Mutation Score
Mutation Operator	Mutations	X-Machine	ARM

Table 2: Use of Mutation Operators for APB-AHB Bridge and coverage results.

			Mutation Score
IF Branch Mutation	42	1.000	0.920
CASE Label Mutation	25	1.000	0.952
IF condition Mutation	34	0.912	0.824

Table 2 shows the use of mutation operators. Table 3 gives coverage results for both the XM test sets and the ARM test sets.

Table 3: Coverage results

	XM	XM	XM	ARM	ARM	ARM
Coverage	percentage	executed	total	percentage	executed	total
Statement	100	91	91	100	91	91
Branch	100	83	83	100	83	83
Basic sub-condition	100	46	46	100	46	46
Focused Expression	91.3	42	46	100	46	46
Multiple sub-condition	46.6	41	88	53.4	47	88

In this example the higher the statement and branch coverage the more faults were found by the IF Branch and CASE Label mutation operators. Condition coverage does not correlate, the higher code coverage of the ARM test set results in the ARM test set finding less of the faults introduced by the IF condition mutation operator. Again this shows how the condition can be a misleading measure of the quality of a test set. From the measures taken in this case study, the two test sets seem to be similar in quality. The major difference being the X-Machine test set took approximately half the time to execute and requires a simpler test environment to execute the test sequences. The design of the X-Machine models took about 3 days, the X-Machine test set was generated automatically. The translation to the final test environment took a matter of hours to design. In contrast ARM's test set took one of their designers approximately 5-7 days to design. An experienced designer that knows how the device operates may be able to construct the test X-Machine model and test set much faster than an inexperienced hardware designer who is unfamiliar with the device in question.

Example 2. The AHBLite-to-AHB Wrapper (*The Wrapper*) is a relatively new device with a reasonably large state space that will make the verification task non-trivial. This device is similar to the APB-to-AHB bridge in Example 1. Although the AHBLite-to-AHB wrapper is not as central or critical to the AMBA bus as the APB-to-AHB bridge, it provides a simple, easy to use access point to AMBA's Advanced High-performance Bus (AHB). The device comes complete with design documentation used

by the engineers and its own test set. With all this information an X-Machine model needs to be reverse engineered from the documentation and design. The ARM design process includes the construction of a state transition diagram to model the behaviour of *The Wrapper*; this greatly helps the construction of the X-Machine model as the state space and transition structure are already defined. This brings the whole design process a little closer to the ideal situation where, a designer would construct a specification as an X-Machine and use it to construct the design. Once again, the design for test conditions are not fulfilled. The problem is that it is difficult, from the way the device has been designed, to ensure that the f functions are error free. This is due to the fact that there is no clear distinction between control path and data path. Since this problem cannot be solved easily no changes are made to the VHDL design to facilitate testing of the data path. One change has been made to *The Wrapper*'s design to satisfy an observability and controllability requirement of the X-Machine test method. *The Wrapper* contains two state machine; two X-Machines are required to model its behaviour. These machines work in parallel and are connected together using a series of internal signals. In order to control the second state machine the internal signals need to be exposed. Figure 2 on the facing page shows the modified device. *The Wrapper* is split into two devices, a Lite2AHB module and a Lite2AHBTest. The Lite2AHB device directly replaces the original Lite2AHB device and is for normal use. The Lite2AHBTest is for testing purposes only. This device is the only device in this study where the X-Machine composition approach is used to model a device. Although the device is relatively small compared to Microcontroller, the two communicating state machine provide sufficient complexity to make testing a challenge. Figure 2 shows the device as it was originally designed.

Figure 2. Architecture adapted for testing.

ARM tested this device with a couple of other devices. As a result the test set is very large and takes a long time to execute. The test set is made up of two parts: a structured test set and a random test set. The structured test set is deigned in much the same was as for the APB-to-AHB Bridge. Corner cases are identified and test cases are generated to explore the corners. The structured test set takes 1-2 hours to execute. The random test set is generated by a C program that constructs test cases by randomly sending different types of transfers through *The Wrapper*. The results are computed and cross checked against the design. Errors are also introduced, by the C program, into the transfers. The random part of the test set takes about 10- 15 hours to execute. Using the mutation metric on such a large test set is not possible.

The X-Machine test set, in contrast, is very low level. The test set takes the form of a simple VHDL test-bench that drives *the wrapper* with the predefined test signals. The X-Machine test set does not

require a complete working bus architecture to test *the wrapper*. As a result, the design time is significantly smaller than for the ARM test set. The X-Machine test set is designed to explore the state space and transition structure of the device under test. No actual transfers are sent through *the wrapper* resulting in a test set that executes much faster that the complete ARM test set, taking only minutes as opposed to the overnight run the ARM test set requires. Observability requirements are met by using the simulator to produce a signal trace for comparison against a fault free version of the device.

As the ARM test set cannot be used, a portion of the random test set is used to compare again the X-Machine test set. Two runs are performed. The first run is a randomly generated test set that takes the same amount of time to execute as the X-Machine test set. The second run is a randomly generated test set ten times longer that the first run.

Coverage Results

Although the X-Machine test set coverage results are encouraging at over 90%, ARM's coverage results are well below the expectation. As this is not the complete test set, this is to be expected. The figures are shown in more detain in the tables.

Table 4:

	A	R	M	A	R	M	XM	XM	XM
Coverage	%	ex.	total	%	ex.	total	%	ex.	total
statement	85.2	127	149	87.0	131	149	96.6	144	149
branch	81.1	142	175	85.7	150	175	94.9	166	175
basic sub-condition	84.1	153	182	88.5	161	183	94.0	171	182
focused expression	68.0	117	172	77.3	133	172	82.6	142	172
multiple sub-condition	18.7	140	748	22.9	171	748	25.7	192	748

Since these test sets are dramatically different in their nature slightly different measures are taken. Both the test sets are simulated, once for every mutation introduced, on a VHDL simulator. For the X-Machine test set, signal traces of the outputs to the device and the state variable are used to compare this simulation with a fault free simulation. ARM's test set detects the error itself, on executing the test set the word \#ERROR#" appears in the simulation transcript file if an error occurred.

Mutation Testing

The Mutation testing is performed on the APB-to-AHB bridge. The process is as follows:

Inject fault.
Compile the mutated VHDL.
Simulate the mutated device in the simulator
Record the result and compare the signal trace
(in the case of X-Machine)

or look for the ``#ERROR#" word in the transcript file
(in the case of ARM)

In the case of the X-Machine test set, one iteration takes approximately 1 minute. The ARM test set takes longer to execute and so takes about 2 minutes to execute one complete iteration. 101 iterations were performed resulting in 27 equivalent mutants and 74 unique mutants. See Table 5 for overall results from the mutation testing, the mutation scores are shown in Table 6

Table 5: Mutation Scores for *The Wrapper*

VHDL File	X-Machine	ARM	Long ARM
APBif.vhd	0.847	0.588	0.588

In this case study the X-Machine test set performs significantly better than ARM's random test generation method. Coverage results for the X-Machine test set are encouraging. Statement and Branch coverage are all over 94%. The basic sub-condition coverage is also over 90%. This is as expected since the missing parts are mostly data path. ARM's test set is much poorer in coverage performance. With only statement coverage reaching just over 85%. The rest of the values fall far short of what is expected from a useful test set. Increasing the number of transfers over *the wrapper* by a factor of 10 only increases the coverage, on average, by a factor of 0*:*1.

Mutation Score

The mutation scores for the X-Machine test set are higher than the mutation scores for ARM's test set. All the faults missed by the X-Machine test set were also missed by the ARM test set. Table 7 shows a break down of the mutation scores for each mutation operator. There is little anomalous behaviour in these results and every thing is as predicted. X-Machine mutation scores for all three mutation operators are high. The ARM test set shows a very low score for case label mutations. However this is only 7 of the 28 live mutants left by the ARM test set that were killed by the X-Machine test set. Increasing the size of the random test set had no effect on the mutation scores for the ARM test set.

Table 6: Mutation Scores for *The Wrapper*

Mutation Operator	Mutations	X-Machine	ARM	ARM Long
IF Branch	49	0.861	0.556	0.556
CASE Label	14	0.818	0.364	0.364
IF condition	38	0.842	0.684	0.684

6. Discussion

This work has attempted to bring Stream X-Machine testing to Hardware designs. Stream X-Machine testing contains various assumptions that restrict the way in which the engineer can design the device. For example, one of the assumption is specification completeness. Completeness only allows the Stream X-Machine model to make branch decisions based on the current input. It does not allow branch decisions to be based on past events or the values stored in memory. Although such behaviour can be simulated, it would, in most cases, require extra states and so make the design more complex. In order

to simplify the design process and make the modelling of a process using the Stream X-Machine model easier the Stream X-Machine model is modified to create a Multi Stream X-Machine. This new Multi Stream X-Machine model has be created to ease the design of a hardware design by removing the completeness design for test condition. Also the model reflects the fact that the hardware design and will have multiple inputs and outputs instead of the single input and output stream of the original Stream X-Machine model, i.e. it is a better match with concurrent hardware designs. A test method has been created for the Multi Stream X-Machine model. Although the generation method becomes more complex when the completeness assumption has been removed, test set generation has been implemented and has been shown to automatically generate test set for designs containing thousands of states.

Now that there exists a method to effectively test a hardware design, it is necessary to assess whether or not such a method was useful when it comes to actually designing hardware systems.

This paper involved two devices that perform important tasks on the AMBA on-chip bus. The Multi Stream X-Machine model provides a simple and intuitive method for specifying hardware design. Multiple inputs and outputs can be modelled easily using multiple input and output streams. Also a Multi Stream X-Machine provides guarded functions which makes capturing a process with memory easier and more intuitive. Multi Stream X-Machines were reverse engineered from both microprocessors and test sets were automatically generated from these specifications. The two test sets were compared with those test set constructed by the designers. The results showed that, although the designers' test set found a large proportion of errors, the X-Machine's test set found more. The differences between the two test sets are small but do show that the automatically generated X-Machine test sets are no worse than those laboriously hand crafted test sets already in use. Using the X-Machine test method would thus be a valuable assurance to the designers on the quality of the test set. ARM plc kindly provided two devices used on its AMBA bus. Along with the hardware descriptions ARM gave access to the engineers that designed the test sets and all the documentation generated by their design process. The Multi Stream X-Machines were reversed engineered and the X-Machine test sets were compared with ARM's own test sets. The results showed that, although the X-Machine test found more of the faults than ARM's test set, the difference was small. However the X-Machine test sets showed one clear advantage; the X-Machine test set was, in both cases, significantly smaller than ARM's test set. The design times for both X-Machine test set and ARM's test set were very similar. Remember that this comparison is of experienced ARM test designer with many years of experience testing similar devices against someone with very little experience of hardware design and testing let alone testing ARM devices. Thus a person with no experience of testing hardware designs can perform as well as an experienced engineer. Also, the Multi Stream X-Machine method described in this paper is a complete method starting at specification and ending at verification test. If the method is followed strictly, ensuring the design for test conditions are met, the test generation method becomes a simple automated. This means that even when the scale of the device gets considerably larger, the test set design time remains small. There are important differences in the design of ARM's test set and the design of the X-Machine test set. The focus of the X-Machine test set is to find faults; differences in the design that would result from faulty state configuration or faulty transition. ARM's test set is focused on operating the device in its environment and checking to see if: it complies with the AMBA protocol, interacts with other devices on the AMBA bus and whether it can perform all the tasks it may be required to perform. As a result the X-Machine test set analyses the device at a much lower level that the ARM test set. This means that the ARM test set would be able to pick up errors introduced into the device at the specification phase. The X-Machine test set would simply compare the behaviour of the device with the behaviour of the specification. The specification errors are dealt with by model checking the X-Machine for

the required properties. Rather than trying to do the model checking and verification task in one step, the X-Machine method breaks up the task into two simpler problems. This has shown a large cultural difference between the testing philosophy of the X-Machine test method and those of a leading hardware designer like ARM, who are not used to working with formal specifications.

When designing a test set it is difficult for a designer to know when to stop. X-Machine test sets could act as the minimal level of testing. A designer could start be ensuring that the test set can show P-equivalence and then build from there. This study has shown that metric based test generation methods maybe good at finding errors but they are not the most efficient.

Although metrics (such as code coverage) exist, if used carelessly they can produce a misleading or false picture. The case studies in this paper have highlighted an interesting problem. In all the case studies shown, higher condition coverage resulted in lower mutation scores for the if condition mutation operator. This is easy to explain: when measuring the condition coverage we are concerned with the inputs and what percentage of the possible inputs are tried. Actually what we want to measure is if the behaviour of the conditional changed, can the test set reveal the change? For example, consider a simple AND gate. To test this we might choose a test set consisting of one test case (1*;* 1) and expect the output of (1). If the behaviour of the condition changes to that of an OR gate. The this test set cannot tell the difference between the AND gate and the OR gate. On the other hand if we choose (1*;* 0) to test the AND gate's behaviour, we can differentiate between the AND gate and an OR gate. Both test sets you give the same condition coverage under all the coverage metrics used in this study but they both differ in their ability to discover a change in behaviour. This is a over simplified example, but as the complexity of a conditional grows, it becomes more difficult to identify a useful test case from a redundant one for the purpose of detecting a change in behaviour: condition overage may not be the best way to measuring the ability of a test set to detect a change in behaviour.

Conclusions.

This paper has made the following contributions.
1. A Novel X-Machine Model is described that is designed to be a better match with hardware designs and make capturing a process easy and intuitive while still allowing the generation of meaningful test sets.
2. A Testing Method is described in which shows how the Multi Stream X-Machine model can be used to generate meaningful test sets from the specification. This mean that quality test sets can be generated in parallel with the design of the device, saving valuable time in the design process.
3. Experimental Work: This report documents the first comprehensive study of the effectiveness of the X-Machine test sets in a modern industrial setting. The case studies in this report take two devices and compares the performance and error detecting ability of the X-Machine test method with the industrial strength test sets used by the engineers that created the devices.
4. Deficiencies in other test methods: This report also highlights problems with code coverage as a measure of test set quality. Although high coverage is better than low coverage, high coverage dose not necessarily mean that the test set has good error detecting properties.

This study is centred around hardware designs written in VHDL. However, the theoretical ideas presented are applicable not only to hardware designs but also to software systems. We have answered many question relating to X-Machine testing for hardware designs. However, this work has also identified more questions for further work.

X-Machines and testing

What is required is a method for testing or model checking an X-Machine to allow errors in the specification to be found before the X-Machine test set is generated. Also, it is possible to use the X-Machine test method to generate a higher level test set from a higher level model. For example, an X-Machine model of the different types of transfers sent over the AMBA bus could generate a test set similar to ARM's test set. A complete test set might contain test sets from multiple X-machine test sets generated from a hierarchy of X-Machine models capturing different types of behaviour.

Data Path coverage

None of the case studies in this report meet the design for test conditions. Designers *must* make a clear separation between the control path and the data path. If this is achieved, the X-Machine test methods can give total assurance that all faults in the control path have been found. Further work is needed in developing methods that will supplement to the X-Machine method that can guarantee fault coverage of the data path.

Reliable metrics

Although there are many metrics that claim to measure the quality of a test set. Many of them conflict in their opinion of how well the test set can find faults. More work is needed in measuring the quality of test methods. Also measures of device complexity are not very useful in give a picture of how difficult a device is to test. Although counting the number of states and transitions can give an indication test set size, two designs with the same number of states and transition can still take different amounts of time to test. If good methods are identified to measure to how difficult a device is to test. It may be possible to use these ideas to design devices that are easy to test.

References.

[1] http://www.estec.esa.nl/wsmwww/leon/.

[2] V.K. Agarwal and A.S.F. Fung. Multiple fault testing of large circuits by single fault test sets. *IEEE Transactions on Circuits and Systems*, 1981.

[3] R.C. Aitken and P.C. Maxwell. Better models or better algorithms - techniques to improve fault-diagnosis. *Hewlett-Packard Journal*, 1995.

[4] A. Aziz, F. Balarin, S.T. Cheng, R. Hoyati, T. Kam, S.C. Krishnan, R.K. Ranjan, T.R. Shipple, V. Singhal, H.Y Wang, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. HISS: A BDD-based environment for formal verification. *31st ACM/IEEE Design Automation Conference*, pages 454 - 459, June 1994.

[5] Tudor Balanescu. Generalised stream X-Machines with output delimited type. Technical report, Faculty of Sciences Pitesti University, 2000.

[6] K. Bogdanov. *Automated testing of Harel's statecharts*. PhD thesis, The University of Sheffeld, Jan 2000.

[7] K. Bogdanov, M. Fairtlough, F. Ipate, and C. Jordan. X-machine specification and refinement of digital devices. Technical report, Department of Computer Science, University of Sheffeld, 1997.

[8] Jorg Bormann, Jorg Lohse, Michael Payer, and Gerd Venzl. Model checking in industrial hardware design. In *Design Automation Conference*, pages 298 - 303, 1995.

[9] D. Borrione, J. Dushina, and L. Pierre. A compositional model for functional verification of high level synthesis. *IEEE Transactions on VLSI Systems*, 8(5), October 2000.[10] J. Glen Brookshear. *Theory of Computation, formal languages, automata and complexity.* The Benjamin/Cummings Publishing Company, Inc., 1989.

[11] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677{691, August 1986.
[12] David M. Calcutt, Frederick J. Cowan, and G. Hassan Parchizadeh. *8051 Microcontrollers, Hardware and Software Applications*. Arnold, 1998.
[13] Sarah Chambers. *Applying X-Machines to the retospective testing of software*. PhD thesis, University of Sheffield, Department of Computer Science, 2000.
[14] Z. Chein, L. Wei, A. Keshavarzi, and K. Roy. IDDQ testing for deep submicron ICs: Challenges and solutions. *IEEE Design and Test of Computers*, 19(2):24 - 33, March - April 2002.
[15] Jimmy Liu Chein-Nan and Jou Jing-Yang. An efficient functional coverage test for HDL descriptions at RTL. In *IEEE International Conference on Computer Science: VLSI in Computers and Processors*, pages 325 - 327, 1999.
[16] Kwang-Ting Cheng and A. S. Krishnakumar. Automatic generation of functional vectors using the extended finite state machine model. *Design Automation of Electronic Systems*, 1(1):57 - 79, 1996.
[17] T. Sun Chow. Testing software design modeled by finite state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178{187, May 1978.
[18] O. Coudert and J.C. Madre. Verification of sequential machines based on symbolic execution. In *Proc. of the Workshop on Automatic Verification Methods of Finite State Systems*, June 1989.
[19] Nigel Cutland. *Computability, An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.
[20] S. Eilenberg. *Automata, Languages and Machines*. Academic Press, 1974.
[21] Norman E. Fenton. *Software Metrics: A Rigorous Approach*. Chapman & Hall, London, 1995.
[22] F. Ferrandi, G. Ferrara, D. Sciuto, A. Fin, and F. Fummi. Functional test generation for behaviorally sequential models. In *Proceedings Design, Automation and Test in Europe Conference*, page 403, 2000.
[23] F. Ferrandi, F. Fummi, and D. Sciuto. Implicit test generation for behavioral VHDL models. In *Proc. IEEE ITC*, pages 587 - 596, 1998.
[24] T. Filkorn, M. Payer, and P. Warkentin. Symbolic verification of sequential circuits synthesized with CALLAS. In *6th International Workshop on High-Level Synethesis*, 1992.
[25] Harry Foster. Applied boolean equivalence verification and RTL static sign-off. *IEEE Design and Test of Computers*, 18(4):6 - 15, July - August 2001.
[26] M. Holcombe and F. Ipate. *Correct Systems, Building Business Process Solutions*. Springer Verlag Applied Computing Series, 1999.
[27] F. Ipate. *Theory of X-Machines and Applications in Specification and Testing*. PhD thesis, University of Sheffeld, Department of Computer Science, July 1995.
[28] F. Ipate and M Holcombe. Specification and testing using generalized machines: a presentation and case study. *Software Testing, Verification and Reliability*, (8):61 - 81, 1998.
[29] Robert B. Jones, John W. O'Leary, Carl-Johan H. Segar, Mark D. Aagaard, and Thomas F. Helham. Practical formal verification in microprocessor design. *IEEE Design and Test of Computers*, 18(4):16{25, July - August 2001.
[30] T. Kim, V.K. Chen. On comparing functional fault coverage and defect coverage for memory testing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1999.
[31] Y. K. Malaiya and A. P. Jayasumana. Enhancement of resolution in supply current based testing for large ICs. In *Proc. IEEE VLSI Test Symp.*, pages 291 - 296, 1991.
[32] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(20), December 1976.
[33] S. Mourad, J.L.A. Hughes, and E.J. McCluskey. Effectiveness of single fault-tests to detect multi-

ple faults in parity trees. *Computers & Mathematics with Applications*, 13(5-6):455 - 459, 1987.
[34] Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons Inc, 1979.
[35] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3 - 18, January 1992.
[36] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676 - 686, June 1988.
[37] D. J. Richardson and L. A. Clarke. Partition analysis: A method combining testing and verification. *IEEE Transactions on Software Engineering*, 11(12):1477 - 1490, December 1985.
[38] Marc Roper. *Software Testing*. McGraw Hill Book Company, 1994.
[39] E. Rudnick, R. Vietti, A. Ellis, F. Corno, P. Prinetto, and M. Sonza. Fast sequential circuit test generation using high-level and gate-level tech- niques. In *Proc. of DATE*, pages 570 - 576, 1998.
[40] Sadegh Sadeghipour. *Testing Cyclic Software Components of Reactive Systems on the Basis of Formal Specification*. PhD thesis, The Technical University of Berlin, 1998.
[41] Susana Stoica. Generating functional design verification tests. *IEEE Design and Test for Computers*, pages 53 - 63, July-September 1999.
[42] V Szekely, M Rencz, S Torok, and B Courtois. Cooling as a possible way to extend the usability of IDDQ testing. *Electronics Letters*, 33(25):2117 - 2118, December 1997.
[43] Sardar Tasiran and Kurt Keurzer. Coverage metrics for functional validation of hardware designs. *IEEE Design and Test of Computers*, 18(4):36 - 45, July-August 2001.
[44] E. J. Woyuker and T. J. Ostrand. Theories of program testing and application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3):236 - 246, May 1980.
[45] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366{427, December 1997.
[46] S. K. Vanak, Complete functional testing of hardware descriptions, PhD thesis, University of Sheffield, 2002.