Massachesetts Institute of Technology
Artificial Intelligence Laboratory

Working Paper 280 November 1985

Test Programming by Program Composition and Symbolic Simulation

by Mark H. Shirley

Abstract:

Classical test generation techniques rely on search through gate-level circuit descriptions, which results in long runtimes. In some instances, classical techniques cannot be used because they would take longer than the lifetime of the product to generate tests which are needed when the first devices come off the assembly line. Despite these difficulties, human experts often succeed in writing test programs for very complex circuits. How can we account for their success?

We take a knowledge engineering approach to this problem by trying to capture in a program techniques gleaned from working with experienced test programmers. From these talks, we conjecture that expert test programming performance relies in part on two aspects of human problem solving.

First, the experts remember many cliched solutions to test programming problems. The difficulty lies in formalizing the notion of a cliche for this domain. For test programming, we propose that cliches contain goal to subgoal expansions, fragments of test program code, and constraints describing how program fragments fit together. We present an algorithm which uses testing cliches to generate test programs. Second, experts can simulate a circuit at various levels of abstraction and recognize patterns of activity in the circuit which are useful for solving testing problems. We argue that symbolic simulation coupled with recognition of which simulated events solve our goals is an effective planning strategy in certain cases. We present a second algorithm which simulates circuit behavior on symbolic inputs at roughly the register transfer level and generates fragments of test programs suitable for use by our first algorithm.

Table of Contents

1. Introduction

Classical test generation techniques (e.g. the D-algorithm [1]) rely on highly-optimized searches of very large search spaces. Unfortunately, modern circuits are approaching sufficient complexity that large percentages of the total design time (e.g. 30 - 50%) must go toward creating tests with these techniques. In some instances, they simply cannot be used because they would take longer than the lifetime of the product to generate tests which are needed when the first devices come off the assembly line. Despite these difficulties, *human* experts often succeed in writing test programs for very complex circuits. How can we account for their success?

We are taking a knowledge engineering approach to this problem by trying to capture in a program techniques gleaned from working with experienced test programmers. However, we are not building an expert system to directly mimic the thoughts of human experts. Instead, we are analyzing their methods to get broad hints about their reasoning techniques. In a parallel effort, we are collecting a library of solutions to specific testing problems.

This work is motivated in part by several well-recognized shortcomings of classical test generation techniques:

- These algorithms work with a gate-level description of the circuit, which causes the familiar combinatoric explosion of run time with increasing circuit size. A further problem is that circuit manufacturers often consider accurate gate-level descriptions proprietary, making it difficult for customers to obtain the information necessary to generate their own tests.
- Classical techniques have no model of the tester and can't take advantage of a tester's special purpose facilities. These methods produce a sequence of vectors and leave up to the test programmer how best to make the tester apply them.
- Classical techniques produce tests which are difficult for humans to understand and to modify. Since these tests are based on a gate-level model of the circuit's structure, operations applied during testing usually bear no resemblance to the way the circuit normally works.

Testing is clearly a very difficult problem. Yet, despite these difficulties, human experts succeed in writing effective and modifiable test programs for very complex circuits. While people aren't as good as computers at handling large amounts of detail, they do have methods for breaking up and solving testing problems that work well for many real-world circuits. The following characteristics of expert test programming stand out:

- *Experts rely on past test programming experience* (i.e. tricks of the trade).
- *They rely heavily on databook descriptions.* The operations performed by the test program usually stay within the area of legal inputs expected by the designer, so the databook descriptions are appropriate.
- *They understand how the circuit works.* For example, they know what operations are defined on the circuit's interfaces; they know which components implement which operations; and they know patterns of circuit activity relating components which are widely separated in the schematic. This information enables them to focus quickly on ways to accomplish given testing goals.
- *Testing experts are programmers too.* They share general programming knowledge with software engineers and use this knowledge to construct tests.
- *They know the features of the tester and can match them to testing problems.* This process is simple, because particular features are included in testers specifically to handle commonly-occurring and well-understood testing problems.

- *They know when and where classical test generation algorithms work well* and how to use them judiciously.

The goal of our research is to build a program that writes tests using these techniques. This paper describes our progress to date, which has resulted in the specific claims listed below:

1. Tests are best thought of as structured programs, not as unstructured sets of vectors.
2. Test programming is highly cliched, with much knowledge and even portions of test programs carrying over from one circuit to the next. We currently encode these testing cliches as combinations of goal expansion rules, code fragments and constraints on how the fragments fit together.
3. We can build an automatic test programming system using cliches which breaks down test programming problems into small subproblems. Solutions to these subproblems are kept in a library in the form of code fragments and constraints. The system then combines these fragments to create complete test programs.
4. We can build a system which behaves as if it has a global view of "how circuit components work together" by matching testing goals against simulation traces of the circuit in operation.

The first claim stems from two observations about hand-written test programs. First, they tend to be made up of small sections with well-defined purposes. For example, each section might exercise a specific component. Sectioning can increase the program's modifiability if you record which components each section uses. When the circuit is changed, only the test program sections involving modified components need be regenerated.

Second, hand-written test programs tend to be more than flat sequences of test vectors: they include simple programming language constructs, like loops and conditionals. These programs are efficient encodings of the much longer sequences of test vectors actually needed to drive the circuit.

Though the central ideas of testing cliches and using simulation in test generation are independent of this first claim, it permeates our implementation. Thus our system produces test *programs* and not test *vectors*. Our cliches contain pieces of program code and so on. We feel this is a set of ideas that works well together.

The remaining claims are discussed in later sections. We formalize our notion of a testing cliche in section 2. Section 3 illustrates a use of cliches and simple automatic programming techniques to construct test programs, and in section 4 we extend this method to use simulation traces.

2. Testing Cliches

We have said that test programmers learn methods for solving problems that work in many circuits. We call these methods testing cliches -- reusable methods for solving testing problems. Our implementation of a testing cliche has four components:

1. A pattern describing which goals the cliche can solve
2. A set of subgoals the cliche introduces
3. A set of program fragments the cliche puts into the test program. These fragments are used as direct solutions to test programming problems.
4. A set of constraints which describe how the program fragments fit together with each other and with fragments posted by other cliches.

The most important job of a cliche is to decompose testing problems into groups of simpler

problems. In practice, cliches which are used early in the goal expansion process tend to be mostly decomposition rules. If there is an associated program fragment, it usually includes just comments or simple code which glues together the large fragments created by subgoals. Cliches which are used later in the process, when the goals are simpler, tend to be mostly canned solutions. They contain small program fragments which cause the tester to do useful things. At all levels, the constraints control how program fragments, posted on a blackboard, are allowed to fit together.

Here are some examples of testing problems whose solutions we are trying to capture with cliches:

1. How to test a component
2. How to initialize a component
3. How to set up a component's outputs
4. How to move data through a component
5. How to make a component inactive
6. How to move a state machine from one state to another
7. How to take a state machine through a cycle

We work through an example in section 3.1 that shows types 1, 3, and 4.

2.1. A Simple Testing Cliche

One of the simplest testing cliches describes how to test a parallel datapath.

To test a parallel datapath:

1. **Write a code fragment to enable the datapath.**
2. **Select a vector stream that has the desired coverage characteristics.**
3. **Write a fragment to generate this vector stream.**
4. **Write a fragment to move each vector to the beginning of the datapath.**
5. **Write a fragment to pick up values as they reach the end of the datapath and move them to a place where the tester can observe them.**
6. **Write a fragment which observes the values on the primary output and compares them with expected values.**

Create constraints to link these program fragments together. For example the time a vector is generated by fragment #3 must correspond to the time it starts to move to the beginning of the datapath in #4.

This testing cliche can be used to break up the problem of how to test a parallel datapath into subproblems, most of which involve writing test program code. For example, the problem of enabling the datapath must eventually be solved by writing code which makes the tester exercise control over primary inputs to enable the datapath inside the circuit.

One subproblem that doesn't involve writing code is selecting an appropriate vector stream based on the desired fault coverage. For a parallel datapath, several choices are (a) a pair of all-0's / all-1's vectors which detects stuck at faults on each wire, (b) a two vector checkerboard pattern which detects stuck faults and bridges between wires adjacent wires, and (c) a diamond stream which detects stuck faults and bridges between any pair of wires in the datapath. These patterns for a 4-bit datapath are illustrated in figure 2-1. Since the diamond stream has the best fault coverage and its length is only linear in the width of the datapath (i.e. pattern length in vectors = 2 X datapath width in bits), we use this pattern by default.

The DATAPATH cliche is a "trick of the testing trade". It is a standard method of breaking up one test programming problem into subproblems. With a library of testing cliches similar to this one we can build a simple automatic test programmer that displays interesting behavior.

```
A)  0000        C)  0000
    1111            1000
                    1100
                    1110
B)  0101            1111
    1010            0111
                    0011
                    0001
```

Figure 2-1: *Standard Vector Patterns*

3. Test Programming with Cliches

Our test programming system runs in five phases; the first four have been implemented at present. The phases are:

1. Testing cliches decompose the problem of generating a program into subproblems. Decomposition continues until directly solvable subproblems are reached; the end result is a tree of cliche invocations.
2. Each cliche invocation in this tree can potentially contain program fragments and constraints on how the fragments must fit together. All program fragments and constraints are collected.
3. The constraints are reduced to temporal constraints on the execution times of test program statements. This yields a set of linear inequalities in two variables. We then generate a solution for this system of equations.
4. We generate code for a generic tester based on the ordering imposed on the temporal variables. Essentially, we sort the program fragments by execution time.
5. Finally, we compile the code to run on a particular tester.

The first step, decomposing the test programming problem, is handled by a set of backward chaining rules. Each rule corresponds to a single testing cliche. A cliche's pattern and subproblems form a rule's single post-condition and preconditions. Creation of program fragments and constraints is handled by a small procedural component within each rule.

There are three types of constraints: *temporal constraints* are constraints on the execution times of program statements (e.g. this statement must execute before that one), *structural constraints* are direct controls on the structure of the test program (e.g. this assignment statement is within that loop body), and *resource constraints* are annotations assigning scarce resources to different uses at different times (e.g. this circuit node has a certain value at this time and can't have any other).

The times of primitive tester actions are represented by integers. Each statement in a test program is associated with a temporal variable, and that temporal variable will eventually be bound to an integer representing the time of execution of the statement. Thus we can control timing relationships between program statements by controlling the relationships between the temporal variables. This is in turn accomplished by creating a set of equations describing the timing relationships and then solving that set of equations for integer values.

The equation solver can handle these algebraic relations between pairs of variables: $=$, $\neq$, $<$, $\leq$, $>$, $\geq$, plus AND and OR connectives between expressions. Disjunction is handled in the equation solver by a search algorithm. For convenience, we then build macros to express relations such as DISJOINT-INTERVALS and OVERLAPPING-INTERVALS between larger intervals.

Structural constraints currently allow direct and rather inflexible control over the structure of the finished test program. These constraints are satisfied directly by the algorithm which performs code generation and are currently not checked against the temporal constraints for conflicts.

Resource constraints are warnings about assignments of resources that might conflict with other assignments. We go over each pair of resource constraints that might actually conflict and create a single temporal constraint that makes sure they won't. For example, suppose A = 1@T1, meaning node A is assigned the value 1 during the interval T1, and A = 0@T2. Because a single physical node can't have more than one value at the same time, we convert this potential conflict into the temporal constraint (DISJOINT T1 T2), which in our system expands into (OR (< T1 T2) (> T1 T2)). This is an expensive process, since it can result in N^2 disjunctions for N resource constraints.

One final comment about the algorithm: the code that is generated is fit for a very simple functional tester - essentially a general-purpose computer with extra machine instructions to control and observe the I/O pins of the device under test. We expect to model testers with special purpose hardware which, for example, quickly generates vector streams or interprets bus cycles. However, we haven't yet modeled the details of any real testers. In particular, we are not addressing the problem of choosing the proper data formats or pin timings. We believe this problem is separable from the problem of planning dataflow through the circuit.

3.1. An Example

In the following example, we see the program, called TP for Test Programmer, generate a portion of the test program for the circuit in figure 3-1. This circuit is a simplified version of the MIC-2 microprocessor's datapath [2]. The inputs of the four function ALU can be driven from various places in the circuit: IN0 can be driven from a direct parallel input (node IN) or an output of the register file (the A-OUT of REG-FILE) selected by MUX-A. IN1 can be driven from the other output of the register file (the B-OUT of REG-FILE) or from a constant 0 selected by MUX-B.

The register file is a single input, dual output memory with 16 cells. The address lines AA and BA select outputs for A-OUT and B-OUT respectively. BA also controls which register is loaded from the C input, which happens on the rising edge of the clock when the enable is high. Our device model distinguishes between datapaths (represented by dark wires) and control wires (represented by lighter wires). To make the example concrete, we will say the datapaths are 16 bits wide, but the important thing is that they carry parallel values. The rest of the wires control the circuit, and their widths are shown.

We assume direct controllability of all the nodes on the left and observability of the node on the right (i.e. OUT). All other nodes (which aren't labeled) are internal to the circuit and are accessible to the tester only through intermediate components.

In this example, we generate a portion of a test program which verifies whether MUX-A, is working properly. The example is implemented, though we have changed a few details in the explanation to make things clearer. In particular, we present program code that would result if TP stopped at various times and merged several program fragments. In the current implementation the fragments remain separate until the end, when they are merged all at once. Also, the cliches and program fragments are rendered in English and pseudo-Algol to improve readability. The actual cliches and program fragments have a lisp-like syntax.

We start by asking TP to write a test program for MUX-A. The following cliche responds to this request:

> **To test a two-input mux, run one datapath test from the mux's first input (in0) to its output. Then run another datapath test from the second input to the output.**

TP maintains an agenda of independent programming tasks. Each task involves writing a

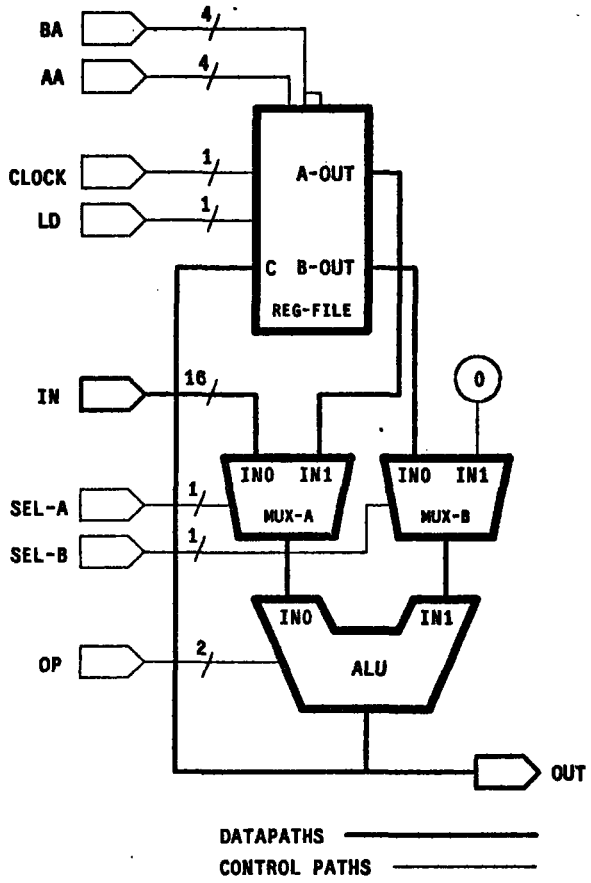


Figure 3-1: *Parallel Datapath*

section of the test program that exercises a single component or an aspect of a component's behavior. In this example, the cliche above breaks up the problem of writing a test program for MUX-A into two tasks, each of which involves testing the mux's ability to pass information from one of its inputs to its output. These top-level tasks can be solved separately. However, each may involve many subtasks which depend upon each other in complex ways. If, due to some peculiarity of the circuit, one of the top-level tasks can't be completed, then that section of the test program is simply omitted; the resulting program tests as much of the circuit as it can.

TP works on each of these programming tasks in turn. Both tasks are solved by using the DATAPATH cliche which we saw earlier. Since the tasks are similar we describe only the second one, that of writing a datapath test from MUX-A's right input (IN1) to its output.

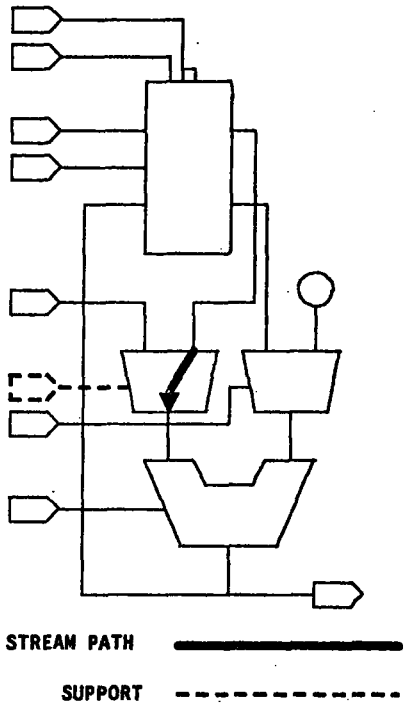
The current system uses this version of the DATAPATH cliche, which presupposes the decision to use a diamond stream.

To test a parallel datapath:

1. **Write a code fragment to enable the datapath.**
2. **Write a fragment to generate a diamond stream.**
3. **Write a fragment to move each vector to the beginning of the datapath.**
4. **Write a fragment to pick up values as they reach the end of the datapath and move them to a place where the tester can observe them.**
5. **Write a fragment which observes the values on the primary output and compares them with expected values.**

Create constraints to link these program fragments together.

First, TP works on writing code to enable the datapath. In this case, the datapath is very simple: it runs from the right input of MUX-A, through the mux, to the output. The library holds a cliche called **enable-through-mux** which specifies how to do this.[1] Because the select input of MUX-A is directly controllable by the tester, this cliche proposes the program fragment shown in figure 3-2.




```
FOR <...> DO
BEGIN
    <...>
    TESTER-ASSIGN(SEL-A, 1);   /* Occurs at TEST-TIME */
    <...>
END;
```

Figure 3-2: *Enable the datapath through MUX-A*

This loop repeatedly selects the right input of MUX-A, thus enabling the datapath. This enabling action can occur an arbitrary number of times, as indicated by the <...> placeholder for the iteration clauses. These placeholders will be filled with code from other cliches.

A temporal variable, TEST-TIME, is associated with the tester-assign statement. This temporal variable denotes the short interval during which a single test vector is passing across MUX-A. It was created by the DATAPATH cliche and passed down to the ENABLE-THROUGH-MUX cliche so that the resulting program fragments can be synchronized.[2]

The next subproblem involves writing code to generate a diamond stream. A simple way to do this is to fill an array with the appropriate sequence of test vectors, then step an index through the array, fetching vectors and putting them on the circuit's inputs. The **generate-diamond-stream** cliche implements this method with the fragment in figure 3-3.

---

[1] While this cliche is somewhat specialized, it is circuit independent. Enabling a longer datapath is handled by a cliche which breaks up the datapath into smaller parts and then constrains the parts to be enabled at the same time. Eventually, the problem is reduced to enabling through single components.

[2] There are several temporal variables not shown in the figure. For example, there are variables associated with the start and finish times of the loop.

```
DECLARE ARRAY: diamond-array INIT (diamond-pattern),
        INTEGER: data, index;
<...>
BEGIN
    index := 0;
    WHILE index < length(diamond-array) DO
    BEGIN
       data := diamond-array[index];
       <...>
       index := index + 1
    END
END;
```

Figure 3-3: *Code to generate the diamond stream*

WHILE loops include a loop prefix and postfix plus a body prefix and postfix. In this example the loop prefix initializes INDEX, the body prefix fetches the array element, and the body postfix increments INDEX. The loop postfix is unused. This cliche captures some knowledge about the capabilities of the tester and the tester language. If the tester had special purpose hardware to generate common patterns like the diamond stream, then we would include a cliche to setup and control that special purpose hardware.

The next subproblem is the most complex - moving vectors from a primary input to the beginning of the datapath (i.e. the IN1 of MUX-A). The path used and the resulting code is shown in figure 3-4.

The path is found via line justification, which involves searching backward from the mux's input to any primary input. Since we're moving a diamond pattern, the path chosen must be able to transmit parallel data, and the path shown is the only possible solution.

The code fragment is a combination of three fragments, which enable the datapath to pass through MUX-A, ALU, and REG-FILE respectively. The statement marked *a* is from the body of a loop analogous to the one in figure 3-2 except that it selects MUX-A's other input. The statement marked *b* is from a loop which enables information to flow from the IN0 of ALU to the output. The statements marked *c* come from a cliche for moving a stream through a register file. This cliche handles the address lines, the enable, and the clock. It presupposes the choice of moving the stream through register cell 0.

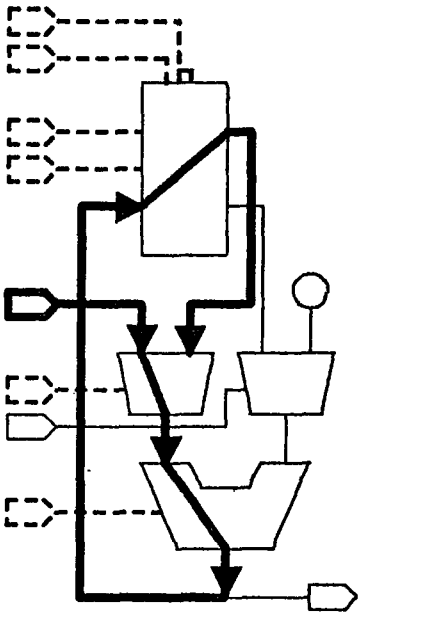
This last cliche is interesting because it contains instructions for manipulating the simple state machine which controls the register file, i.e. the clock. The cliche includes an initialization for the clock (line *c1*), and it introduces a loop body which leaves the clock in the same state at the end of an iteration as it was at the beginning (lines *c4* and *c5* together).

TP currently has no explicit representation of the constraint that a loop body must leave a state machine as it found it. The cliche, however, contains a canned solution for moving a stream through the register file that obeys this so called *cycle constraint*. With cliches such as this one, we have a way to supply solutions to subproblems which obey constraints that TP has no explicit representation for. We feel this is an important advantage of cliche-based problem solvers in the early stages of system design when the system's builders are recognizing new kinds of constraints and attempting to formalize them and represent them in the program.[3]

The final subproblem is to move the diamond stream from MUX-A's output to a primary output of the circuit. In this example, TP solves this problem using a special purpose cliche (shown in figure

---

[3]Naturally, a difficult problem is to encode enough of an unrepresented constraint in terms TP understands so that TP doesn't violate it when combining program fragments. Here, we encode a special case of the cycle constraint in terms of resource constraints involving the clock. This is not a general solution, and manipulation of state machines is an important area of continuing work.




```
c1  TESTER-ASSIGN(CLOCK, 1);          /* At INIT */
    <...>
    FOR <...> DO
        BEGIN
            <...>
a           TESTER-ASSIGN(SEL-A, 0); /* At TIME-1 */
b           TESTER-ASSIGN(OP, :noop);/* At TIME-1 */
c2          TESTER-ASSIGN(LD, 0);    /* At TIME-1 */
c3          TESTER-ASSIGN(BA, 0);    /* At TIME-1 */
c3          TESTER-ASSIGN(AA, 0);    /* At TIME-1 */
c5          TESTER-ASSIGN(CLOCK, 0); /* At TIME-1 */
            <...>
c6          TESTER-ASSIGN(CLOCK, 1); /* At TEST-TIME */
            <...>
        END;
```

Figure 3-4: *Parallel datapath from a primary input to MUX-A*

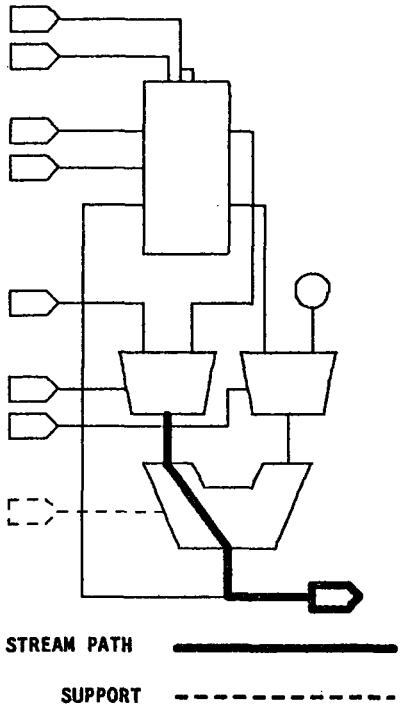
3-5) which contains a direct solution in the form of a program fragment. Using cliches like this, a human test programmer can give TP hints about how to solve problems in a specific circuit. TP can also solve this problem by searching for a parallel datapath from MUX-A to any primary output.

At this point, TP has expanded every subgoal introduced by the original DATAPATH cliche.

So far, we've emphasized program fragments at the expense of the constraints that control how the fragments are put together, but both are equally important. There is just as much testing knowledge in how the pieces of a test program must fit together as lies in the pieces themselves.

Next, TP collects all of the constraints from the cliches it has used and begins to solve them. There are about 30 temporal relations such as the one relating clock initialization time, *init*, with the clock's first use, *time-1*. The relation is *(< init time-1)*; this says that the clock must be initialized before its value is used. The TESTER-ASSIGN statement at time *init* is a small program fragment separate from the loop, and is related to the loop by the temporal relation. In fact, most program fragments tend to be small and have many relations to other small fragments.

There are several structural constraints. For example, the data variable must be read from the




```
FOR <...> DO
BEGIN
    <...>
    TESTER-ASSIGN(OP, :noop);  /* Occurs at OUTPUT-TIME */
    TESTER-OBSERVE(OUT, data); /* Occurs at OUTPUT-TIME */
    <...>
END;
```

Figure 3-5: *Parallel datapath from MUX-A to a primary output*

array before any other code inside the WHILE loop executes (see figure 3-3). This constraint is handled after constraint satisfaction by the code generation phase of the algorithm.

Finally, there are the resource constraints. The most pervasive one is that current testers are single processor machines. Therefore, all code fragments must be combined into a single test program. Currently, this constraint is built directly into the code that implements our system. Other resource constraints involve assigning values to circuit nodes. For example, the cliche for moving a stream through a register file introduces two program fragments (the clock initialization and the loop). It also creates an interval of time between the execution of the initialization (represented by *init*) and the beginning of the loop (represented by *loop*) during which the clock cannot be touched. These are called protection intervals. The protection interval is compared against every other assignment to the clock. If another cliche wants to set the clock to a different value at time OTHER, we generate this temporal constraint:

```
(or (< OTHER INIT)
    (> OTHER LOOP))
```

Once the constraints have been collected and resource constraints converted into temporal constraints, an equation solver for systems of linear inequalities produces a solution assigning each temporal variable to an integer. If no solution is possible, the system backtracks and chooses different cliches. The integers represent the execution times of the statements associated with the temporal variables; in the case of loops, they represent the times within a prototypical execution of the loop body.

The last step is to merge the program fragments in the order specified by the temporal variables.

The end result is the test program in figure 3-6, which verifies that one path through the mux can transmit parallel data without any faults.[4]

```
/* This section performs a DATAPATH-TEST */
/* on SELECT-A from IN1 to OUT           */

BEGIN
    TESTER-ASSIGN(CLOCK, 1);
    TESTER-ASSIGN(AA, 0);
    TESTER-ASSIGN(BA, 0);
    TESTER-ASSIGN(LD, 0);
    TESTER-ASSIGN(OP, :noop); /* need a peephole optimizer */
    TESTER-ASSIGN(SEL-A, 0);
    TESTER-ASSIGN(OP, :noop); /* need a peephole optimizer */
    DECLARE ARRAY: diamond-array INIT (diamond-pattern),
            INTEGER: data, index;
    index := 0;
    WHILE index < length(diamond-array) DO
    BEGIN
       data := diamond-array[index];
       TESTER-ASSIGN(CLOCK, 0); /* at TIME-1 */
       TESTER-ASSIGN(SEL-A, 0);
       TESTER-ASSIGN(IN, data);

       TESTER-ASSIGN(CLOCK, 1); /* at TEST-TIME */
       TESTER-ASSIGN(SEL-A, 1);
       TESTER-OBSERVE(OUT, data);
       index := index + 1
    END
END;
```

Figure 3-6: *Finished Program for testing half of MUX-A*

3.2. What have we done?

What is the knowledge embedded in TP and where is it? In the current system, all of the testing knowledge is embedded in the rules. Most of these rules implement testing cliches such as the DATAPATH cliche. The remaining rules implement line justification by describing how to achieve a data stream on the output of a component given a data stream on one input.

TP also has access to a schematic. Since the cliches are centered on individual components, the schematic is needed to match the names in one cliche with the names another. Finally, some simple programming knowledge is embedded in the lisp code that merges loops.

What knowledge and mechanisms were critical to solving the problem? Most important is the way cliches specify how to break up the top-level goal into subproblems. Many of these subproblems are then solved by proposing program fragments. The knowledge of good ways to decompose test programming problems comes from talking with experienced test programmers.

3.3. Limitations and Enhancements

As was indicated above, our treatment of state machines is not very general. There is not yet any explicit representation of state machines in the prototype version of TP. However, a straightforward application of our techniques will allow a test programmer to supply canned solutions for manipulating state machines in the same way that he can provide solutions for manipulating datapaths. Unfortunately, doing this in the obvious way results in an awkward encoding of the cycle constraint in terms of resource constraints. This is an important area for further work.

As with all chronological backtracking schemes, performance of the backward-chaining

[4] Our program also optimizes loop invariants.

algorithm which selects cliches is highly dependent on the order of subgoals. In our case, subgoal order makes the difference between solving the problem quickly and solving it slowly (between seconds and minutes). The problem is twofold. First, we fully expand the goal tree before checking whether any of the constraints are contradictory. This problem is straightforward to fix. Since the constraint satisfaction algorithm is already incremental - we can add new constraints in the middle and continue - we can intermingle the constraint satisfaction process with the search for appropriate cliches.

A more interesting problem is that line justification is performed in the same backward-chaining search that selects cliches. Put another way, the search for which datapaths to use is intertwined with the search for appropriate cliches. We can take advantage of very much more efficient schemes for choosing datapaths if we can integrate them with our approach.

An interesting idea for separating these searches comes from examining what test programmers actually do. We conjecture that they first select a dataflow plan, and then work out whether the timing constraints can be met. For example, to test MUX-A, one would first decide to use the datapath from IN through MUX-A, ALU, and REG-FILE to the IN1 of MUX-A, before working out any of the details about timing. By proposing all of the dataflow at once, and then determining where the datapaths intersect and which controlling state machines they share, we may be able to determine which temporal constraints really matter, and then focus our efforts on solving just those constraints. Said in AI terminology, test programmers first propose several skeletal plans (i.e. dataflow graphs) ignoring certain aspects of the problem (i.e. detailed timing). Then they choose the most promising skeletal plan and flesh it out.

Another interesting extension lies in the direction of giving TP a way of pipelining the vector streams it generates. This would involve creating timing and resource constraints between one iteration of a loop body and the next. There is an extensive literature on this subject, whose results we may be able to apply directly (e.g. see [3]).

A more fundamental problem lies in the idea of proposing parts of the solution and then searching for a way in which the parts fit together. This approach is likely to work poorly in situations were there are many plausible parts of solutions which fit together in relatively small numbers of ways. This seems to be the case when generating tests for the highly specialized VLSI components being designed and manufactured today. We feel the idea presented in the next section is a promising start on this problem.

4. Test Programming with Simulation Traces

In the previous section, we approached the problem of creating tests by first proposing pieces of the program independently, then fitting them together by solving a set of constraints. Here, we propose a new method that works by examining sets of events in the circuit that already "fit together" and determining which of these sets of events solves our testing goals. These pre-fitted sets of events are simulation traces of the circuit in operation.

For each circuit we have a library of *standard operations*. These are ways of driving the circuit from its interfaces to do interesting things. We represent standard operations as programs whose statements control the inputs of the circuit. One interesting source of standard operations is the set of operations defined on the device's interfaces, e.g. a read cycle at a bus interface. Later, we'll discuss other sources of standard operations.

Next, we simulate the circuit under the control of each standard operation and record the result. In addition to recording the events at each circuit node (e.g. node A became LOW at time 123), we record causal links between events. These links or justifications become an audit trail for asking why particular events occurred within the circuit.

When we want to write a portion of a test program, we match our testing goals, i.e. the events we wish to see occur within the circuit, against the events in the simulation traces. If we find a match, then we work backward through the justifications to the statements in the standard operation which ultimately caused the matched events. Since repeating these statements will cause the events to reoccur, we extract them from the standard operation and propose that they be part of our test program.

In figure 4-1 we contrast the idea of test programming from simulation traces with classical test generation algorithms. Classical algorithms start with what they want to happen at the test focus and work backward through the circuit to the primary inputs. Test programming from simulation traces starts at the primary inputs and "works in" to the test focus.

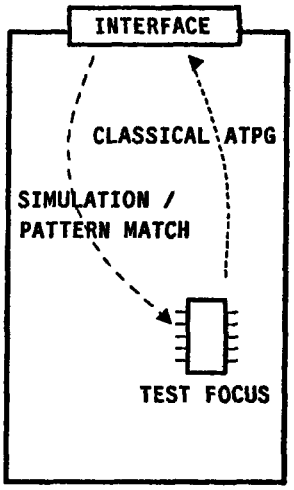


Figure 4-1: *Classical ATPG vs Simulation / Pattern Match*

A TV set provides a simple analogy. No one adjusts a TV by examining the picture, pulling out the schematic, and tracing wires back to one of the controls. There are few enough controls on a television that we can afford to *try them* individually or together and *observe* the results. We recognize a solution (i.e. a clear picture) when we see it.

This is a useful strategy if the space of valid operations at the interface is small when compared with the search involved in working backward through the circuit. We begin our search at a place where the search space is constricted - where there are few choices- and work out from there. In our domain, that place is usually the circuit's interface.

In addition, this strategy is particularly useful when there are components whose behaviors are very difficult to invert. We are currently working on an example of a microprogrammed processor that has this characteristic. The behavior of the micro-sequencer + microprogram is very difficult to invert but easy to simulate.

4.1. Creating Program Fragments from Examples

We have implemented a prototype test programming system that uses simulation traces as described above. To do this, we first introduce a new level of description intermediate between cliches (which describe how to solve testing goals) and program fragments (which describe how to drive the tester). This intermediate level describes patterns of activity which occur within the circuit. We call these patterns of activity *events*.

The most primitive event is the *constant* event, which has four components [*node*, *data*, *interval*, *justification*]. The term constant event is meant to indicate that the circuit node holds its value for some length of time: neither the data nor the justification change for the duration of the interval. A

slightly more complex event is a *change* event, which is [*node*, *old-data*, *new-data*, *instant*, *new-justification*]. Typical data values for these events are *low*, *high*, integers, and symbolic values such as *(+ ?input 1)*.

Events are hierarchical. Composite events can be defined in terms of more primitive events with constraints among them. To the temporal constraints mentioned above, we add the constraint of dependency, i.e. event A must depend on event B (through some chain of justification links). A simple example of a composite event is register-load-event, which is a change event which describes the clock rising, together with constant events on the input and output whose time intervals meet the setup, hold and delay times.

We add five mechanisms to TP to implement test programming from simulation traces. These are (1) the simulator, (2) the library of standard operations, (3) a way to convert testing goals to event patterns, (4) the event pattern matcher, and (5) the code extractor. We discuss each of these additions in turn.

4.1.1. The Simulator

The simulator we are using is a simplified version of MARS [4], an event-driven simulator which propagates symbolic values. It employs a simplifier to reduce the size of the resulting symbolic expressions. The simulator records justifications between events and has built in the assumption of node-value persistence. Our simplified version simulates devices described as flat networks, unlike MARS, which can simulate hierarchical device descriptions. We have typically used our flat simulator to model circuits at roughly the register transfer level.

The most important feature of this simulator for our purposes is its ability to propagate symbolic values. This allows us to simulate in one run the execution of many operations which are different only in their data parameters. For example, the result of simulating an adder given two variables would be the expression *(+ ?A ?B)*. For testing purposes, the important thing is that an addition operation occurred, not the specific values added, since we're going to replace the actual data with data useful for testing anyway (e.g. with something like the diamond stream).

4.1.2. The Library of Standard Operations

Standard operations are small programs for driving the circuit's inputs, usually implementing common interface operations such as bus cycles. All of these programs are currently written by hand, though eventually we may be able to derive some of them from a databook specification of the interfaces.

Applying a standard operation to the circuit results in a record of the simulation, called a simulation trace. The trace is then associated with the standard operation in the library. This process is demand driven, i.e. the simulator is called when the event pattern matcher asks for a simulation trace and this trace is not already present in the library.

4.1.3. Convert Testing Goals to Event Patterns

Testing goals are the things that cliches tell us to accomplish. For example, the datapath-test cliche tells to accomplish these goals: move information to the beginning of the datapath, enable the datapath, and pick up the information when it gets to the output. The corresponding event pattern, which is nothing more than an event with variables in it, looks like this:

```
A simple composite event
    with subevents:
        subevent-a: (constant-event ?primary-input  ?data '?input-time)
        subevent-b: (constant-event DATAPATH-INPUT  ?data '?time-1)
        subevent-c: (constant-event DATAPATH-OUTPUT ?data '?time-2)
        subevent-d: (constant-event ?primary-output ?data '?output-time)
    and constraints:
        (depends-on? subevent-b subevent-a)
        (depends-on? subevent-c subevent-b)
        (depends-on? subevent-d subevent-c))
```

This pattern describes four events. The first and last subevents involve circuit I/O pins, though the pattern doesn't specify which. The middle subevents involve the input and output of the datapath. Given this pattern, the matcher tries to find four events in the circuit involving the same data, though the events can occur at different times. The constraints in this example say that the data has to move from any primary input to the beginning of the datapath, then through the datapath and out to any primary output. If a match is found among the simulation traces, all information necessary to enable the datapath will appear in the justification links.

Currently we finesse the problem of converting from testing goals to event patterns by requiring the test programmer to provide equivalent information in both forms. This redundant information is then included in the testing cliches.

4.1.4. The Event Matcher

This matcher takes an event pattern, examines the simulation trace database looking for traces that match the pattern and returns variable binding environments which describe the matches it has found. In order to determine whether a particular primitive event (i.e. a level or change event) corresponds to a pattern, the matcher unifies the nodes, data slots, and times of the event with those of the pattern. In the case of a primitive event, unification is sufficient condition for matching.

Since events are defined hierarchically, the event matching algorithm is recursive. A composite event pattern matches by asking each inferior event pattern to generate a stream of candidate matches. The composite event pattern then filters out those combinations of inferior matches that don't meet the constraints between inferior events.[5] We have implemented this matcher and use it in the next example (section 4.2).

4.1.5. The Code Extractor

When we find a match between the events we are looking for and the events in a particular simulation trace, we work backward through the justification links to find the ultimate causes of the simulated events. In our system, these ultimate causes are always statements in the standard operation - sometimes the ultimate cause is the entire standard operation and sometimes it's a subset. We then extract the statements which caused the simulated events from the standard operation and put them together as a single program fragment. This program fragment can then be part of a test program, combined with other fragments, etc.

4.2. An example

In this example, we see TP generate a portion of a test program by matching a testing goal against a simulation trace. The circuit is the datapath from figure 3-1. As before, our goal is to test MUX-A, but we use this more complex version of the mux testing cliche:

> **To test a two-input mux, run one datapath test from the mux's first input (in0) to its output. Then run another datapath test from the second input to the output. During each datapath test, hold the value of the other input constant. Do this once for a constant value of 0, and once for a constant value of all 1's.**

---

[5] Certain kinds of composite event patterns arise frequently in testing problems (e.g. several events that must occur at the same time). We improve the efficiency of the matcher by hand coding in lisp the generators for those composite events.

To keep the example short, we will just look at the subproblem of controlling both of MUX-A's inputs at the same time.

The following standard operation is one of many in the library for driving this datapath. This operation loads two registers in the register file with dsymbolic values, and then sends these values through the alu while its adding.[6]

```
/* This stimulation pattern loads registers 1 and 2 with variables, and then */
/* adds them together. */
BEGIN
    DECLARE MACRO LOAD-REGISTER (data, address)
    BEGIN
        CONTROL-PINS CLOCK=0, LD=0, BA=address, IN=data, SEL-A=0, OP=:NOOP;
        CONTROL-PINS CLOCK=1, LD=1;
    END;
    LOAD-REGISTER(?d1, 1);
    LOAD-REGISTER(?d2, 2);
    CONTROL-PINS OP=:PLUS, AA=1, SEL-A=1, BA=2, SEL-B=0;
END;
```

Applying this operation to the circuit yields the simulation trace in figure 4-2. Starting with CLOCK, the names across the top denote circuit nodes. Simulation times are listed down the left side. The table has been broken into two halves and the times repeated. "." means that neither the value nor the value's justification has changed.

TIME	CLOCK	AA	BA	SEL-A	SEL-B
0	LOW	??	1	0	??
1	HIGH	.	.	.	.
3	LOW	.	2	0	.
4	HIGH	.	.	.	.
6	.	1	2	1	0

TIME	OP	A-OUT	B-OUT	IN	OUT
0	NOOP	??	??	?D1	?D1
1	.	.	?D1	.	.
3	NOOP	??	??	?D2	?D2
4	.	.	?D2	.	.
6	PLUS	?D1	?D2	.	(+ ?D1 ?D2)

Figure 4-2: *A Simulation Trace*

The goal of controlling both inputs of MUX-A at the same time while selecting its right input (which is circuit node A-OUT) is expressed by this event pattern:

```
A simple composite event
    with subevents:
        subevent-a: (constant-event IN    ?background '?time)
        subevent-b: (constant-event A-OUT ?data       '?time)
        subevent-c: (constant-event SEL-A 1           '?time)
    and constraints: no constraints
```

We want to find some time in the simulation trace where IN has a value we can control, A-OUT has a value we can control, and the mux's right input is selected, i.e. SEL-A = 1. Looking down the simulation trace, we see that the only time when all of these events occur together is simulation cycle 6. Thus the event pattern matches the simulation trace with the following bindings: *?time = 6, ?data = ?d1, ?background = ?d2*. Collecting all statements in the standard operation which took part in causing the matching events yields the next program fragment, which is a subset of the original

---

[6] We've introduced some additional syntax here. CONTROL-PINS is a macro for several TESTER-ASSIGN statements which occur at the same time.

standard operation. This program fragment has been surrounded with a loop so that it can be used over and over again to implement the stream of values needed for a parallel datapath test.

```
FOR <...> DO
/* This code came from a simulation of DATAPATH-ADD-VARIABLES-1 on DATAPATH */
BEGIN
    CONTROL-PINS CLOCK=0, OP=:NOOP, SEL-A=0, IN=?d1, BA=1;
    CONTROL-PINS CLOCK=1, LD=0
    WAIT 5
    CONTROL-PINS SEL-A=1, IN=?d2, AA=1;
END
```

This program fragment is a solution to the problem of controlling MUX-A's inputs. It can then be merged with fragments resulting from other subproblems to produce a complete test program. Note that there is still much work to be done, since there are still unbound variables in the program fragment. But this is a good start.

4.3. What have we done?

First, notice that we have generated part of a test for MUX-A from a standard operation that was written to exercise the alu. The matcher has found a pattern of activity in the circuit that we put there, but for a different purpose. The promise of this technique is that it can find patterns of activity that the designer put there, and use those patterns, perhaps in ways that the designer didn't foresee, to solve testing problems.

Some of the knowledge that makes this example work is bound up in the simulation models, particularly in what justification links are created. The process of walking back through the justifications and extracting the relevant portions of the standard operation is sensitive to exactly what links have been recorded. Unfortunately, deciding just what you want to say caused some event is dependent on the problem you're trying to solve. We hope that it's really dependent on the class of problem and that we can find reasonable guidelines for what constitutes relevant cause in the domain of digital testing.

Also, a considerable amount of information lies in the selection of standard operations. We feel the most promising sources of standard operations are the operations defined at the circuit's interfaces (e.g. a bus interface). These interfaces usually allow a small set of fundamentally different operations (not more than 20 or so). Each might have several parameters, but these parameters are simulated with symbolic data.

We can also use sequences of interface operations. For an instruction processor (e.g. microprocessors and certain kinds of peripheral) simple programs may prove a good source of standard operations. For example, they often contain sequences of instructions that move data into several registers, perform a computation, and move data back out to memory -- a very useful sequence when testing.

Using programs as standard operations suggests an intriguing idea. What about using previously written *test* programs? In effect, we could mine earlier versions of a test program for good ideas and for the things it did right serendipitously. This, in conjunction with annotating the program with test focus and components relied upon, may help reduce the cost of updating a test program due to ECO's or other kinds of design changes.

4.4. Limitations and Enhancements

Part of the power of this technique stems from the simulation of symbolic data. Unfortunately, the simulation of symbolic values on control signals results in multiple possible futures, which are expensive to represent. Much interesting work has been done in this area (e.g. [5]), but we are uncertain about the added problem-solving power in our domain bought by this extra complexity. For the moment, we are splitting up standard operations that would cause multiple futures.

Another weakness is the expense of symbolic simulation itself. Though the need for justification links precludes use of an efficient commercial-quality simulator (e.g. HILO, DECSIM), it is possible that, with suitable annotation on the components, we could determine after-the-fact which particular data values were crucial to the course of the simulation and which might just as easily been some other value. This is an interesting area for future development.

Two particularly promising enhancements are (1) cutting off justifications at easy-to-achieve events, and (2) temporal generalization of partial solutions found by matching. Recall that the process of extracting the relevant portions of the standard operation involves walking back through the justification links. Consider the example of a microprocessor running a program. It is conceivable that every event could be linked directly to the boot event (i.e. the first event) via the chain of program counter events. Hence, any program fragment we build will cause the microprocessor to boot before it does any useful work.

On the other hand, setting the program counter is an easy thing to do. If we're trying to control some portion of the circuit and we've walked back the justifications to an assignment to the PC, there's no reason to go further. We can stop there and insert an operation to directly control the PC.

A serious problem with merging partial solutions generated by matching simulation traces is that their timing is rigid. Notice the WAIT statement in the output of the code extractor. The statement is present because five simulation cycles elapsed between events in the original trace. Simulated events bear fixed temporal relationships with each other, thus statements in the extracted code fragment also have fixed temporal relationships. This can make it difficult to fit together partial solutions from different simulation traces, because the program fragments aren't flexible. The interesting problem is that, while some of these rigid timing relationships are important to the circuit's operation, some aren't. How can we separate the essential timing relationships from the incidental ones?

We have a promising start on this problem. The idea is to use the portion of the simulation trace that matched a testing goal as a guide for building a set of temporal constraints. The first step is to store a set of equations with each component type describing its timing. Then, starting at the matched events, we walk back over the justification links collecting temporal constraints from each component passed over. We then include the temporal constraints we've collected with constraints from other testing cliches and then use our constraint satisfaction method to solve them all together.

4.5. Relationship Between the Two Approaches

Test programming with cliches takes as input the circuit and a library of testing cliches. The cliches then decompose a programming problem to create a set of program fragments and a set of constraints which describe how the fragments fit together. If the constraints can be solved, then the fragments are combined according to the solution. Otherwise, a new problem decomposition is tried.

Within this framework three kinds of information come from expert test programmers: decompositions, fragments and constraints. The motivation behind test programming with simulation traces is to reduce the dependence on test programmers for program fragments by generating them from example simulation traces. Our main source of examples derives from the set of operations defined on the circuit's interfaces.

In the new scheme, test programmers provide decompositions and event patterns. The pattern matcher and code extractor together convert these event patterns into program fragments; the new program fragments can then be combined with other fragments in the usual way. In fact the two schemes can easily coexist with some cliches containing decompositions, fragments and constraints and other cliches containing decompositions and event patterns.

5. Related Work

Our work builds on several ideas from the research areas of testing, automatic programming, and problem-solving by analogy. There is a large testing literature, most of which covers how to generate tests efficiently in the enormous search spaces that result from using-gate level models (see [6] and [1]). As has occurred in many other fields too, a recent and significant advance involves recognizing that hierarchical device descriptions can focus reasoning and reduce search (see [7], [8], [9], [10], [11]). While we are testing our ideas at a single level of abstraction, we expect they will generalize to work with multi-level device models.

The straightforward temporal representation we use here is based on that of Joyce [12]. We have enhanced Joyce's method by first creating a partial solution based on the set of non-disjunctive constraints. Then we use this partial solution to guide the search through the space of extensions to the solution involving the disjunctive constraints. In effect, we have reversed the main steps of Joyce's method. In practice, this has increased the efficiency of handling disjuncts which we rely on heavily to represent resource constraints.

Of the work described in the automatic programming literature, our system most closely resembles Barstow's PECOS system [13] in its use of a library of refinement rules. However, our system is much simpler than PECOS. We believe this simplicity may reflect fundamental differences between symbolic programming (e.g. sorting and graph algorithms) and test programming. For example, Barstow's refinement rules map between programs, from the abstract to the slightly more detailed. Our refinement rules map between testing goals expressed in a fairly limited vocabulary: code fragments and constraints are produced as a side effect. If this strategy works well, it will be because broadly useful pieces of test programs can be written to be almost independent. Relying on an equation solver to fit the fragments together will work because of the primacy of timing relationships in testing.

The use of analogy in problem solving has been a fertile area of AI research for some time. For examples, see [14], [15]. Central to most of this work is the problem of what constitutes a pair of similar objects or situations. Our ideas are somewhat similar to Carbonell's, who treats previous experience essentially as a cache of solutions. His similarity measure is based on matching the initial part of his problem solver's goal tree with the initial part of a goal tree generated on a past problem. If they match, then the problem solver uses the remainder of the old goal tree, checking to make sure the old goal expansions are still appropriate.

Carbonell's ideas may be very useful in our domain if we can define appropriate similarity metrics between circuits, e.g. they're both memory boards, since this level of description determines TP's initial goal expansions. However, at this point in our research, we don't wish to assume that the problem solver has had previous experience with similar circuits except in the form of testing cliches. Thus our program experiments with operations at the circuit's interfaces and finds matches between the results and its testing goals.

6. Conclusion

Classical test generation techniques rely on highly-optimized searches of very large search spaces, which results in a combinatorial explosion of run times with increasing circuit size. In some instances, classical techniques simply cannot be used because they would take longer than the lifetime of the product to generate tests which are needed when the first devices come off the assembly line. Despite these difficulties, human experts often succeed in writing test programs for very complex circuits. We are trying to duplicate their success.

We take a knowledge engineering approach to this problem by trying to capture in a program

techniques gleaned from working with experienced test programmers. From these talks, we conjecture that expert test programming performance relies in large part on two aspects of human problem solving.

First, the experts remember many cliched solutions to test programming problems. The difficulty lies in formalizing the notion of a cliche for this domain. For test programming, we propose that cliches contain goal to subgoal expansions, fragments of test program code, and constraints describing how program fragments fit together. We have presented an algorithm which uses cliches to generate test programs.

Second, experts are able to simulate a circuit at various levels of abstraction and recognize patterns of activity in the circuit which are useful for solving testing problems. We have presented a second algorithm which simulates circuit behavior on symbolic inputs at roughly the register transfer level and generates fragments of test programs suitable for use by our first algorithm.

Acknowledgments

Randall Davis, Gordon Robinson, Walter Hamscher, Glenn Kramer, Narinder Singh, Dan Weld, Brian Williams and the members of the Hardware Troubleshooting Group at MIT supplied many helpful discussions and constructive comments on earlier versions of this paper.

References

1. Roth, J. P., *Computer Logic, Testing, and Verification,* Computer Science Press, Inc., Rockville, Maryland, 1980.

2. Tanenbaum, A. S., *Structured Computer Organization,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

3. Kogge, P. M., *The Architecture of Pipelined Computers,* McGraw Hill Book Company, New York, NY, 1981.

4. Singh, N., "MARS: A Multiple Abstraction Rule-Based Simulator", Tech. report HPP-83-43, Stanford University Heuristic Programming Project, December 1983.

5. de Kleer, J., *Causal and Teleological Reasoning in Circuit Recognition,* PhD dissertation, Massachessetts Institute of Technology, September 1979.

6. Breuer, M. A. and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems,* Computer Science Press, Inc., Rockville, Maryland, 1976.

7. Lai, Kwok-Woon, *Functional Testing of Digital Systems,* PhD dissertation, Carnegie-Mellon University, December 1981.

8. Genesereth, M. R., "Diagnosis Using Hierarchical Design Models", *Proceedings of the National Conference on Artificial Intelligence*, AAAI, August 1982, pp. 278-283.

9. Shirley, M. and R. Davis, "Generating Distinguishing Tests Based on Hierarchical Models and Symptom Information", *IEEE International Conference on Computer Design*, IEEE, October 1983, pp. -.

10. Davis, R., "Diagnostic Reasoning Based on Structure and Behavior", *Artificial Intelligence*, Vol. 24, No. 3, December 1984, pp. 347-410.

11. Singh, N., *Exploiting Design Morphology to Manage Complexity,* PhD dissertation, Stanford Department of Electrical Engineering, April 1985.

12. Joyce, R., "Reasoning about Time-dependent Behavior in a System for Diagnosing Digital Hardware Faults", Tech. report HPP-83-37, Stanford University Heuristic Programming Project, August 1983.

13. Barstow, D. R., *Knowledge Based Program Construction,* Elsevier North Holland, Inc., New York, NY, 1979.

14. Winston, P., "Learning Structural Descriptions from Examples", in *The Psychology of Computer Vision,* Patrick H. Winston, ed., McGraw Hill Book Company, New York, NY, 1975.

15. Carbonell, J. G., "Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition", Tech. report CMU-CS-85-115, Carnagie-Mellon University Department of Computer Science, March 1985.