Generating Tests by Exploiting Designed Behavior

Mark Harper Shirley
MIT Artificial Intelligence Laboratory
545 Technology Square
Cambridge, MA 02139
(mhs@mit-htvax.arpa)

Abstract

One of the hardest problems in digital circuit design is test pattern generation for a complex device. This is difficult in part because it requires reasoning about how to control a device whose behavior can be extremely complex. Knowledge of the specific operations that the device was designed to perform can help solve this problem. The key observation is that a device's *designed* behavior is often far more limited than the device's *potential* behavior. This limitation translates into a reduction of the search necessary to achieve planning goals. We describe an implemented program based on this idea.

1 Introduction

When reasoning about how to manipulate complex engineered systems (e.g. for test generation or diagnosis), knowing how the system is designed to behave is an important kind of knowledge to apply. This paper demonstrates how this knowledge can be used in several ways to produce an effective problem solver for one of the hardest problems in electronic design: test pattern generation for VLSI devices.

Test generation encounters several traditional AI concerns, including conjunctive planning, and remains an important, unsolved problem: existing programs are notably unsuccessful on large circuits. Human test programmers are more successful, in part because they use knowledge from many sources and a variety of reasoning techniques. Here we focus on one kind of knowledge – designed behavior – which they find important.

At the core of our test generation system is a planner based on one key observation: a device's *designed* behavior, i.e. the operations that a device is intended to carry out, is often far more limited than its *potential* behavior. For example, very few sequences of inputs to a disk controller correspond to valid commands. When solutions to planning problems can be found within designed behavior, this limitation can translate into a reduction of the search necessary to achieve planning goals.

In testing, the goals involve manipulating individual components. This work relies on what we call the Designed-Behavior Hypothesis: that every component in the device can be fully and efficiently exercised while staying within the device's designed behavior. When this is true, we can reduce the search necessary to generate tests. Our program successfully generates tests in situations where this assumption is valid, and (quickly) fails to generate tests where it is not. An example is shown which illustrates both cases.

We represent the space of designed behavior with a **behavior graph**, a structure that describes how data is supposed to flow through the device. We generate these graphs by simulation of the operations defined at the device's interfaces, e.g. the instruction set defined for a processor. The list of operations is assumed to be complete.

The planner operates by matching a testing goal, like loading a register with a certain value, against the behavior graphs and specializing successful matches to create solutions. This planning method is most appropriate for devices whose command interfaces are narrow in the number of distinct operations they admit, since each operation requires the non-trivial effort of creating a behavior graph. In the domain of digital circuits, we believe the method is appropriate when generating tests for many of the building blocks of microprocessor systems (e.g. processors, UART's, graphics controllers, and the like).

The next section gives an overview of the test generation problem, and section 3 describes the consequences of several characteristics of this problem for representation and reasoning strategy. Section 4 introduces a simple processor which provides examples throughout the paper, while section 5 contains the details of our method. The method's performance on the processor is shown in section 6. Finally, a simple account of processor design is presented in section 7, which provides justification for the performance achieved.

2 The Task

We are interested in the task of test generation for VLSI components. This task is part of the quality control phase of circuit manufacturing where we verify that the physical circuit in fact produces the designed behavior. Test generation takes a design description for a circuit and produces a set of tests. Taken together the tests must specify sets of inputs and predicted outputs such that, when the inputs are applied to the circuit and all outputs match the predicted values (i.e. the circuit passes all tests) we can be confident the circuit was manufactured correctly.

Tests are traditionally created by partitioning the design into components and generating a test for each component by (1) working out how to test the component as if it were alone, and (2) working out how to execute that test within the context of the larger circuit. The second problem is conjunctive planning, since

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology and at GenRad Inc. Support for the laboratory's Artificial Intelligence research on digital hardware troubleshooting is provided in part by the Digital Equipment Corporation, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505.

it involves achieving multiple, potentially-interacting goals: we have to work out how to manipulate the circuit's primary inputs to cause a desired pattern of activity around a component.

The classical approaches (e.g. [Roth66]) use a combination of heuristic search and constraint propagation using gate-level representations of the circuit's structure and behavior. Current runtimes are excessive for anything other than combinational circuits and simple state machines, and we think this is a direct consequence of the use of gate-level representations. More recent research (e.g. [Singh85]) applies similar search techniques to hierarchical models of device structure and behavior, yet still does not take advantage of knowing the designed behavior. For complex circuits, the human experts remain much more successful than any existing test generation program, and the next section describes several ways – inspired by what the experts do – of limiting search.

3 Characteristics of the Problem Domain

In this section we describe two characteristics of the problem domain and the effects they have on our choices of representation and reasoning strategy.

3.1 Purposefully Designed Systems

Teleology, or the notion of purpose, has been used in several programs for reasoning about physical systems. In [DeKleer79], for example, the knowledge that a device had some purpose, even though unspecified, was sufficient to aid in determining how an analog circuit worked. We require more information than this; we need to know what that purpose is, i.e. the tasks the system is supposed to accomplish. Our program then uses this information to focus on solutions to testing problems.

Figure 1 illustrates the relative sizes of solution spaces that different test generation algorithms search. The sets contain possible states of a circuit, i.e. the different ways of consistently assigning values to the circuit's nodes. If the components comprising the circuit are disconnected, then there is no constraint between their states, and the set of possible circuit states is the cross product of the sets of individual component states. Connecting the components together makes many potential states inconsistent, resulting in a smaller set of possible states. This is the space that a planner with perfect ability to propagate constraints could search. Since local constraint propagation techniques are incomplete in their ability to detect global conflicts, algorithms using these techniques search a somewhat larger space and backtrack when inconsistencies are found.

Within the space of consistent states of the circuit lies the space of behaviors the circuit was designed to perform. This is the space our test generator searches. The ratio of designed behavior to possible behavior depends upon the particular circuit, but the more specialized and complex the circuit, the smaller it tends to be. [1]

[1] In the case of the MARK-2 processor, this ratio is on the order of $2^{2^{200}}$. Naturally this number says nothing about how the spaces are searched nor about the frequency and distribution of solutions within the spaces. We haven't yet done the performance measurements for some typical circuits that would yield these numbers. But, as suggested by the performance of our method on a microprogrammed processor, we expect our method to be a useful addition to the spectrum of techniques helpful for solving test generation problems.

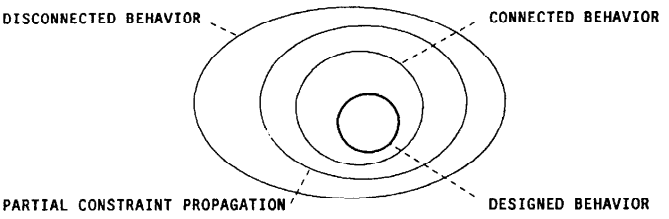

Figure 1: Relative sizes of sets of circuit states

We explicitly represent the space of a circuit's designed behavior with the behavior graphs mentioned above, which essentially are annotated simulation traces. For efficiency, circuit operations are parameterized, and the corresponding behavior graphs contain expressions with variables in them, e.g. a READ cycle is described in terms of a symbolic address and symbolic data.

The point of the behavior graph representation is this: contained within the set of graphs are many patterns of activity around every component in the circuit. "All our planner must do" is find the patterns of activity that are useful in testing. We assume the designer can provide a complete list of operations defined for the circuit's interfaces. Because this list is complete, the behavior graphs generated from them represent all normal circuit behavior. Hence, failure to find a useful pattern of activity indicates that there is no solution within the normal operations of the circuit.

3.2 Extreme Complexity

A second and dominating characteristic of this domain is the extremely complex behavior of modern VLSI components. Algorithms which manage this complexity are essential. One way to do this is to identify useful abstractions[2].

The traditional partitioning of the problem into two parts, one that focuses closely on a single component and another that involves the rest of the circuit, suggests that we might usefully build our problem solver in two parts, each part potentially working at a different abstraction level. And this is what we do: a micro-planner is responsible for solving conjunctive goals around individual components, and a macro-planner is responsible for taking the output of the micro-planner and then controlling grosser aspects of the circuit state, namely its data registers.

We must describe the circuit structure and behavior for these two parts of our program. For simplicity's sake, the prototype system models the circuit structure as a flat network of components, roughly at the level of the block diagrams in databooks. [3]

The system uses four levels of behavioral description. The simulation models for the individual components are the foundation. We only require them to predict outputs from inputs,

[2] In fact, because the circuits are designed, it is not surprising that there are abstractions readily available; they are the abstractions used by the designer.

[3] Note that, although the model is flat, it is considerably above the level of individual gates.

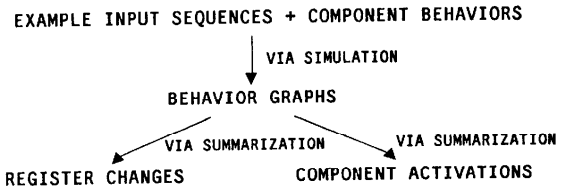

Figure 2: Behavior Representations

so we can implement them directly as Lisp functions. The next level is the set of behavior graphs mentioned above. The final two are summaries of the behavior graphs. The **Component Activation Summary** links component operations to the behavior graphs which contain one or more instances of them. For example, if the simulator notices that the ALU performs an add sometime during an instruction, then the ALU's add operation is linked to the behavior graph generated by simulating that instruction. The **Register Transfer Summary** captures the cumulative effects each behavior graph has on the circuit's internal registers. Figure 2 shows how the various behavior representations are generated.

3.3 Summary of the Domain Characteristics

When reasoning about VLSI circuits, two observations stand out: first, that they are engineered artifacts, designed to accomplish a purpose, and second, that their behavior can be extremely complex. The first observation gives us a way to overcome the difficulties resulting from the second. These observations have several implications which together guide us to the strategy of explicitly representing the space of designed behavior with behavior graphs and of searching those graphs for solutions to testing goals.

We recognize that test generation is a very difficult problem: partial solutions are likely to be the norm in this field for many years to come. Consequently, rather than trying to build a test generator which solves all problems, we are aiming for one which solves some of the problems quickly – and just as important – which fails quickly on the rest so that other methods can be applied. Naturally, it is undesirable for a test generator to fail, but it is much worse for it to run for a large and unpredictable amount of time with no result either way.

Our method quickly and exhaustively searches the circuit's designed behavior for solutions to testing goals. If a solution is present, then our program will find it, and it does so quickly because the search space is limited. To the extent that the Designed-Behavior Hypothesis is true, i.e. that efficient tests for components exist within the designed behavior, then our method is an effective test generation strategy. The example in section 6 shows the performance of our program on a simple processor.

The question of what the designer should do when the test generator fails can be answered by stepping back from the test generation problem and considering its place within the larger context of circuit design and manufacturing. Test generation is but one part of the design task. It is possible to trade off ease of testability against performance and reliability considerations. Thus, if generating tests for a given circuit requires too much effort or is impossible, then the designer can consider modifying the circuit to make the problem easier. Any redesign can be accomplished using Design for Testability techniques.

The next section introduces a simple processor which we use as an illustration throughout the rest of the paper, and the section after that describes the details of our method.

4 The MARK-2 Microprocessor

The circuit shown in figure 5 is a simple horizontally microprogrammed processor. It is a slightly modified version of the fictional MAC-1 processor presented as a teaching example in [Tanenbaum84]. The central portion of this circuit is a 16-bit-wide datapath; the righthand portion is a sequencer, including a ROM holding 80 lines of microcode implementing 23 instructions; and the lefthand portion is a RAM. The address and data busses and their associated signal lines are this circuit's primary inputs and outputs. The internal nodes are not directly accessible and must be controlled indirectly through intermediate components.

5 The Details

Our method has three parts: (1) a preprocessor for creating the behavior graphs, (2) a micro-planner for finding patterns of activity among the behavior graphs which solve testing goals, and (3) a macro-planner for managing global aspects of the circuit's state which aren't captured in the behavior graphs. We discuss each part in turn.

5.1 Preprocessor

The purpose of the preprocessor is to create a database of behavior graphs which describe how information flows through the circuit. We do this using simulation, although it could be done in other ways (e.g. by observing a working version of the circuit). We use simulation because it can easily create behavior graphs containing symbolic data, thus reducing the number simulation runs (or observations) we need. For example, suppose we're interested in the ADD instruction of the MARK-2. Rather than simulate the instruction once for each combination of numbers the processor could add, we need only simulate the instruction once, using variables for data.

The inputs to the simulator are the complete set of circuit operations collected from the databook, plus perhaps some interesting sequences of operations provided by test experts. We represent these simulator inputs as programs, because that works well with another approach to test generation discussed in [Shirley85].

We divide the circuit state into two components: **control state** and **data state**[4]. Control state is initialized the same way for each simulation run, while data state is initialized with variables. For example, the program counter is initialized to ?PC and is left by most instructions holding the expression (+ ?PC 1).

The simulator itself is event-driven, where an event is a circuit node changing its value. Its essential features are (1) propagating and simplifying symbolic expressions and (2) recording justifications between events. These justification links form a flow graph which is one component of the behavior graph. The expressions

[4]This division depends on the level at which the circuit is described, but for any given level it is fairly clear and generally corresponds to the division of controller and datapath. For the MARK-2, the control state is the microprogram counter and the micro-instruction register, and the data state is everything else (e.g. the accumulator and the program counter).

```
A composite event                              Specific Patterns
  with subevents:                              INPUT-1   INPUT-2
    OP      = :PLUS    AT ?Time                    0     65535
    INPUT-1 = ?Value-1 AT ?Time                65535         0
    INPUT-2 = ?Value-2 AT ?Time                    1     65535
    OUTPUT  = ?Output  AT ?Time                    2     65535
  and constraints:                                 4     65535
    (and (controllable? ?Value-1)                  8     65535
         (controllable? ?Value-2)                 16     65535
         (observable? ?Output))                <more patterns>
```

Figure 3: *An event pattern*

describing the values of nodes are functions of the values on circuit inputs during the simulation run and the initial states of the data registers. The variables in these expressions are marked to say where they originated, consequently, the expressions form another, much simplified, flow graph linking the values at each node back to primary inputs. The behavior graph is the union of these two flow graphs.

5.2 The Micro-Planner

The micro-planner's task is to take a test for a single component, expressed in terms of the component's I/O pins, and to rewrite it in terms of the circuit's pseudo-inputs and outputs (i.e. primary I/O pins plus data registers). The core of this planner is a matcher which takes component tests and finds instances of them in the behavior graph database. If successful, this planner produces a sequence of inputs for the circuit which will do most of the work needed to cause the test to occur within the circuit. The remainder of the work is done by the macro-planner.

Where do the component tests come from? We get them from two sources: for standard combinational devices and sequential devices, we ask the domain experts. For other combinational devices, we use a conventional test generator. In principle, we could also use a recursive application of our test generation method, though we have not yet tried to do this.

Component tests are expressed in two parts: (a) descriptions of prototypical operations that the component must be able to perform (e.g. a register must be able to load data) and (b) actual test data (e.g. have the register try to load these numbers). We provide a rich language for describing prototypical operations; the actual data are just sequences of numbers.

The pattern in figure 3 describes a prototypical add operation of the MARK-2's ALU. The four subevents describe the relations between place, value and time that define the ALU's add operation. The additional constraints further specialize this pattern to match only instances of add operations that are useful for testing the ALU. In order to help, the expressions on the two input nodes must be controllable, i.e. they must be invertible functions of two distinct primary inputs. This allows us to apply test data to the ALU by transforming the data using the inverse functions, then feeding the inverted data into the two primary inputs. As the inverted test data moves through the circuit to the ALU, it is transformed back to the desired value, thus exercising the ALU properly. Similarly, there must be a output of the circuit which is an invertible function of the ALU's output. With that we can determine what actually came out of the ALU, by applying the inverse function to data that we observe at the circuit output.

The Component Activation Summary from section 3.2 allows

```
   Time      OP        INPUT-1    INPUT-2   OUTPUT
-------------------------------------------------------
   98        :PLUS     ?Addend    ???       ???
   99        .         .          .         .
   100       .         .          .         .
-> 101       .         .          ?AC       (+ ?Addend
?AC)
   102       .         .          .         .

    ??? indicates an unknown value
    .   indicates an unchanged value (i.e. ditto)
    ->  indicates the match
    ?Addend = Contents(RAM, ?ADDR)
```

Figure 4: *Portion of ADDD's behavior graph (shown as a table)*

the matcher to divide the set of behavior graphs into relevant and irrelevant subsets depending on whether the component "did anything interesting" during a particular behavior. The matcher then tries each potentially relevant behavior graph in turn until it finds a situation which meets all of the constraints. The matching process itself is straightforward unification augmented to check the additional pattern constraints.

The matcher finds numerous places where the MARK-2's behavior graphs match the subevents of the pattern in figure 3, i.e. when the processor is "put through its paces," there are many situations in which the ALU performs an addition. But only one of these potential matches meets the additional restrictions needed for it to help test the ALU. This match occurs in the behavior graph for the ADD-Direct-from-memory instruction (ADDD), which increments the value of the accumulator by an amount stored in main memory. The relevant portion of this graph appears in figure 4, displayed in tabular form. In all of the other potential matches the ALU is either incrementing the program counter or the stack pointer: in all of these cases, one of the ALU's inputs is a constant, and thus is not controllable.

Recall that our purpose here is to apply specific test data to the ALU. An additional result of matching an event pattern against a behavior graph is a variable binding environment. The bindings resulting from matching the pattern in figure 3 against the behavior graph in figure 4 are:

```
?Time     ==  75
?Value-1  ==  Contents(RAM, ?Addr)
?Value-2  ==  ?AC
?Output   ==  (+ Contents(RAM, ?Addr) ?AC)
```

The program can then back-solve the equations in this binding environment to determine how it must set up the circuit's registers in order to apply particular values to the ALU's inputs. In this case, one addend (i.e. ?Value-1) must be in the RAM at some address which can be specified later, and the other addend (i.e. ?Value-2) must be in the accumulator.

At this point the micro-planner has now identified the ADDD instruction as the one to use to test the ALU's addition operation. Moreover, it has also determined how the data registers must be set up so that the ADDD instruction can do this job. It is now the job of the macro-planner to work out a sequence of circuit operations which will initialize the data registers properly.

5.3 The Macro-Planner

The purpose of the macro-planner is to generate sequences of operations (in this case sequences of instructions) which set up the circuit's state so that the single test operation found by the micro-planner will start with the right data. Since a test operation will sometimes leave important data in one or more registers (e.g. ADDD leaves the ALU's output in the accumulator), a second job of the macro-planner is to work out how to move results to one of the circuit's outputs where it can be observed.

The macro-planner searches through the space of instruction sequences to find one which moves data around properly. The planner is implemented as straightforward best-first search with a metric based on the sequence length and the amount of useful data contained in the registers. The operator descriptions (i.e. descriptions of the effects of instructions) are contained in the Register Transfer Summaries, which are generated by our system from the behavior graphs. [5]

An example of the end product of all of this is the sequence of circuit operations below. The micro-planner chose the ADDD operation for testing the ALU and provided information for the macro-planner, which subsequently included three operations to set up circuit state and two operations to observe it.

```
Setup inputs ----> 1: (PRELOAD-MEMORY ?AC ?ADDR)
      "            2: (LODD ?ADDR)
      "            3: (PRELOAD-MEMORY ?VALUE ?ADDR)
Test ------------> 4: (ADDD ?ADDR)
Observe outputs -> 5: (STOD ?ADDR)
      "            6: (OBSERVE-MEMORY (+ ?VALUE ?AC) ?ADDR)
```

6 Performance on an Example Circuit

Figure 6 shows the fault coverage that our prototype system achieves with the MARK-2 processor[6].

The program was able to generate tests for the highlighted components, while it failed to do so for the others. Note that the program can quickly work out how to test virtually all of the datapath, which includes more than half of the physical circuit, since data paths are typically much wider than control paths. [7] Our program also quickly fails on the controller, an area which the expert says is partially testable with effort, but really should be modified. Thus our program's failure points out areas of the circuit that a redesign program should focus on.

It is also important to realize that the program succeeds in testing the datapath despite the fact that the datapath is controlled almost completely by 80 lines of intricate microcode. Part of the program's success can be attributed to its use of abstract descriptions of the circuit's behavior (e.g. while controlling the register state). The program generates these abstract descriptions from the designed behavior. Also, the use of simulation and pattern matching is critical: reasoning backwards through the microcode sequencer is very difficult, yet this is *exactly* what goal-directed planners and classical test generation algorithms would try to do. A clear advantage of our method is that it can work forward through extremely complex components using simulation (a well-behaved reasoning method) to solve problems involving the components on the other side.

[5] We include two special operations describing capabilities of the tester hardware. These are the preload-memory and observe-memory operations which write data into and read data from the RAM.

[6] This circuit has 16 components and 91 nodes. Each simulation run is approximately 50 clock cycles long, taking about 10 seconds of real time on a Symbolics 3640. Generation of prototype tests, i.e. tests with symbolic data, for this circuit takes 1 minute, including both the time taken for successfully creating tests for some components and failing to do so for others.

[7] The program was only partially successful with the ALU and the shifter. At first this seemed odd, since the ALU is perhaps the most easily controlled component in the entire system. However, it turns out that with the current instruction set, there is no simple test for the boolean-AND operation of the ALU, because that operation is used only for masking. The situation with the shifter is similar. Generating tests for these components requires a case analysis which we have not yet implemented.

7 Explaining the Performance

Why is there a radical difference in coverage between the datapath and the controller? We conjecture that the performance of our method depends on the relationship between the circuit operations and the operations of the components that implement them. In particular, the more "direct" the relationship between individual component operations and circuit operations, the better our method performs. For example, the ADD operation of the ALU very directly implements part of the ADDD instruction of the processor, and consequently, our method is easily able to solve testing goals involving that ALU operation.

We believe the directness of the mapping between component and circuit operations varies over different parts of the circuit, and that this variation is a direct result of the design process. Consider the following account of processor design: a designer starts with a specification for an abstract machine which includes the programmer accessible registers and the instruction set. His job is to implement this abstract machine in hardware while meeting myriad performance, reliability, and cost constraints.

The specification can be viewed as a set of dataflow graphs, one for each instruction, describing how data is transformed as it moves from register to register. Naturally the designer can't implement these dataflow graphs directly in hardware: without some sharing there would be a wasteful duplication of functionality. So he looks for ways of merging the graphs together.

To do this, he adds to the graphs components which perform identity transformations. For example, he might insert a register in one graph to shift some of its operations later in time, thereby allowing components to be shared with another graph via time-multiplexing. In another situation, he might introduce identity boxes into two graphs and implement them using a single multiplexor. When the flow graphs fit together, the designer collects all the control signals from all the graphs and creates a finite state machine (FSM) to provide these signals at the right times. This FSM is implemented using any one of the well-known methods (e.g. with a PLA).

By this account, the way the datapath is designed (incremental refinement and merging) is very different from the way the controller is designed (stylized implementation of a state machine). The availability of components which directly implement large portions of individual processor operations (e.g. ALU chips), plus the fact that the merging process doesn't normally change existing components (it just adds new identity boxes), means that many datapath component operations tend to "very directly implement" circuit operations. The state machine design process, however, need yield no such simple relationships. The behavior of the whole controller (a state machine) is very different from the behavior of any controller component (e.g. a register, ROM, or MUX).

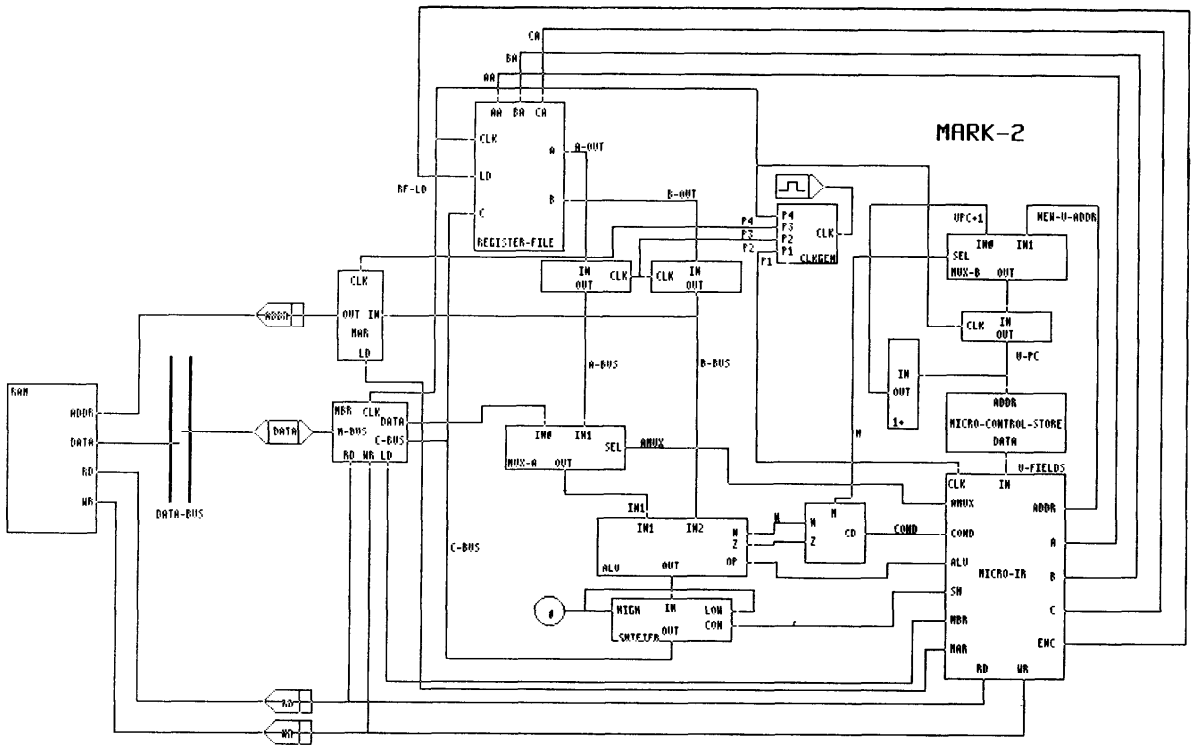

Figure 5: The MARK-2 Processor

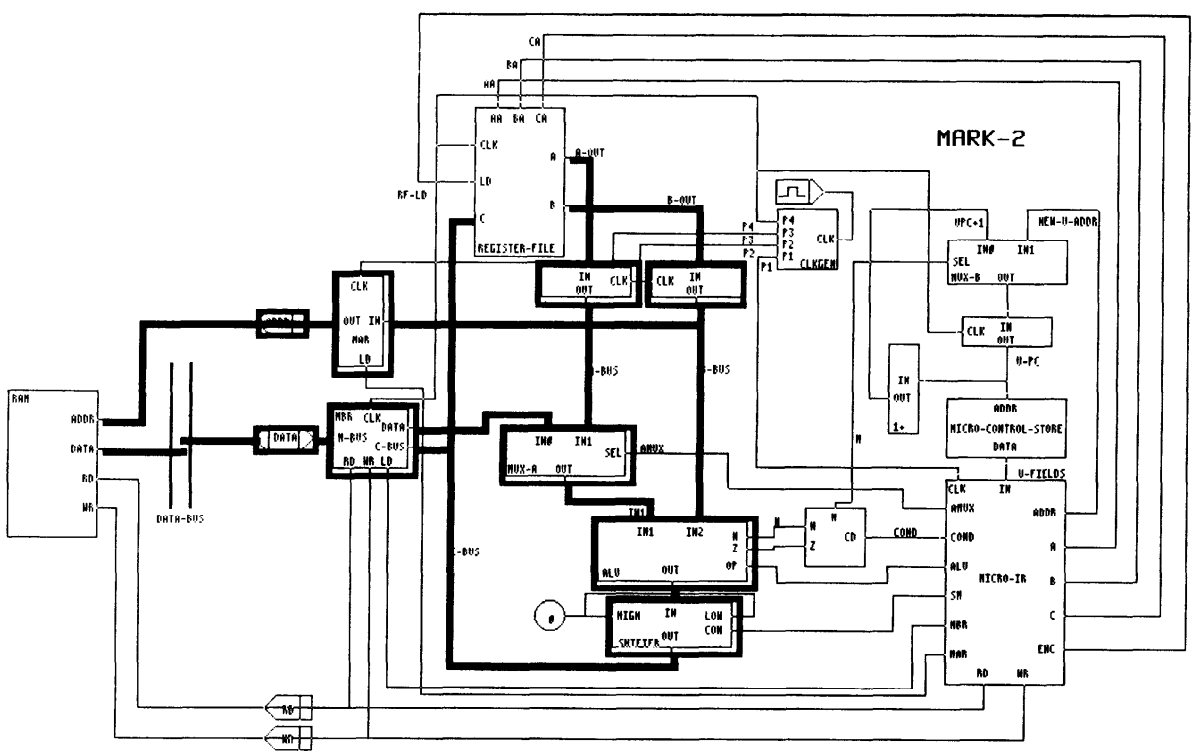

Figure 6: Coverage of processor components

This account also suggests a difference between our approach to test generation and the classical approaches. The usual result of the design process, a circuit topology, is the superposition of the dataflow graphs, plus the identity boxes added to allow sharing, plus the controlling state machine. This topology doesn't include the list of operations the circuit can perform, nor does it make explicit how data moves through the circuit, e.g. timing and selection relationships, both of which were explicit in the flow graphs before they were merged.

Classical test generation algorithms perform poorly, because they attempt to generate tests directly from the circuit components and topology. Testing experts perform well, because they know the same things designers know and can reconstruct and use the designer's flow graphs. Finally, our program is successful for the same reason, it reconstructs and uses behavior graphs, which are an approximation of the flow graphs. [8]

8 Conclusion

We have demonstrated how knowledge of a system's designed behavior can help solve test generation problems, or more generally, problems involving manipulation of complex engineered systems. The key to our method is that a device's designed behavior can be far more limited than its potential behavior. This limitation can translate into a reduction of the search necessary to achieve planning goals.

A prototype system has been implemented and run on a small number of examples. This system has 3 components: (1) a preprocessor to create the behavior graphs using symbolic simulation of input sequences out of the databook; (2) a micro-planner to match testing goals gotten from domain experts against the simulation runs (the matcher is guided by the Component Activation Summaries); and (3) a macro-planner to control global aspects of the circuit's state (the behavior graphs also supply the planner's operator descriptions).

The system has generated tests for a simple processor's datapath. It was successful despite the fact that the datapath is controlled by 80 lines of intricate microcode. A clear advantage of our method is that it can, by using simulation, reason forward through complex components, such as the controller, in order to solve problems involving the components on the other side.

9 Acknowledgments

The following people have read drafts or otherwise contributed to the contents of this paper: Randall Davis, Gordon Robinson, Yehudah Freundlich, Jeff Van Baalen, Walter Hamscher, Glenn Kramer, Howard Shrobe, Raúl Valdés-Pérez, Brian Williams, Peng Wu. I would especially like to thank Gordon Robinson, who is the testing expert whose methods inspired much of this work.

[8] The domain experts report an interesting dichotomy. They say that, for humans, test generation problems are either very easy or very hard. The problems are easy when they can see clearly which manipulations of the device's interfaces are useful for exercising components inside the circuit, and hard otherwise. In our work, we have have made some effort, albeit informally, to match the performance of our test generation system to the expert's intuitions about which problems are easy for humans and which are hard. And the expert does report that the results on the MARK-2 match his expectations fairly closely.

References

[DeKleer79] de Kleer, J., *Causal and Teleological Reasoning in Circuit Recognition,* Diss., Massachusetts Institute of Technology, 1979, AI-TR-529.

[Roth66] Roth, J. P., "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development,* Vol. 10, July 1966, pp. 278-291.

[Shirley85] Shirley, M., "An Automatic Programming Approach to Testing," *IEEE Workshop on Simulation and Test Generation Environments,* Sept 1985.

[Singh85] Singh, N., *Exploiting Design Morphology to Manage Complexity,* Diss., Stanford Department of Electrical Engineering, April 1985.

[Tanenbaum84] Tanenbaum, A. S., *Structured Computer Organization,* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.