MUTATIONS AND TESTS FOR VLSI CHIPS

by

Shmuel Katz

Technical Report #314

March 1984
MUTATIONS AND TESTS FOR VLSI CHIPS

by.

Shmuel Katz
Computer Science Department
Technion - Israel Institute of Technology
Haifa, Israel

ABSTRACT

Test generation is considered for a family of existing testing methodologies for VLSI chips. The methodologies have in common that errors on a chip are reflected as changes (called mutations) in a program-level description of the implemented algorithms. Testing criteria are developed for two types of mutants, and techniques from program verification are used to define a collection of predicates which input data must satisfy in order to be an adequate test. The actual generation of the values is considered, and it is shown under what conditions a chip which passes such a test is guaranteed error-free.
1. INTRODUCTION

In this paper test generation is considered for a family of known testing methodologies for VLSI chips implementing a collection of algorithms which are either loopless, or with loops executed a constant number of times. Such a situation occurs on a chip implementing 'machine' commands, or other service routines, where the 'outer loop' of repeated activation of the algorithms can be ignored.

Both low-level logical design testing and day-to-day acceptance testing of production chips are considered. The former involves determining whether the low-level design is a true reflection of the requirements from a higher level specification. The design is usually of sufficient complexity to make complete analysis infeasible. In the case of acceptance testing, the goal is to detect errors in the manufacture of individual chips, and the chip itself, of course, cannot be physically examined and analyzed. Thus, in both types of testing, the only way to establish reliability is to execute test cases for appropriate data. Since exhaustive testing is almost always impractical, it is generally not clear what can be claimed about a chip which has passed some non-exhaustive tests. Here it is shown that if certain assumptions are made about the degree of variance of the chip from some abstract model (i.e., about the nature of potential errors), and about the types of tests which are acceptable, test data can be chosen so that if the tests are passed, the chip is guaranteed not to have any error of the class defined. This data can be chosen once for an abstract design, and used to check any number of low-level designs or production chips.
The basic approach of the methodologies considered is to develop a program for each algorithm, which will serve as a canonical form for what is intended to occur on the chip for that algorithm. This program can be verified both with respect to any high-level specification of the algorithm, and also with respect to alternative models such as a 'machine' version, separating control and data and sending pulses between them.

The form of the branches and boxes in the flowchart representation of the program will reflect the control aspects of the machine model, while the execution boxes and tests, along with the variables, represent the data and atomic execution aspects.

The potential errors on the chip are described by assuming the existence of another flowchart, C, which corresponds to the actual computation on the chip. The types of errors which are to be treated are expressed as the kinds of differences allowed between C and the specification flowchart. These differences will define the concept of a mutation of a flowchart. Of course, this exercise will be worthwhile only if it is felt that the possible mutations capture many of the most common errors which occur on chips. We claim that this is the case for the mutations treated here, and that the framework established allows adding additional classes of mutations without altering the basic concepts.

In the following section, the definitions are made more precise and the classes of errors which are guaranteed to be discovered such a testing methodology are specified.

Section 3 shows how the conditions which the test data must satisfy can be generated. In Section 4, an example is presented applying the generation strategy using one of the methodologies from the class considered.
The basic steps of the methodology given in Section 2.1, below, are a simplified version of a full testing methodology first developed by I. Berger and M. Yoeli at the IBM Israel Scientific Center. They, however, had a different approach to the way in which the needed steps should be realized.

A top-down multi-level design methodology with connections to testability is given in [Basset and Saucier]. Their approach includes the program and machine levels, shows the connection between them, and uses these to generate tests of basic blocks. In [Min and Su] a special "register transfer language" (RTL) is defined and used for specifying algorithms, and then analyzed to generate tests, where hardware errors are 'reflected' in the RTL description.

In [Lai and Siewiorek] a general framework is presented, where the particular testing strategy, and even what is being tested, is left as a parameter, but the idea of analyzing a graph-like description of the algorithm is emphasized.

A survey of other views of chip testing may be found in [Roth].

2. PRELIMINARIES

2.1 Basic Definitions

A flowchart \( F \) is defined in the standard way, where both the functional (assignment) boxes, and the tests are termed basic blocks.

A mutation of a flowchart \( F \) is simply a change in \( F \). The mutations considered here are (a) one of the basic blocks is replaced by some other basic block (i.e., the assignments or tests are changed), or (b) the endpoint of one branch is moved so as to end at a different
point than originally. Note that (b) includes the possibility of one block being removed, by having the branch entering it 'detour' around it. This is termed a skip mutation.

The basic mutations are intended to capture an error in one of the low-level functions. The structural mutations would capture an error in the sequencing of the pulses which 'run' the low-level instructions (with the skip-mutations corresponding to a missing pulse).

The mutation factor is the number of mutations allowed. A mutant of a flowchart F is another flowchart M, which differs from F by some allowable mutation factor, that is, by applying the allowed number of mutations to F, it is possible to obtain M.

The basic mutants BM of a flowchart F have the same structure as F, and differ only by mutations of type (a), i.e., different basic blocks.

The structural mutants SM of a flowchart F differ only by mutations of type (b), i.e., have functionally identical basic blocks, but differ in some branches.

The conceptual flowchart of a chip, C, is a (nonexistent) flowchart program which corresponds to the actual executions of the chip for all inputs. In other words, if the algorithm actually on the chip were executed for every possible input, a collection of executions would be obtained which corresponds to the executions of some flowchart program, here denoted by C. (In fact, usually there are many programs which would yield the same collection of executions, and one of them is chosen arbitrarily.) At no point will C ever be explicitly described, but we express the allowable errors by requiring C to be at most a mutant of the specification flowchart F. Even though F is assumed
to be either loopless, or to contain only loops executed a constant number of times (independent of input), \( C \) may contain loops, due to a structural mutation which redirected an arc to 'create' a loop.

**Loop detection capability** is the assumption that the beginning of an infinite computation for given input values can be algorithmically determined within a fixed time after the beginning of the execution. For the loopless flowcharts treated here, the longest path can be determined. Then, if bounds are known on the time needed for the basic operations, the testing methodology may assume loop detection capability. If loop detection capability is assumed, then executing an input value which forces a non-terminating execution is one way to reveal an error. In the continuation such a capability is assumed.

As mentioned, the steps below are a simplified version of a testing methodology first developed by I. Berger and M. Yoeli at the IBM Israel Scientific Center, expressed in terms of the above definitions.

(0) Determine a (loopless) flowchart \( F \) describing an implementation. Preferably, \( F \) should be shown to reflect the high-level design of the desired algorithm, e.g., by proving correctness w.r.t. a more abstract specification.

(1) By analyzing the internal structure of the implementation of each basic block of \( F \), devise a "thorough test" which if executed on that basic block, guarantees its correctness. Techniques from finite automata theory can be used for this task ([Chow], [Roth]).

(2) Extend these thorough tests, so that there is a series of input test data for each basic block, which will have the values of the thorough test from (1) when the basic block is reached. This is done under the assumption that the conceptual flowchart \( C \) is a basic mutant of \( F \), differing only in the block being tested.
(3) Find a collection of input test data such that for each single non-basic mutant of \( F \) at least one input of the collection must yield an incorrect result, or loop indefinitely. (Recall that the structural mutants of \( F \) need not be loopless.)

(4) Execute the test data from (2) and (3) on the chip, and determine the correctness of all the results with respect to an output specification, assuming a loop detection capability. The specification is generally a predicate, but any other kind of 'oracle' which can determine whether values are correct or not, is sufficient.

Note that the requirements on the tests developed in step 3 imply that each branch of \( F \) would be passed during execution of at least one of the tests (if these tests were executed on \( F \), rather than on \( C \)). Of course, if executed on \( F \), all the tests would necessarily yield correct results, since \( F \) is either assumed correct or has been proven correct in step (0).

2.2 Assertions about the Testing Methodology

If we have successfully completed the above five steps, we now show what claims can be made about the chip. In the following section we deal with how we might achieve the requirements of, in particular, steps (2) and (3).

Lemma 1 If \( C \), the flowchart which corresponds to the actual execution of the chip, is a mutant of \( F \) with a mutation factor of at most 1, and steps (1)-(4) are successfully completed, \( C \) must be equivalent to \( F \).

Proof: If a basic block in \( C \) differed from the corresponding one in \( F \), then because of the assumption of at most 1 mutant (the "single fault" assumption), all of the control paths were identical, and the
"thorough test" for that block in step (2) would have discovered the inconsistency. Conversely, since the tests in (2) all were successful, all the basic blocks are correct, either because they were thoroughly tested, or because the assumption made in (2) of correct control paths was wrong, and by the single fault assumption, all the basic blocks are correct. Thus \( C \) could only contain a structural mutation. However, by definition the collection of inputs in (3) contains one input which would have to yield an incorrect result or loop forever if \( C \) contained such a mutation, contradicting the successful execution assumed in step (4). Thus \( C \) must be equivalent to \( F \).

Note that in order to prove \( C \) equivalent to \( F \), it is not required that \( F \) is provably equivalent to the design of the chip, or even provably correct (although these are clearly desirable). It is sufficient to assume that \( F \) will give correct results for the inputs chosen, and that \( C \) is a mutant of \( F \) with at most a single mutation.

If the single fault assumption does not hold, there is a danger that multiple mutations could "cancel out" each other for the test values chosen, but not for all other values, so that an incorrect chip could still pass steps (0)-(4).

This clearly would be a rare occurrence. It is possible to define classes of mutants for which it is impossible. First we denote by \( \psi(x, z) \), the required output specification of \( F \). A mutation \( M \) is called error-causing if \( \neg \psi(x_o, z) \) is true or the program does not terminate for some values of \( x_o \) in the single-fault mutant of \( F \) with just mutation \( M \) (denoted \( C_M \)). That is, if not cancelled out by another mutation, an error will be caused by the mutation.
Lemma 2 Consider a program $F$ with specification $\psi(x,z)$, and a mutant $C$ with up to $m$ mutations. If there is an error-causing mutation $M$ contained in $C$ such that whenever $\neg \psi(x,z)$ is true or the program loops in $C_M$, it is also true in $C$, then the tests developed for $F$ must reveal an error in $C$.

Proof: By Lemma 1, one of the input values (tests) developed for $F$ must discover the error in $C_M$ by yielding an incorrect result. That same input value must also yield an incorrect result when executed on $C$, thus revealing an error.

This lemma clearly applies, for example, if there is an error-causing mutation in $C$, so that all paths in $C$ which include the mutation do not intersect any other mutation. Thus the structural mutation seen in Figure 1 would be detected (assuming it causes an incorrect result) even though there is another mutation along the path of the original $F$ which passes $M_o$.

![Diagram](image-url)

Figure 1. A mutation detected by the methodology.
3. THE CONDITIONS FOR THE INPUT DATA

3.1 From Program to Predicates

In this section we consider some of the issues involved in choosing the collection of test data in steps (2) and (3). Two basic approaches are reasonable for this task: (1) guess values and check their appropriateness efficiently, or (2) analyze and choose appropriate test values the first time. Each approach has certain drawbacks, and a combination of the two is a definite possibility. We briefly return to this issue in Section 3.4.

In either of the above approaches, it is extremely valuable to be able to state the requirements for each test value as a predicate, generally a collection of inequalities. Then choosing correctly corresponds to finding a value which satisfies the inequalities, while checking a chosen value is simply determining whether the predicate is true for that value.

The basic aid proposed in order to generate these requirements is symbolic execution. Using this technique, we can determine (a) the relations defining a class of input values which force execution of a given path (i.e., the transfers of control will be such that the desired path is followed) called the path-condition; and (b) the changes in the variables which occur along this path — called the path-function. The use of symbolic execution can be mechanized whenever the flowchart contains no loops, or only loops executed a constant number of times (where the constant is independent of the input values). This is the case in the flowcharts considered here as modelling the correct execution, even though it is not true of all mutants. We shall use this assumption in the continuation.
3.2 Forward and Backwards Substitutions

Two alternative algorithms are used for generating the path conditions and path functions.

The first, called "forward substitution" (first used by [Deutsch]), moves through the program, at each point expressing all the possible changes from the beginning of the program to that point. The data structure used for the path function is an n-tuple of the program variables, each component containing a symbolic expression, initially the variable name itself. The path condition is a symbolic logical expression, initially the input specification of the program. Whenever an assignment \( y \rightarrow g \) is reached in the program, the component of the n-tuple corresponding to each variable in the expression \( g \) is substituted for the variable in order to construct a symbolic version of \( g \). This symbolic version of \( g \) becomes the new component of the n-tuple which corresponds to \( y \). Whenever a test box is encountered, the symbolic boolean expression is similarly computed, and this expression is added to the path condition for the 'true' branch from the test, while its negation is added to the path condition for the 'false' branch. In addition, a list of the nodes passed in the symbolic execution is 'carried' with the predicate, and is recorded at each node as its 'ancestors'. An example of applying this technique can be seen to the right of the flowchart path seen in Figure 2.

The last path condition (at the lower right) can be simplified to \( y > -2 \) (since \( y+12 \neq y+10 \) is trivially true), but is left in its original form in order to enable following the stages in the substitution.
Figure 2. A flowchart path with the stages of both substitution algorithms.
The second algorithm is known as backward substitution, (described in [Manna] and first mechanized in [King]). This method is based on passing backwards (from the end of the path), symbolic expressions which correspond to the conditions required further along the paths, and the functions computed further along the path. The path function is again represented as an n-tuple corresponding to the program variables, initialized to the names of the variables. The initial path condition is traditionally the output specification of the program, since this technique originates in systems for proving correctness, but we will begin with simply 'true'. When an assignment $y := g$ is reached in the backwards scan, the expression $g$ is substituted (as it is) in place of each occurrence of the variable name $y$ both in the n-tuple and in the path conditions. When a test is reached, the boolean expression of the test is added (as it is) to the path condition from along the 'true' branch, while the negation of the test is added to the path condition from the 'false' branch. Thus these will be passed backwards from the point where they occur in the program. The application of this algorithm to the path of Figure 2 can be seen in the figure to the left of the flowchart.

When each algorithm is completely executed, the result from each is the path condition and path function for each path from the beginning to the end of the program. Thus the choice between them has been considered a matter of taste. The forward method has been considered by some to be more intuitive, and to be easier for generating test conditions. Also, for each assignment, only one substitution in the path function must be made, while in the backward algorithm, a substitution is needed for each appearance of the variable name. On
the other hand, the backward substitution can avoid considering computations which are extraneous for the final results of the variables on a path, or for the particular logical conditions being moved backwards.

In our case, we shall use the intermediate stages of both algorithms in order to treat the possibility of mutations. Thus the forward substitution will yield the path conditions and path functions from the beginning to each point in the program (this requires only a single symbolic execution of each branch, assuming no loops), and the backward substitution will yield the conditions and functions from each point to the end of the program (again a single pass over the program branches). The forward substitution, also yields the ancestors of each node.

3.3 The Uses of the Substitutions

Once this "double" analysis has been made, it can be used in the following ways:

(a) The simple result of either direction could be used to prove the correctness of the flowchart F, w.r.t. some specification (as suggested in step (b) above).

(b) This result could also be used to choose data which ensures executing each path through F, by 'solving' the inequalities in the path condition to yield an input satisfying these conditions.

(c) The forward-substitution path conditions at each point can be used to express the requirements on input data for reaching each basic block, so that it can be tested as needed in step (2).

(d) The combined result can be used to generate the condition on the input which ensures that a structural mutation yields an incorrect result.
Informally, this is done for a structural mutation by combining the forward path conditions at a point \( x \) with the backward conditions and functions at a point \( y \), and requiring that the result obtained by this combination also satisfy the negation of the correct result for the input, or force execution of a loop.

In order to define these conditions more precisely, again some notation is required. A path function for a path from the start point of a program to a point \( i \) will be denoted as \( f_i(\bar{v}) \), a function for a path from a point \( j \) to the end of the program by \( e_j(\bar{v}) \), and the corresponding conditions by \( F_i(\bar{v}), E_j(\bar{v}) \). A structural mutation connecting point \( i \) to point \( j \) will be denoted \( i \rightarrow j \). If no loop is closed by the mutation, a path function which would include such a structural mutation \( i \rightarrow j \) is defined by

\[
m_{ij}(\bar{v}) = e_j(f_i(\bar{v})).
\]

The condition for following that path in the single-fault mutation is

\[
M_{ij}(\bar{v}) = F_i(\bar{v}) \land E_j(f_i(\bar{v})).
\]

That is, the conjunction of the condition for reaching \( i \) and the condition for following the path from \( j \) to the end of the program when \( f_i(\bar{v}) \) is the initial value.

Of course, in general each point may be reached along a number of paths, and there are a number of paths from each point to the end of the program. \( M_{ij}(m_{ij}) \) will denote any path condition (path function) which would pass through the mutation \( i \rightarrow j \), and lead to execution from the beginning to the end of the program - if that mutation occurs.

In order to identify whether a loop is closed by the mutation \( i \rightarrow j \), it is sufficient to check whether \( j \) is in the list of ancestors of \( i \). In that case, the 'continuation' conditions and
predicates for $j$, $e_j$ and $E_j$, only hold for paths from $j$ which do not (again) pass $i$. Alternatively, a condition which ensures infinite looping is acceptable. This condition guarantees that no branch out of the loop through $i$ can be followed, since the variables in the appropriate tests will never have the needed values [Katz and Manna].

Applying these formulas to the flowchart path of Figure 2, if the assignment $w = 3$ were removed (a 'skip' structural mutation), then the path function $m(y, w)$ is $e_j(f_1(y, w))$ where $f_1(y, w)$ is $(y+12, y+7)$ and $e_j$ is $(w+y, w)$. Thus, we use the $f_1$ values in $e_j$, giving $(y+7) + (y+12), y+7) = (2y+19, y+7)$. The path condition is

$$f_1(y, w) \land E_j(y, w) \lor y+7 > 5 \land y+12 
eq y+10 \land \text{true}$$

and thus is unchanged.

If the structural mutation connecting the last arc in the path back to the point before the test $y=w+3$ were considered, a loop is now closed. If an entire flowchart were considered, rather than just the one path, either the paths from the test which do not again pass the last arc (i.e., the true exit from $y=w+3$) can be used as previously to obtain path conditions and functions, or the conditions for an infinite loop can be generated, assuming a loop-detection capability during testing.

The condition $y_t \neq w_t+3$ must be invariably true, where $y_t$ and $w_t$ are the possible values of $y$ and $w$ at the point of the test. Clearly, $y > -2$, the path condition for first reaching the last arc, is part of the condition for the nonterminating path. From the forward path condition at the test, $y_t$ is initially $y+12$, and then
\[ y_t > 10 \]. The predicate \( y_t > 10 \land y_t \neq w_t + 3 \) can be shown invariant by substituting the backward path function at \( t \), \((3+y,3)\), in the predicate to express the effect of a pass through the loop, and showing that the predicate with the new values remains true (as in partial correctness proofs):

\[
[ (y_t > 10) \land (y_t \neq w_t + 3) ] \Rightarrow [ (3+y_t > 10) \land (3+y_t \neq 3+3) ].
\]

Thus, non-termination is guaranteed, if the initial \( y \) satisfies \( y > -2 \), and if this mutation occurs.

Generating the path conditions and path functions, and then the conditions and functions for each mutation is a mechanical task which clearly should not be done manually. Consider a program with \( \varepsilon \) branches (one from an assignment, and two from a test), \( n \) basic blocks, and \( m \) basic assignment blocks, where clearly \( m \leq n \) and \( \varepsilon \leq 2n \). Then there are \( n \) mutations of basic blocks, and \( \varepsilon \cdot (n-1) \) single structural mutations (each branch could be redirected to \( n-1 \) other 'targets' instead of the one in the original flowchart). The number of tests needed for a basic block will not be considered here, but is a not-large constant. Thus for the mutations treated here, there are \( O(\varepsilon (n-1) + n) \) conditions for structural or basic mutations. For the special graphs treated here, this means up to \( O(n^2) \) conditions.

3.4 Choosing Test Values

Once the predicates and functions for each mutation have been generated, the next stage is to find the appropriate test values. A sufficient condition for test values which reveal, for example, a single structural mutation \( i \rightarrow j \), may now be formulated as finding a value \( v_o \) such that

\[
M_{ij} (v_o) \wedge \neg (v_o \wedge L_{ij} (v_o)) .
\]
That is, for input \( \tilde{\mathbf{v}}_0 \), if the mutation occurs, the execution for that input would pass through the mutation, and then terminate with an incorrect result. Of course, if no mutation (including the one in question) is encountered during the execution, the program would terminate with a correct result, since the original program is assumed correct.

Finding sets of input values which satisfy the collection of inequalities is not an algorithmically easy task, and is equivalent to the satisfiability problem. However, in the context seen here, there are numerous heuristics which make the problem tenable.

If \( \mathcal{W} \) is the collection of inequalities obtained by using the incorrectness criterion (c) for all the mutations, then we can describe a straightforward schema of an algorithm for finding a test set:

\[
\begin{align*}
T & \leftarrow \mathcal{W}; \\
V & \leftarrow \{\}; \\
\text{while } & \text{ } T \neq \{\} \text{ and } \text{cond} \text{ do begin } \tilde{\mathbf{v}} \leftarrow ?; \\
& \text{select a potential n-tuple of input values,} \\
& S \leftarrow \{I \mid \mathbf{I} \in T \land \mathbf{I} = \text{true}\}; \text{ S is the set of inequalities from } T \text{ which is true after substitution of } \tilde{\mathbf{v}}. \\
& \text{if } S \neq \{\} \text{ then begin } \forall \mathbf{v} \in \mathcal{V} \{\tilde{\mathbf{v}}\}; \text{ add } \tilde{\mathbf{v}} \text{ to } V \\
& \quad T \leftarrow T \setminus S \text{ remove the inequalities now covered.} \\
& \text{end;} \\
\end{align*}
\]

This schema might not terminate if candidate values do not satisfy any additional inequalities beyond those already satisfied by previously selected values, and if \text{cond} is simply true. This
difficulty can be overcome by adding code to define \texttt{cond} as a boolean variable which is \texttt{false} when no new helpful values have been generated for some iterations. In that case, values satisfying the inequalities remaining in \( T \) could be selected either by applying a deeper analysis, or by simply requesting the user to provide values satisfying those inequalities.

The selection of candidate values can, of course, be improved beyond a purely random choice. The goal of a better selection would be to reduce the size of the test set \( V \) so that each input value would satisfy an entire collection of inequalities. This may be achieved by grouping together inequalities which arise from 'similar' mutation paths, and then using inequality solvers (or interactively providing values) to guarantee in advance that each candidate value will be successful. Note that the analysis would have to be done throughout the selection process, and not merely when no new helpful values have been generated.

In a deeper analysis of the values needed to satisfy the inequality arising from a mutation which ends at point \( j \), it is extremely helpful to consider the differences between the path functions and conditions of the unmutilated original paths through the point \( j \), and those of the mutants. In other words, the values which in the original program would not have occurred at \( j \), but which could occur because of the mutation, are the candidates for yielding incorrect results. Since the techniques for improved selecting are heuristic, can be computationally expensive, and do not improve the worst-case number of tests needed, the precise details will not be presented here.
4. AN EXAMPLE

In order to demonstrate some of the techniques of the previous sections, we consider a simple program to compute the maximum of three numbers, using as basic blocks only comparisons between two numbers, and assignments.

If the initial input values of the variables $x$, $y$ and $z$ are $a$, $b$, and $c$, respectively, then the specification of the program is that the final value of the variable $m$ must satisfy

$$(m = a \lor m = b \lor m = c) \land (m \geq a \land m \geq b \land m \geq c)$$

$m$ is one of the input values

$m$ is greater than or equal to all the input values.

The flowchart $S$ for the program is shown in Figure 3. The branches are marked by numbers '0-7, and the basic blocks by letters 'A' to 'F'.

Merely choosing inputs with no consideration of mutations can lead to a collection which is not appropriate for our purposes, even though other common testing criteria may be satisfied (see, e.g. [Yeh] for several views of program testing). For example, the pair of inputs $(x=2, y=2, z=2)$ and $(x=3, y=4, z=5)$ will force each branch of $S$ to be executed, but are extremely bad for detecting mutations. Executing these two inputs in turn on each possible structural mutation will yield correct results (i.e., $m=2$, and $m=5$, respectively) for no fewer than ten structural mutations which in fact are incorrect programs. For example, the mutations $1 \rightarrow C$ (connecting the branch 1 to block C instead of B), $3 \rightarrow E$, $4 \rightarrow F$, and $2 \rightarrow E$ all yield correct results for these inputs, but are incorrect. (The reader is invited to discover the other mutations which are not revealed by this test.)
Figure 3: A flowchart for the maximum of three numbers.
Another popular testing requirement, covering each path through the program also does not suffice here, even though in this program each branch will then be followed at least twice. Thus, even if we add to the two tests above, the tests \((x=3, y=4, z=1)\) and \((x=5, y=5, z=6)\), there are still four error-causing mutations which are not detected.

Following the steps indicated in the previous sections, we generate the forward and the backward path functions and conditions for the program. Since in this simple example, only the variable \(m\) changes, the path function can be expressed as the value of \(m\). The results of the forward and backward substitution are shown in Figure 4.

The next stage involves generating the conditions and function for the single structural mutations. Since there are eight branches and six basic blocks, with each branch originally reaching one of the basic blocks, there are five possible structural mutations for each branch, i.e., forty single structural mutations. The conditions and final values can be systematically (and, hopefully, automatically) generated. For example, the mutation \(1 \rightarrow C\), yields the two conditions

\[
\begin{align*}
(1) & \quad a \geq b \quad \{\text{to get to 1}\} \land b < c \\
(2) & \quad a \geq b \land b \geq c
\end{align*}
\]
<table>
<thead>
<tr>
<th>forward substitution</th>
<th>backward substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>predicate</td>
<td>function for the predicate (the value of m)</td>
</tr>
<tr>
<td>0 true</td>
<td>(undefined)</td>
</tr>
<tr>
<td>1 a ≥ b</td>
<td>(undefined)</td>
</tr>
<tr>
<td>2 a &lt; b</td>
<td>(undefined)</td>
</tr>
<tr>
<td>3 a &lt; b</td>
<td>b</td>
</tr>
<tr>
<td>4 a ≥ b</td>
<td>a</td>
</tr>
<tr>
<td>5 a ≥ b ∧ a &lt; c</td>
<td>a</td>
</tr>
<tr>
<td>a &lt; b ∧ b &lt; c</td>
<td>b</td>
</tr>
<tr>
<td>6 a ≥ b ∧ a ≥ c</td>
<td>a</td>
</tr>
<tr>
<td>a &lt; b ∧ b ≥ c</td>
<td>b</td>
</tr>
<tr>
<td>7 a ≥ b ∧ a &lt; c</td>
<td>c</td>
</tr>
<tr>
<td>a &lt; b ∧ b &lt; c</td>
<td>c</td>
</tr>
</tbody>
</table>

Figure 4. Forward and backward substitution for the flowchart computing the maximal value.
Path (1) leads to the function  \( m = z = c \) while (2) leads to \( m = y = b \).

As another example, the mutation \( 4 \rightarrow E \), has the condition \( a \geq b \land \text{true} \) and the function \( m = z = c \).

The conditions for discovering the mutations can now be expressed, based on equation (c) of Section 3.4.

The negation of \( \psi \) is \((m \neq a \land m \neq b \land m \neq c) \lor (m < a \lor m < b \lor m < c)\).

Since all values assigned to \( m \) are \( a, b, \) or \( c \), the only way to achieve this predicate is for \( m \) not to be the maximal value and we shall only relate to the second disjunct, i.e., take \( \neg \psi(m) \) to be \( m < a \lor m < b \lor m < c \).

For \( 1 \rightarrow C \), we require (substituting for \( m \))

\[
[(a > b \land b < c) \land (c < a \lor c < b \lor c < c)] \lor [(a > b \land b > c) \land (b < a \lor b < b \lor b < c)]
\]

which reduces to

\[
(*) \quad (b < c \land c < a) \lor (a > b \land b > c)
\]

while for \( 4 \rightarrow E \), the condition is

\( a \geq b \land (c < a \lor c < b) \).

It is now simple to see that none of the four suggested sets of input values so far suggested, satisfy these conditions, and that the mutations would thus go undetected. However, choosing a test such that \( a > b > c \) (e.g., \( (5,4,3) \)), would satisfy both of the conditions, and thus would reveal either of these mutations.

When the mutations possible in this program are examined, even this simple example, reveals some interesting anomalies. Consider, for example, the structural mutation seen in Figure 5 which would occur if branch 7 were connected to the test D, instead of F, (i.e., \( 7 \rightarrow D \)).
The only paths affected are clearly those which previously passed from 7 to END, and now continue at D.

The condition for reaching the END point becomes

\[
[(a \geq b \land a < c) \lor (a < b \land b < c)] \land c \geq c,
\]

while the original condition was \((a \geq b \land a < c) \lor (a < b \land b < c)\).

Thus the path condition is actually unchanged.

The path function is also unchanged since no additional assignments occur on the new path. Thus this mutation does not affect the correctness of the program, and is not an error-causing
mutation. We will be unable to choose input which will differentiate between a chip containing this change and one corresponding to the original program, and, in fact, such a change is "innocent" as far as the criteria given here are concerned (only if timing considerations were added, would the change be detectable).

CONCLUSIONS

Just as, e.g., the 'single stuck at one' assumption (that a single wire always sends ones), has proven useful for testing hardware designs, VLSI testing which ensures that no single mutation is present, is extremely likely to also detect other types of errors, even though this cannot be proven. Within the framework suggested, it is possible to define other classes of mutations, which capture additional common types of errors, thereby increasing the reliability of the test.

Among classes of mutations which are possible - are (a) to add one or more basic assignment blocks to the program, in order to capture the possibility of erroneous extra instructions in the machine level, (b) to add a basic test block, one of whose exits continues as in the original program, while the other acts as a structural mutation, (c) to treat double or even triple structural mutations. All of these possibilities affect only the uses of the substitutions in order to generate conditions, and for each it is not difficult to state the path condition and function for a path including such a mutation.

In the other direction, it is possible to 'make do' with the structural mutations which do not close loops (thereby removing the need for loop-detection capability during execution) or even using
just the 'skip' mutations, rather than considering the possibility of any structural mutation. These simplifications can significantly reduce the difficulty of finding a test set. Although in practice the errors in the examples we have treated have been detected by such simpler requirements of course less is guaranteed to be discovered with a weaker criterion.

ACKNOWLEDGEMENT

I would like to thank Dr. Israel Berger and Prof. Michael Yoeli for discussions which motivated this work.
REFERENCES


