Saturn: An Automatic Test Generation System for Digital Circuits

Narinder Singh

Department of Computer Science
Stanford University
Stanford, CA 94305

Schlumberger Palo Alto Research
3340 Hillview Ave.
Palo Alto, CA 94304

ABSTRACT

This paper describes a novel test generation system, called Saturn, for testing digital circuits. The system differs from existing test generation systems in that it allows a designer to specify the structure and behavior of a design at a collection of abstraction levels that mirror the design refinement process. The system exploits the abstract design formulations to increase the efficiency of test generation by ignoring irrelevant detail whenever possible. These capabilities are made possible by using general representation and reasoning methods based on logic, which provide a declarative representation of a design, and permit using a single inference procedure for reasoning both forwards and backwards through the design for test generation.

I Introduction

With the advances being made in the technology of manufacturing digital devices it is now possible to build systems of unprecedented complexity. An integral part of manufacturing such systems involves testing them in order to ensure their correct operation. The complexity of these systems has led directly to the complexity of generating a set of test vectors to verify the correct operation of the device. Traditional approaches to testing devices have either failed to distinguish between the most detailed device structure and its design, or have provided a limited vocabulary for capturing higher level design descriptions. The inability to represent and reason with abstract design formulations will make it impractical to generate tests for complex devices using traditional methods, where the cost is exponential in the size of the design.

It is possible to manage the complexity of generating tests for complex devices by capturing their description (their subparts and the relationships between them) at a collection of abstraction levels. By capturing higher-level design formulations we can dramatically improve the efficiency and quality of solutions for test generation, and also reduce the size of a design. It is possible to capture such high level design descriptions by recording the evolving descriptions of a design during the refinement process. Though not ideal for test generation, these descriptions are much more efficient to reason with compared to a flat low-level (e.g., gate level) description of a device.

This paper describes Saturn, a novel test generation system that permits a designer to specify the structure and behavior of a design at a collection of abstraction levels. The system exploits abstract design formulations to increase the efficiency of test generation by reducing the depth and branching factor of nodes in the search space. This paper is outlined as follows: section 2 defines the test generation task, which is followed by the description of the Saturn test generation system. Section 4 will examine the important dimensions along which a design can be abstracted to capture its higher level formulations, and section 5 will present some experimental results that demonstrate the utility of reasoning with abstract design formulations. Finally, the last section will conclude with a summary and a description of our current research efforts.

II Test Generation

Test generation involves generating a collection of tests, which check the functionality of a device. The design of the device is assumed to be correct. However, the manufacturing process which realizes the physical device from the design specification is assumed to be imperfect. In this paper we are focusing on testing the functionality of a device, and are not addressing testing other properties, e.g., power consumption, and the steady state voltage and current parameters. In addition, we are only interested in checking if a device is functioning or not, rather than identifying the source of any failures.

The goal of test generation is to come up with a sequence of tests, such that if the device satisfies these tests it is guaranteed to be consistent with its design. This goal must be satisfied subject to certain constraints (e.g., minimizing the length of the test vectors). In practice, it is impractical to generate the minimal set of test vectors to test a device, so a small set is acceptable. The result of the test generation process is a collection of tests, where each test specifies the values for some inputs and the expected values at some outputs.

If the possible manufacturing failures for a device are either stuck-at 1 or stuck-at 0 faults at the boolean gate level, the test generation process must generate tests to check each possible fault for all the boolean gates in the device. For example, Figure 1 shows a 2-bit adder device, which is made up of 2 full-adders, which themselves are implemented using a collection of boolean gates. Testing this device includes testing the inputs and output of the the exclusive-or gate *xor*1 to see if they are stuck-at 1, or stuck-at 0. A test for checking if the output is stuck-at 0 requires controlling the inputs so that the output should be 1 if the device is working, and checking the actual output for these inputs. For example, if the inputs of the exclusive-or gate are 1 and 0 and the output is 1, then the output cannot be stuck-at 0.

Testing a device is complicated by the fact that usually a small fraction of its ports are directly controllable or directly observable. In order to test a subpart of a device we must set the directly controllable inputs so that the part being tested has the desired inputs and its output is propagated to a directly observable port. For the previous test for *xor*1 this involves controlling the inputs of the adder to 1 and 0, and observing a 1 at the output if none of the internal nodes can be directly controlled/observed. The exact value propagated to a directly observable output is not important as long as it depends on the output of the part being tested.

The key problem in testing is to generate a reasonably small set of vectors that test all possible failures of a device. The execution of a single test is relatively inexpensive, and the number of vectors required to test a device is linear in the number of

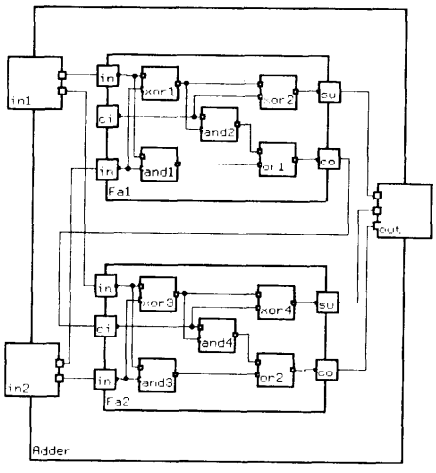

Figure 1: A design for an adder with 2-bit inputs.

parts [6]. A naive test generation algorithm would enumerate all input/state combinations; however, the number of test vectors generated will be unacceptably large. The key problem in reducing the number of test vectors is the cost of test generation, which is exponential in the depth of the circuit [9]. This complexity is compounded for sequential circuits where we may have to "unfold" a copy of a finite state machine for each possible state.

The full-adder example was used to illustrate the test generation task for a simple device. For large real-world devices this task is significantly more complex. In practice, the most optimistic empirical estimates show that the cost of test generation is proportional to the cube of the circuit size [6].

III Saturn

The inputs to the Saturn test generation system are: the design description for the device, the directly controllable inputs and outputs of the device, user specified tests for parts of the device, and the user specifications of the abstraction levels at which the parts of a design are to be tested. These are specified in the language Corona [20], which defines a vocabulary of *terms* in a prefix form of predicate calculus. The system uses the resolution residue planning procedure [3] implemented in the knowledge representation system MRS [19] to automatically generate tests from the design descriptions. The system exploits the meta-level control capabilities of MRS to increase the efficiency of test generation by selecting the most abstract formulation for propagating values through a design. The performance of the system is further enhanced by checking the consistency of evolving solutions, and sharing the results for one test with others by caching tests and solutions to subtasks for these tests.

Saturn is an algorithmic test generation system similar to the D-algorithm [18]. Both systems achieve tests by propagating values forwards and backwards through a design. However, the D-algorithm can only generate tests for stuck-at faults at the boolean gate level, which is extremely inefficient for large designs. Saturn, on the other hand, can generate tests for designs described at arbitrary abstraction levels, with user specified fault models. Hitest [17] and SCIRTSS [8] are examples of more recent systems that attempt to exploit higher level design formulations for generating tests, and allow the user to specfy heuristics to control the reasoning process. The SCIRTSS system only permits specification of designs at two specific abstraction levels, as opposed to an arbitrary collection of abstraction levels. The Hitest system uses the PODEM [7] algorithm for achieving tests, where the system randomly generates inputs for the design and propagates these forwards to see if these achieve a test. Unfortunately, the random generation of inputs does not work well for testing sequential circuits which require a specific sequence of stimuli to propagate values from their inputs to an output.

The remainder of this section will illustrate the operation of the Saturn test generation system for the 2-bit adder device pictured in Figure 1.

A. Describing Designs

The design specification for the adder includes the behavior for: the adder as a whole, the full-adders, the ten boolean gates, and the connections. The following collection of facts specify the structure of the design by specifying the components and their type.

```
Andg(a1)          Port(a1-in1 a1)···Conn(a1-out o1-in2)···
Org(o1)           Port(o1-in1 o1)···Conn(o1-out a3-in1)···
Xorg(x1)          Port(x1-in1 x1)···Conn(x1-in1 a1-in1)···
Full-adder(f1)    Port(f-in1 f1)  ···Conn(f-in1 x1-in1)  ···
Adder(a)          Port(a-in1 a)   ···Conn(a-in1-s f1-in1)···
```

The facts on the first line assert that a1 is an and gate, a1-in1 is a port of the module a1, and that the port a1-out is connected to the port o1-in2. The remaining lines similarly define the type of the other components in the design and their interconnection.

The behavior of the modules and the connections is specified by a collection of facts in conjunctive-normal-form (CNF)[1]. The lower case letters in the following behavior descriptions stand for universally quantified variables, and the "@" character is used to define the time at which a fact is true. For example, the fact a1-in1@3=0 asserts that the value of the port a1-in1 at time 3 is 0. In this design, connections have 0 delay, and the gates have a delay of 5 time units.

```
¬Conn(y z) ∨ ¬y@t=u ∨ z@t=u
¬a1-in1@t=0 ∨ a1-out@t+5=0
¬a1-in2@t=0 ∨ a1-out@t+5=0
¬a1-in1@t=1 ∨ ¬a1-in2@t=1 ∨ a1-out@t+5=1
...
¬f1-in1@t=u ∨ ¬f1-in2@t=v ∨ ¬f1-cin@t=w ∨
¬Majority(u v w)=x ∨ f1-cout@t+15=x
Majority(x y y)=y
Majority(y x y)=y
Majority(y y x)=y
...
¬a-in1@t=u ∨ ¬a-in2@t=v ∨ a-out@t+30=u+v
```

The fact on the first line defines the zero delay behavior for connections. It states that if y is connected to z, and if y at time t has some value u, then z has the same value at the same time. The next three facts define the behavior of the and gate a1. The first two state that the output is 0 if any input is 0, and the third states that the output is 1 when both inputs are 1. The next fact defines the behavior of the carry output of the first full-adder in terms of the majority function Majority. The next three lines define this function, where the result is equal to the two arguments that are identical. The fact on the last line defines the behavior of the adder as a whole. In specifying these behaviors, we have enumerated the input combinations (e.g., the and gate a1), made use of built in functions (e.g., +), and defined new functions (e.g., Majority).

[1] The rule $A \wedge B \rightarrow C$ is equivalent to the clause $\neg A \vee \neg B \vee C$.

B. Automated Deduction

The Saturn test generation system uses the resolution residue planning procedure [3] for propagating values through a design. This procedure takes a collection of facts in CNF, a set of assumable facts (primitive actions that can be directly executed), and a specification of a goal. The inference procedure returns a set of assumable facts which together with the original design description entail the goal. An example of the basic rule of inference for resolution is given below:

$$\frac{\alpha \vee \beta \qquad \neg\alpha \vee \gamma}{\beta \vee \gamma}$$

This rule states that if you know that the clauses on the top two lines are true, then you can conclude that the clause on the bottom line is also true. The resolution procedure matches a literal in one clause against the negation of that literal in the second clause. In this example the literal α in the first clause is matched with its negation $\neg\alpha$ in the second clause. If such a match can be found, then we can conclude that the clause consisting of the disjunction of the remaining literals from the matching clauses must also be true. In this example the remaining literals from the first clause are $\{\beta\}$, and those from the second clause are $\{\gamma\}$. Therefore, the clause $\beta \vee \gamma$ must also be true.

The resolution residue planning procedure adds the negated goal to the original design, and repeatedly applies the resolution rule of inference until a clause with all assumable literals is derived. This procedure can be used to control and observe values in propagating tests through a design. For the adder example, suppose we are trying to control the output of the and gate a1 to 1 at time 32. The negation of the goal, ¬a1-out@32=1, is resolved with the last clause of the behavior of the and gate (defined earlier) to give ¬a1-in1@27=1 ∨ ¬a1-in2@27=1 by matching the variable t to 27. In other words, in order to control the output of the gate to 1 at time 32 both inputs must be controlled to 1 at time 27. This inference procedure is repeated untill the port to be controlled is one of the directly controllable inputs.

The same inference procedure can also be used to observe port values. For example, suppose we want to observe the value 1 at the first input of the same and gate at time 3. The negation of the goal[2], a1-in1@3=1, is resolved with the last clause of the behavior of the and gate to give ¬a1-in2@3=1 ∨ a1-out@8=1 by binding the variable t to 3. That is, in order to observe the value 1 at the first input of the and gate at time 3 we must control the other input to 1 at the same time and observe the value 1 at its output at time 8. This inference procedure is repeated untill the port to be observed is one of the directly observable outputs.

C. Algorithm

The system first examines the topology of the design and computes the estimates of the number of inference steps required for controlling and observing every port (separate estimates for controlling/observing) based on the directly controllable inputs and the directly observable outputs. The cost of controlling the directly controllable inputs is 1, and the the cost of controlling an end port of a connection is one more than the cost of controlling the starting port. The cost of controlling an output of a module is one more than the sum of the costs of controlling all its inputs. Similarly, the cost of observing a directly observable output is 1, and the cost of observing the starting port of a connection is one more than the minimum of the costs of observing all the end ports (e.g., the minimum at a fanout point). Finally, the cost of observing an input of a module is one more than the sum of the costs of controlling all the other inputs added to the minimum cost of observing an output. This scheme can be generalized in a straightforward manner to handle devices with feedback [22].

The actual test generation phase starts after the initial cost estimates are calculated. The system starts with the most abstract design description and only examines the substructure when it is absolutely necessary to do so. Assume that the user has specified that all composite components in the adder must be tested by testing their subparts, and that the possible faults are for the ports of boolean gates to be stuck-at 1 or 0. This requires testing the adder by generating tests for the two full-adders, which themselves must be tested by testing their gates.

Generating tests for the gates of the full-adders requires propagating the values at their ports to the boundary of the full-adder. We can propagate these values through the design using the resolution residue inference procedure described earlier. For example, one test for checking that the output of and2 is 1 when both inputs are 1 requires: controlling the inputs of the full-adder to 1, 0 and 1, and observing the value 1 at the carry output. To complete the test generation process for the full-adder we must achieve the remaining tests for and2, and all the tests for the remaining gates. This process generates 17 tests to test all stuck-at faults for the gates of the full-adder.

An important step in the test generation process is to minimize the number of tests by combining tests with compatible inputs. Finding a minimal set of test vectors is equivalent to minimizing a collection of expressions, which is known to be NP-hard for boolean expressions [4]. Instead of minimizing a collection of test vectors, the system employes a more computationally tractable, though non-minimal, compression algorithm (the cost is proportional to the square of the number of test vectors). The algorithm compares the inputs for each test with the inputs of the other tests using the unification pattern matching process [16]. If the inputs of a test do not match any other it is left unchanged. Otherwise, the test is deleted and the variables in the other matching test are bound to the values that make the inputs of the two tests equivalent. The minimization process reduces the 17 original tests for the full-adder to 6.

After minimization, the tests must be *abstracted* to the next higher level in the hierarchy. In this case each port of the full-adder f1 has exactly one subport, and there is no transformation of the values across the port subport boundaries. The abstraction of these tests to the level of the full-adder results in the same values, but for the parent ports. This is not always the case. For example, for the adder the top-level behavior is defined in terms of integer values, and the behavior of its subparts is defined in terms of boolean values.

At this point we have derived the tests for the full-adder f1 from the definitions of the tests for its parts. It is possible to share this definition with all identical, or similar components, e.g., the other full-adder f2. In order to share this definition the test vectors are *generalized*. The generalization, at present, is limited to apply to a class of components that only differ from the original in their delay. Other generalizations that we have not implemented include generalizing tests for components with similar behavior, e.g., parameterizing tests for an adder based on the number of bits at each input.

In order to complete generating tests for the adder the above process must be repeated for the two full-adders. In propagating the tests for the full-adders through the design, the system will never examine the substructure of either full-adder. It is possible to use the behavior descriptions of the full-adders directly to propagate values from their inputs to the outputs and vice-versa.

D. Control of Reasoning

In this subsection we will examine the following control strate-

[2]Goals for observing port values are negated, e.g., the goal for this example is ¬a1-in1@3=1.

gies that the Saturn test generation system employs in order to increase the efficiency of inference for test generation: consistency checking, heuristics to guide search, and caching.

1. Consistency Checking

In general, there will be more than one choice at each decision point in the search space. The goal of consistency checking is to prune inconsistent paths in the search space as early as possible. For example, it is impossible to control the output of the or gate in a full-adder to 1 by controlling both its inputs to 1. In choosing a subgoal sg_i at a node in the search space, consistency checking corresponds to seeing if it is possible to prove $\neg sg_i$, in which case this node is pruned. The utility of consistency checking is dependent upon selecting the appropriate amount effort in attempting to prove $\neg sg_i$. The drawback of too little effort is that inconsistencies are not caught early, and too much effort has the potential drawback that fruitless work may be done for consistent nodes.

The Saturn system performs consistency checking incrementally by propagating the consequences of every choice through the design. These consequences can propagate information forward and backward through the design. In general, it is computationaly inefficient to *prove* that a given choice is consistent, since this task is non-semi-decidable. The Saturn system performs limited consistency checking by only propagating the values of individual ports, and state variables. This is sufficient for detecting contradictions for atomic clauses, but not for disjunctive clauses. For example, it cannot detect that the four clauses $(a \vee b)$, $(\neg a \vee \neg b)$, $(\neg a \vee b)$, and $(a \vee \neg b)$ are mutually inconsistent.

If an alternative at a choice point is inconsistent with the current state, it is pruned from the search space, and the system backtracks to try an alternate path. Since inconsistencies are not always detected immediately, chronological backtracking can be inefficient. Saturn keeps track of the justifications for each fact, and follows the justifications back to a choice point, and tries another alternative at the source of the inconsistency. This corresponds to dependency-directed backtracking [2], which dramatically improves the efficiency of test generation.

2. Heuristics to Guide Search

Consistency checking cannot eliminate search, since there may be more than one consistent alternative at a choice point. The Saturn system uses heuristics to select the most promising alternative at a choice point. These heuristics are based on the estimated deductive cost for controlling/observing a port value, and the probability of being able to achieve this goal given the current problem constraints.

The cost of a task is equal to the sum of the costs of its subtasks, which can be looked up directly in the knowledge base after the cost estimates have been calculated. The probability of being able to achieve a task is equal to $\prod_{i=1}^{commonfanouts} (D_i)^{-(N_i-1)}$, where *common fanouts* are the fanout points in the design that need to be controlled for the task, D_i is the size of the domain of possible values at fanout point i, and N_i is the number of subgoals of the task that need to control the fanout point i. As a preprocessing step the system records the number D_i with every fanout node, and also records the fanout nodes that may need to be controlled for controlling/observing every internal port. We can calculate N_i for a task by looking up the fanout nodes that need to be controlled for its subtasks.

Let $C(T_i)$ be the cost of executing task T_i, let $P(T_i)$ be the probability of succeeding in achieving task T_i, and $C(T_i, T_j)$ be the cost of executing task T_i followed by executing task T_j. Since the tasks are independent of each other:

$$C(T_i, T_j) = C(T_i) + (1 - P(T_i)) \times C(T_j)$$
$$C(T_i, T_j) < C(T_j, T_i) \iff \frac{P(T_i)}{C(T_i)} > \frac{P(T_j)}{C(T_j)}$$

That is, the task with the largest probability of success-to-cost ratio should be executed first.

The cost heuristics automatically select the more abstract design descriptions for propagating values since these have a lower cost associated with them. For example, in order to control the sum output of a full-adder it is cheaper to use the behavior of the full-adder as a whole since this only requires controlling the inputs of the full-adder. The cost of controlling the output of the exclusive-or gate driving the sum output must be greater since it includes the cost of propagating the values at the input of the full-adder through the other gates and connections.

Generating tests for an adder with four bit inputs required 272 seconds on a Symbolics 3600 using the cost estimates to select tasks. When the tasks were selected based on their order of generation, the time required was more than 2 hours, a factor of 26 slower.

3. Caching

In the Saturn test generation system, caching is used to share the definition of how to test one component with other similar components, and to share the solutions to subtasks for one test with other tests.

Without test caching, the number of tests to generate is proportional to the number of modules in the design, as opposed to the number of different module types. For a hierarchically formulated design with l levels and approximately n modules per level (five for the full-adder), the number of modules to generate tests for is approximately n^l. If the average number of different types of submodules of a module is m (three for the full-adder), the number of modules to generate tests for is m^l. Since $n > m$, these savings can be quite substantial.

Generating tests for an adder with four bit inputs (with subparts that are four full-adders) required 272 seconds with test caching, and 1230 seconds without it. This represents a factor of 4.5 improvement.

Test generation involves testing all the possible faults for a device. These tasks share many subtasks in common. For example, it is possible to control the output of an and gate to 0 by controlling its first input to 0. This backward propagation is followed to the boundary of the enclosing hierarchy to find a solution. This same solution (the values of the inputs at the enclosing hierarchy boundary) can be cached with the other subgoals back to the hierarchy boundary (e.g., the first input of the and gate can be controlled to 0 with these inputs).

For the 4-bit adder, the cost of test generation is reduced from 591 seconds to 272 seconds by caching solutions to subgoals. This represents a factor of 2.2 improvement. These examples have illustrated the savings for designs with one level of hierarchy. These savings are multiplied for complex designs with additional levels of hierarchy.

IV Exploiting Abstract Descriptions

One of the key features of the Saturn test generation system is that it permits the user to extend the vocabulary for describing designs. This allows a designer to specify a design in terms of the objects, functions and relations that he thinks of. We can exploit the higher level formulations of a design for test generation by capturing them in the design refinement process. By capturing design descriptions in predicate calculus it is possible to use the same descriptions to reason forwards and backwards through a design, as is needed for an automatic test generation system.

Reasoning with higher level design formulations increases the efficiency of test generation by reducing the complexity of the descriptions. In this section we will examine following dimensions along which a design can be abstracted to capture these formulations: structural abstraction, functional abstraction, and value abstraction. By abstracting a design we are augmenting the original description with additional facts. We do not discard the lower level descriptions.

A. Structural Abstraction

Structural abstractions corresponds to the case where a design is augmented by a new module whose subparts are a collection of existing modules. For example, creating a new full-adder module whose subparts are the appropriate interconnection of the five boolean gates. We must define the sum and carry outputs of the full-adder as a function of its three inputs. The behavior of the newly created module must be related to its subparts by defining the relations between the values at its ports and the values at the ports of its subparts (e.g., via connections).

By allowing the user to define structural abstractions we can replace a subtree in the search space with a single node and its children. For example, we can define the input/output relations for the full-adder as a whole as a table (e.g., the Majority function), and directly look up the solutions for controlling an output. Using the lower-level design formulation we must reason backwards through the gates of the full-adder. The search space at the lower level includes: paths with no solutions, different paths with identical solutions, and solutions with redundant conjuncts [22]. Reasoning with structurally abstracted design formulations for test generation provides a savings which is exponential in the difference between the depth of the original and reformulated description. In addition, by sharing the descriptions of identical components using prototypes the size of a structurally abstracted design can be made proportional to the number of distinct module types, as opposed to the total number of modules.

B. Functional Abstraction

Functional abstraction corresponds to the case where the behavior of an existing module is specified in terms of a newly defined function. For example, we can define the behavior of the 2 bit adder by writing clauses that enumerate all the input/output combinations (16 clauses in all). Alternatively, we can define the addition function + (including its properties, e.g., commutativity, associativity), and define the behavior of the adder in terms of this function, as is done in the following CNF clause: $\neg$in1=x $\vee$ $\neg$in2=y $\vee$ out=x+y.

Functionally abstracting designs permits achieving tests using constraint propagation. For example, if a subgoal of a test requires controlling the output of an adder to 4122, we can achieve this goal by propagating symbolic constraints through the design. That is, the goal out=4122 is replaced by the new subgoals in1=x $\wedge$ in2=y $\wedge$ x+y=4122. The original design formulation forces the inference mechanism to employ search to solve this problem. That is, the system must prematurely select input combinations that add up to 4122, e.g., 0 and 4122, 1 and 4121, etc. The size of the search space using this strategy is exponential in the depth of the circuit. For a system of linear constraints, functional abstraction reduces the cost to a polynomial function of the number of constraints. However, if the constraints are nonlinear it may be advantageous to use search if there are good heuristics to guide the inference mechanism to paths likely to have solutions.

C. Value Abstraction

Value abstraction corresponds to the case where the behavior of an existing module is defined in terms of more abstract values.

Device	# Tests Flat High Level	# Tests Refined Gate Level	Flat Time (sec.)	Refined Time (sec.)
full-adder	16	9	.84	33
$adder_2$	16	9	.6	82
$adder_3$	64	12	1.8	121
$adder_4$	256	13	6.6	190
$adder_8$	65536[a]	13	>1800 [a]	772

[a](ran out of swap space on a 3670)

Table 1: Utility of refining designs.

The original values are partitioned into equivalence classes such that all values in the same equivalence class map to a unique value at the abstract level. For example, at the lower-level we can model the behavior of a multiplier in terms of integer values at its inputs and outputs. At a more abstract level we can define the behavior of the multiplier in terms of the objects positive and negative, i.e., $\neg$in1=positive $\vee$ $\neg$in2=negative $\vee$ out=negative, etc..

Controlling the output of the multiplier to a negative number using the original design formulation requires prematurely selecting specific integers at the inputs with opposite sign. Using the value abstracted design description we find the two solutions: in1=positive $\wedge$ in2=negative, and in1=negative$\wedge$ in2=positive. This delays the actual selection of specific values at the inputs, some of which may be inconsistent with the current state.

Value abstraction reduces the branching factor of the nodes in the search space by reducing the number of alternatives that achieve a goal. A linear reduction in the branching factor reduces the size of the search space by a factor that is exponential in the depth of the circuit.

V Experimental Results

In this section we will present examples that illustrate the utility of abstracting and refining designs, and describe the complexity of devices for which Saturn has been used. In these examples it is assumed that the possible failures are stuck-at faults for the boolean gates.

Table 1 shows the utility of refining designs for a collection of devices of increasing complexity. For example, $adder_i$ stands for an adder with i bit inputs. The columns for the flat design formulation correspond to a black box description of the device at a high level, which must be tested by testing all input combinations. The columns for the refined design formulations correspond to descriptions that have been refined to the gate level[3]. The number of tests and time grows exponentially using the high-level formulation alone, whereas the number of tests remain roughly constant using the refined descriptions and the time grows quadratically. For small devices the cost of test generation is cheaper using only the high level descriptions. However, for large devices both the time and quality of solutions (smaller number of tests) improves using the refined descriptions.

An example illustrating the utility of abstracting designs is given in Table 2 for an adder with 4 bit inputs. The flat gate level formulation corresponds to a description of the adder in terms of boolean gates alone, while the hierarchical description defines the adder in terms of 4 full-adders, which are themselves defined in terms of boolean gates. In addition to reducing the time required to generate tests by a factor of 3.5, the abstracted description provided better tests (a smaller set). This is due to the imperfections of the test minimization algorithm. The repeated application of the minimization algorithm at the

[3]Each adder has been refined into a collection of full-adders, which themselves are refined into a collection of boolean gates.

Formulation	Time	# Tests	Cost Factor
Hierarchical	272 sec.	13	1
Flat Gate Level	962 sec.	16	3.5

Table 2: Advantage of abstracting a design description.

boundaries of the full-adders with a smaller set of tests provides better solutions compared to a single application with a large set of test vectors.

The largest design for which we have generated tests consists of 3 multipliers and 2 adders (with 4 bit inputs). This design has approximately 650 objects, seven levels of hierarchy and values ranging from bits to integers. The Printer Adapter card of the IBM PC is the most realistic example for which we have generated tests. The interesting aspects of this board are that it has feedback paths, bi-directional signals, and tri-state busses. The system generated 40 tests using a high level formulation of the board in approximately 30 minutes[4]. Currently we are in the process of generating tests for the motherboard of the IBM PC-AT. This device is significantly more complex since it includes complicated microprocessors (Intel 80286,87), and peripheral chips.

VI Conclusion

In order to generate tests efficiently for complex devices we must reason with higher level design formulations. Existing test generation systems have restricted the vocabulary for describing designs, thus preventing a designer from specifying the higher level design formulations that are created in the design refinement process.

By capturing higher level formulations of a device we can reduce the size of the search space exponentially by reducing its depth and branching factor. We have empirically validated this by demonstrating that both the time and the quality of solutions can be improved by reasoning with higher level design formulations. Thus, by capturing a design formulation at a collection of abstraction levels we can increase the complexity of the devices that we can generate tests for.

We believe that this approach is suitable for large real-world designs. At present we have demonstrated this for the printer adapter card of the IBM PC, and we are in the process of modeling and generating tests for the motherboard of the IBM PC-AT.

Acknowledgments

This paper has benefited from comments and suggestions by Mike Genesereth, Glenn Kramer, Mark Shirley, and Vineet Singh. This research was funded in part by Schlumberger Palo Alto Research.

REFERENCES

[1] Comerford, R. and Lyman, J. "Self-Testing Special Report," *Electronics*, March 10, 1983, pp 109-124.

[2] Doyle, J. "Truth Maintenance Systems for Problem Solving," AI-TR 419, Massachusetts Institute of Technology, January, 1978.

[3] Finger, J. and Genesereth, M. "Residue: A Deductive Approach to Design Synthesis," HPP-85-1, Stanford University Heuristic Programming Project, January, 1985.

[4] Garey, M. and Johnson, D. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979, pp 161-164.

[4] Additional information for these examples can be found in [22].

[5] Genesereth, M. et. al. "The MRS Dictionary," HPP-80-24, Stanford University Heuristic programming Project, January, 1984.

[6] Goel, P. "Test Generation Cost Analysis and Projections," *Proceedings of the 17th Design Automation Conference*, June, 1980.

[7] Goel, P. "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers*, vol. c-30, no. 3, 1981, pp 215-222.

[8] Hill, F. and Huey, B. "SCIRTSS: A Search System for Sequential Circuit Test Sequences," *IEEE Transactions on Computers*, May 1977, pp 490-502.

[9] Ibarra, H. and Sahni, S. "Polynomially Complete Fault Detection Problems," IEEE Transaction on Computers, vol. C-24, no. 3, March 1976, pp 242-250.

[10] Kramer, G. "Employing Massive Parallelism in Digital ATPG Algorithms," *Proceedings of the 1983 IEEE International Test Conference*, IEEE Press, pp 108-114.

[11] Lai, K. "Functional Testing of Digital Systems," CMU-CS-81-148, Carnegie-Mellon University, December, 1981.

[12] Mark, G. "Parallel Testing of Non-volatile Memories," *Proceedings of the 1983 IEEE International Test Conference*, IEEE Press, pp 738-743.

[13] McCluskey, E. "A Survey of Design for Testability Scan Techniques," *VLSI Design*, December, 1984, pp 38-61.

[14] McCluskey, E. "Minimization of Boolean Functions," *Bell System Technical Journal*, 35, no. 6, November, 1956, pp 1417-1444.

[15] Moszkowski, B. "Reasoning about Digital Circuits," STAN-CS-83-970, Stanford University, June, 1983.

[16] Nilsson, N. *Principles of Artificial Intelligence*, Tioga Publishing Company, Palo Alto, 1980.

[17] Robinson, G. "Hitest— Intelligent Test Generation," *Proceedings of the 1983 IEEE International Test Conference*, IEEE Press, pp 311-323.

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

[19] Russell, S. "The Complete Guide to MRS," KSL-85-12, Stanford Knowledge Systems Laboratory, June, 1985.

[20] Singh, N. "Corona: A Language for Describing Designs," HPP-84-37, Stanford University Heuristic Programming Project, September, 1984.

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

[22] Singh, N. *Exploiting Design Morphology to Manage Complexity*. PhD thesis, Stanford University, August 1985.