DIGEST OF PAPERS

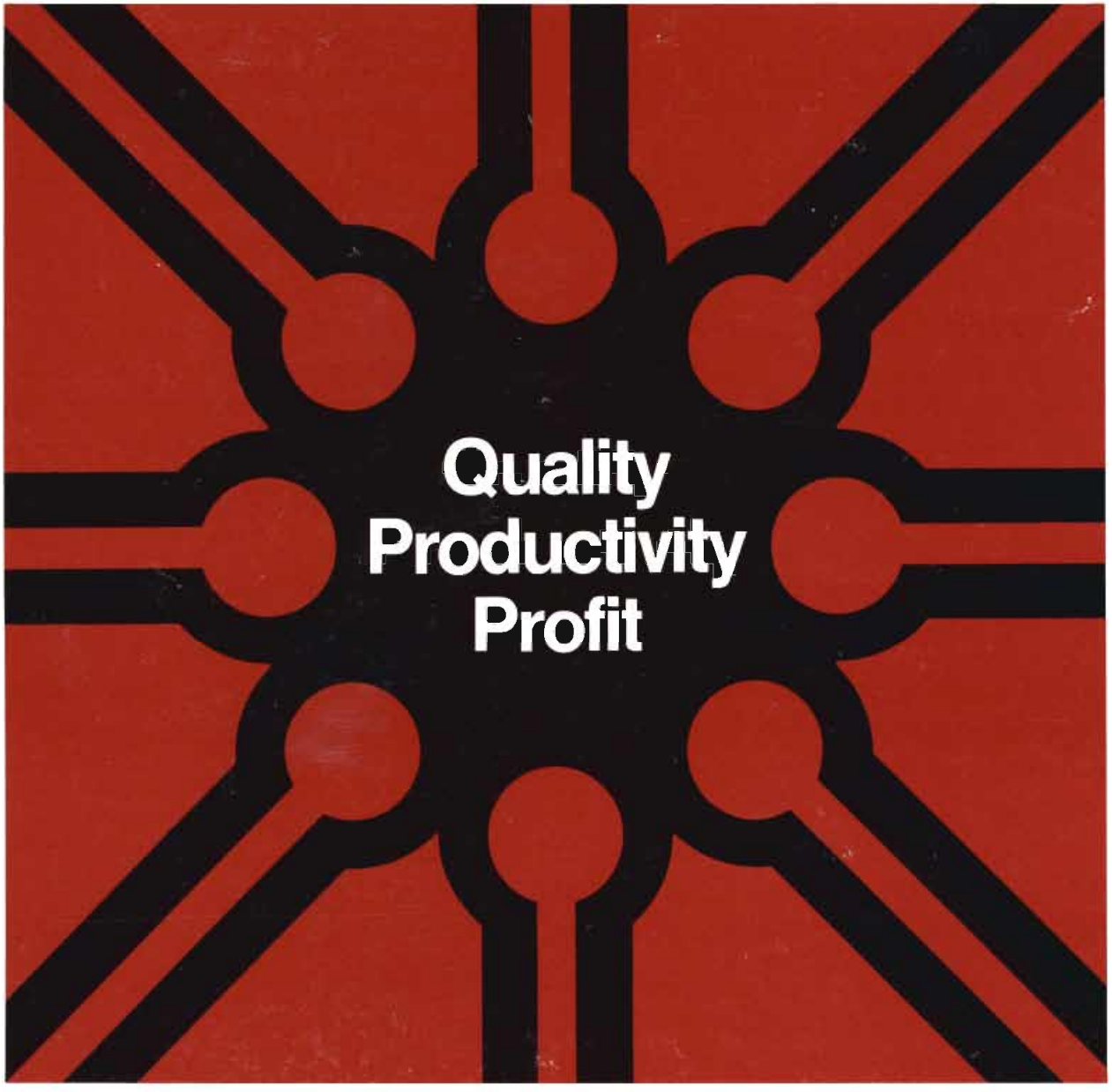
1982 INTERNATIONAL TEST CONFERENCE

NOVEMBER 15-18, 1982

Presented by: The Test Technology Committee
The International Test Formation

IEEE Catalog No. 82CH1808-5
Library of Congress No. 82-82293
Computer Society Order No. 444

Sponsored by:
IEEE Computer Society
IEEE Philadelphia Section



IEEE COMPUTER SOCIETY
INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS
COMPUTER SOCIETY PRESS

TESTING AND STRUCTURED DESIGN

Erik P. DeBenedictis and Charles L. Seitz

Computer Science
California Institute of Technology

This paper describes part of an integrated circuit testing project carried out at Caltech between 1979 and 1982. The central theme and result of the project is a language or notation for describing tests for complex integrated circuits. The evolution of this test language has been guided by many considerations, including (1) its implementation in a working, interactive test system called FIFI, (2) its fit to ideas about the architecture of high-performance test instruments, and (3) its expressivity for a design-for-testability strategy for chip designs structured in the general style presented by Mead and Conway [1].

The scope of this paper is limited, however, to a discussion of the design-for-testability strategy. The test language is not described formally here, but is used in examples with explanations that should suffice to illustrate some of its capabilities and features. A technical report on the project is available from Caltech [2].

The design-for-testability strategy discussed in this paper may appear to be somewhat more abstract than others because it is directed not at the tasks of testing combinational logic, RAMs, ROMs, state machines, and so on, but at the task of testing the *compositions* of such parts given the primitive tests for each of them. By formalizing the testability attributes of the parts and compositions of a structured design, the design of tests becomes structured also. The formalism discussed here is also an executable language. The FIFI test system is a test language interpreter that, when presented with primitive tests and the system representations discussed in this paper, can test the system.

1. Definition of a Structured System

We are concerned with the problem of testing *systems* composed of *parts*. To test such a system, it is sufficient (1) to test each of the parts, and (2) to verify the integrity of the "glue," the wiring and possibly logic that connects the parts. Figure 1 illustrates such a system. Without loss of generality we can discuss testing only one part of a system and all of its associated glue, with the understanding that the testing task is repeated for each part.

Each part may be composed of other parts, invoking the definition recursively. Ultimately, some parts will not be further divided, and these parts are called *elements*. It is assumed that tests are available for all of the elements in a system. These tests are called primitive tests, and are tests that could be applied if the element were directly accessible to the pins of a tester. The difficult part of testing a complex system, and the purpose of the work described here, is the testing of an otherwise testable part when it is embedded in a system.

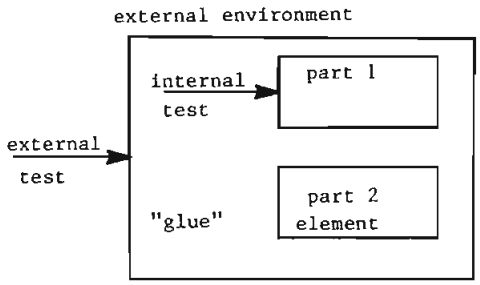


Figure 1: A Structured System

2. Actions Performed Upon Ports

The connection points of the tester, the system under test, and the parts of the system, are called *ports*. A port is a connection to an electrical node (or set of electrical nodes such as a bus or other parallel signals), and all ports connected to the same node have the same name. The *value* associated with the port is normally a voltage interpreted as being in one of at least two ranges. Each port may perform one of the following actions:

force- The port drives a value onto the node. For example, a conventional output forces a value.

feel- The port senses the voltage on a node for a *specified* value. The value is required to be static for the entire duration of the feel. If a feel is performed by a tester and a value other than the specified value is sensed at any time, an error flag is set.

undefined- A port performing the undefined action upon a node has one of two meanings: (1) the part is neither forcing nor feeling the port (tri-state condition for outputs, or the value is irrelevant for inputs) or (2) the part is forcing an unknown value onto the port (perhaps a spurious transition).

CH1808-5/82/0000/0058$00.75 © 1982 IEEE

It is important to realize that force and feel, as defined here, are dual actions that are each associated with values. Feel does not have the meaning of "sense and report," but rather "sense and compare." In a valid test, or in a valid representation of the behavior of a part, the parts connected to a port are performing complementary actions at all times. For example, when one part is feeling the value on a port, some other part must be forcing. Specifically, there are three combinations of actions that match, and are therefore legal:

1. Force matches feel *and* the values are the same. This is the normal condition in which one part sends a signal that another part receives.

2. Force matches undefined *and* the values are irrelevant. This is the condition in which one part sends a signal that is irrelevant to and ignored by another part.

3. Undefined matches undefined *and* the values are irrelevant. Usually this corresponds to one part sending an indeterminate value that is irrelevant to and ignored by another part.

An action by a part upon a port is described in the test language by a notation of the form *port operator expression*. The symbols used in the test language for operators are: < for force, > for feel, and a force followed by the expression "null," <null, for undefined. For example, clk<1 indicates that the clk port is driven to a 1 state. If < and > are visualized as arrows, the arrows point in the direction of *signal flow*, but understand that *information flow* in test language descriptions and implementations is strictly one way: from the tester to the device pins, from the pins to the parts, and so on, regardless of the direction of signal flow. The syntax of the test language enforces this rule in that (1) only port identifiers are permitted to the left of an operator, and (2) the expression on the right of the operator may contain constants or variables, but not port identifiers. Thus it is not possible with the operators described here to create a test program that senses and assigns to some variable the value on one pin in order to apply that value later to another pin. This usual feature of algorithmic programming notations is unnecessary for describing non-adaptive tests, and is incompatible with the pipelining employed in high-performance test instruments.

3. Small Examples

The time-dependent behavior of the parts and systems we would like to describe consist of a sequence of actions. The test language denotes the partial ordering of these actions with a character-based syntax. A group of partially ordered actions is called a *behavior graph*. As is usual with programming notations, the separator character ";" is used to denote sequence, and has lowest precedence. Actions separated by ";" would be parts of sequential test steps. Actions separated by "," occur concurrently.

Even "combinational" parts exhibit time-dependent behavior that must be accounted for in representing their behavior. The following behavior graph is one representation (rather conservative with respect to spurious transitions) of the behavior of a two-input AND gate with inputs A and B, and output C:

```
*[ A>a, B>b;          *[ ] indicates indefinite
   C<a&b;                  iteration
   C<null;            output becomes undefined
   A<null, B<null; ]
```

The behavior consists of an endless cycle of the four actions:

1. The two inputs become defined (stabilize, possibly after spurious transitions) with the values a and b.

2. Some short time after the inputs become defined, the output becomes defined (possibly after spurious transitions) to the AND function of a and b.

3. In a real AND gate, the output, if it was to change at all in response to an input transition, would start to change only after the input started to change. However, from the standpoint of another part connected to the output node, one would not ordinarily depend on this value being retained once the inputs have become undefined. In this sense of use, the output can be thought of as becoming undefined in anticipation of the input changes, and so it is represented as becoming undefined immediately before the inputs become undefined. The precise time relations and tolerances between actions combined with ";" may be defined by language features not discussed here.

4. The inputs become undefined.

The behavior denoted above represents the action the AND gate performs upon its ports. A test for an AND gate is the action that a tester performs upon the ports of the device. A test of the AND gate can accordingly be obtained simply interchanging all force and feel operators, which results in the following test:

```
A<a, B<b;
C>a&b;
C<null;
```

A<null and B<null are omitted because a tester has no real need or mechanism for driving the A and B ports to an undefined value. However, the C<null statement, occurring cyclically before the input changes, has the important meaning to a tester that C may not be expected to be in a defined state when the inputs are changed. This behavior graph then represents the structure of the test vector sequences that would appear in the test of an AND gate, and an actual test would consist of invoking this behavior graph as a procedure a number of times with the variables bound at the call to appropriate values. The way the AND gate test would be defined in the test language is:

```
define procedure andtest
var a b;
   A<a, B<b;
   C>a&b; C<null;
end
```

The AND gate test would be invoked as follows:

```
(call andtest a<0, b<0;  exhaustive test consists
        a<0, b<1;          of four vectors
        a<1, b<0;
        a<1, b<1;)
```

Now, this example, deliberately simple as it is, may appear to be a bit silly -- the gnat and sledgehammer syndrome. Observe, though, that the procedure defined for a complex system containing a large number of parts can be as complex as the interconnections between these parts requires, and can call other procedures for the individual parts. Thus the "parts" abstraction used in structured design as defined here is mapped into the procedural abstraction in the test language. This mapping might be described as of an "inside-out" character.

The behavior of devices with state presents no difficulties. For example, the following represents the behavior of a D-type flip-flop with ports: clk, clock input; D, data input; and Q, output.

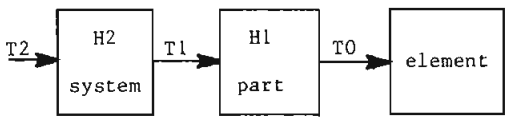
```
*[  D>x;                    input becomes valid
   clk>1;                   clock 0-1
   clk>0, D<null, Q<x; ]    three actions may happen
                              in any order
```

Signals in real systems can be generically classified into two categories: (1) signals that make clean transitions from one state to another, and (2) signals with spurious transitions. Clock signals characteristically require a clean transition, whereas inputs and outputs of combinational logic may have spurious transitions.

If one examines the cyclic actions on the same port in the behavior graph above, one observes that the D input alternates between the actions of D>x and D<null. When the D<null action is effective, port D may experience any behavior, including an arbitrary number of transitions. This notational description corresponds to the D input needing to be defined only for a short period of time surrounding the rising edge of the clock. The clk port alternates between the actions clk>1 and clk>0, with no indication that the clock input may have spurious transitions. The Q port simply assumes successive values, Q<x, and because there is no case in which Q<null, the transitions are represented as clean ones. Purely combinational logic is tolerant of spurious transitions. These spurious transitions are compatible with, for example, the LSSD [3] scan path structure that applies shifted versions of each test vector to combinational logic during the test vector loading and unloading phase. In the formalization of the behavior of a scan path, the output of the scan path would be stated as undefined while the vector is shifting.

4. Compositions

When a test is applied to the pins of a chip in order to test a particular part of the chip, the pattern of signals at the pins is altered by the composition before being applied to the part. The composition is analogous to a filter between the pins and the part. As illustrated in figure 2, the filter is composed of the entire system, excepting the part where testing is directed. The input to the filter is from the tester and the output of the filter is directed to the part being tested. Multilevel compositions correspond to the cascading of several filters.



$T0 = H1\ H2\ T2$ $\qquad$ $T2 = H2^{-1}\ H1^{-1}\ T0$

Figure 2: Filter Representation of a Test

The testing task consists of applying known tests to the elemental parts. The difficulty is that the tests must be applied from the ports of the entire system. Following the filter analogy, if the output of the last filter is given, an input to the first filter which will produce the required output, must be found. In test language terms, the inverse filter is called an *access procedure*, and takes as its argument a primitive test that is to be applied to one particular part. The concept of an access procedure can be applied repeatedly to a multi-level composition of parts. The result of the access procedure is a test that can be applied to the entire composition of parts and will result in the primitive test being applied to the part.

Of course for some notorious designs, such an access procedure may be practically impossible to compute, too lengthy to apply economically, or both, in which case the testability of the design, even though it is composed of individually testable parts, is lacking. Specific testing styles or disciplines -- LSSD is a particularly good example -- provide systematic ways of assuring that access procedures exist, are short and scale well with complexity, and are easy to compute.

There are ways of deriving access procedures for more general classes of compositions, and one of these approaches will now be outlined.

The transfer function of a filter can be described as pairs of behavior graphs. Each pair represents the actions on the external ports and corresponding actions on the internal ports. Each pair may contain variables, allowing it to describe many distinct tests.

An access procedure is defined by two behavior graphs, called an external and internal test. An

access procedure is invoked by presenting it with a behavior graph of actions to be performed on the internal ports. If the behavior graph matches the internal test, the external test can be returned as the result of the procedure. The capability of an access procedure derived in this way from a single external behavior and resultant response is limited: it can work only if presented with behavior graph very similar to its internal test. For practical testing, each part may have several access procedures that utilize its structure in different ways. When composing a system of otherwise testable parts, two things are necessary: (1) the system have an access procedure for each part, and (2) the internal test of the access procedures match the required external tests of the parts. If these two criteria are met, the tests of the parts can be translated by means of the access procedure to tests that can be applied to the entire composition.

The second criterion is the basis for design independence in the generation of testable systems. The design of a part and design of a composition containing that part can be carried out independently if the test behavior at the interface between these parts is specified.

5. Example

Consider testing an elementary part when it is composed with another part, a triple D flip-flop, as shown in figure 3. (We will later use and AND gate for the elementary part.) Note that the elementary part is completely surrounded by the triple flip-flop, so that none of its terminals are accessible from the ports of the composition.

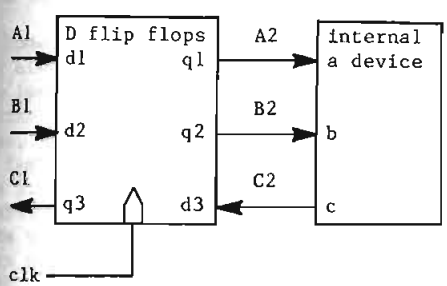


Figure 3: System Consisting of Flip Flops and an Internal Part

The behavior of the triple flip-flop with common clock can be represented by the following behavior graph:

```
*[ A1>a, B1>b, C2>c;
   clk>1;
   A2<a, B2<b, C1<c,
   A1<null, B1<null, C2<null, clk>0; ]
```

This behavior is a simple extension of that presented for a D flip-flop.

An access procedure can be derived from this program by separating the actions on the internal and external ports, as follows:

external ports:

```
*[ A1>a, B1>b;
   clk>1;
   C1<c, A1<null, B1<null, clk>0; ]
```

internal ports:

```
*[ C2>c;
   A2<a, B2<b, C2<null; ]
```

Several <null actions are meaningless and have been removed. The two behavior graphs shown above represent an access procedure. The access procedure is able to apply any test of the form shown for internal ports by applying the program shown for the external ports. The necessary computation required to generate the external test is just variable substitution. If the triple D flip-flop were directly accessible to a tester, this access procedure could be executed by applying the dual of the behavior graph shown above under "external ports:".

We can now specify that the elementary part is an AND gate. A test for an AND gate was described above. Two repetitions of the behavior graph shown above are required to match one test of the AND gate. This matching is shown below:

internal test (two repetitions)	AND gate test
C2>c;	
A2<a, B2<b, C2<null;	A<a, B<b;
C2>c;	C>a&b;
A2<a, B2<b, C2<null;	C<null;

In the example above two (different) applications of the internal test are required to perform one AND gate test. In both applications, the ports A2, B2, and C2 are matched with ports A, B, and C. In the first application, the variables a and b are matched with the variables of the same name. In the second application, the variable c is matched with a&b.

The test language code that represents the access procedure is:

```
define procedure tripleflop
var a b c;
    A1<a, B1<b;
    clk<1;
    C1>c, clk<0;
end
```

The testing of the AND gate is basically the application of the primitive tests for the AND gate to the access procedure. The timing can be abstracted away, however. The testing of the AND gate is performed as follows:

```
(call andtest a<0, b<0, c>0;    exhaustive test
              a<0, b<1, c>0;      consists of four
              a<1, b<0, c>0;      vectors
              a<1, b<1, c>1;)
```

6. Conclusions The method described in this paper allows the generation of tests for hierarchically composed systems to be approached in a structured manner. The applications of this method can cover a spectrum of design disciplines.

At one end of the spectrum, a catalog could be made consisting of parts and their testability attributes. Systems made using only the parts in the catalog would be guaranteed testable, and a test language system (such as the FIFI system developed by the authors) would perform the testing.

At the other end of the spectrum, a designer could customize the design of all the parts in his system. The testability formalism developed here would aid the designer in partitioning the design task, aid in documentation, and provide an efficient manner of testing the system.

If a system is not testable, or if the test designer does not know an efficient manner of testing a system, this method will not help. The method described here merely provides a manner of formally describing the testability attributes of a design. The designer must understand the testability attributes before they can be formalized.

7. References

[1] Carver A. Mead and Lynn A. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[2] Erik P. DeBenedictis, "Techniques for Testing Integrated Circuits," Caltech Computer Science Technical Report #4777 (PhD thesis), August 1982.

[3] E. Eichelberger and T. Williams, "A Logic Design System for VLSI Testability," *Proceedings of the 14th Design Automation Conference*, pp 462-468, 1977.