Functional Vector Generation for HDL models Using Linear Programming and 3-Satisfiability

Farzan Fallah, Srinivas Devadas
Department of EECS
MIT, Cambridge

Kurt Keutzer
Department of EECS
University of California, Berkeley

Abstract—

Our strategy for automatic generation of functional vectors is based on exercising selected paths in the given hardware description language (HDL) model. The HDL model describes interconnections of arithmetic, logic and memory modules. Given a path in the HDL model, the search for input stimuli that exercise the path can be converted into a standard satisfiability checking problem by expanding the arithmetic modules into logic-gates. However, this approach is not very efficient.

We present a new HDL-satisfiability checking algorithm that works directly on the HDL model. The primary feature of our algorithm is a seamless integration of linear-programming techniques for feasibility checking of arithmetic equations that govern the behavior of datapath modules, and 3-SAT checking for logic equations that govern the behavior of control modules. This feature is critically important to efficiency, since it avoids module expansion and allows us to work with logic and arithmetic equations whose cardinality tracks the size of the HDL model.

We describe the details of the HDL-satisfiability checking algorithm in this paper. Experimental results which show significant speedups over state-of-the-art gate-level satisfiability checking methods are included.

I. Introduction

The most common approach to checking correctness of a design is to verify that a description of the design in some hardware description language has the proper behavior as elicited by a series of simulation vectors. There are several drawbacks of this approach. Exhaustive simulation is required to guarantee correctness and is possible only for small circuits. If a manageable number of input stimuli are to be simulated, the problem of selecting or generating these input stimuli arises. Generating random or pseudo-random input stimuli is relatively easy; however, random input stimuli, typically, do not thoroughly exercise the design and therefore miss a lot of design errors.

One can base a functional vector generation strategy on coverage metrics. Commonly used coverage metrics correspond to instruction counts, i.e., how often, if at all, an instruction or statement in the hardware description language (HDL) code is exercised. Full statement coverage corresponds to the input stimuli exercising every statement in the given HDL code at least once. Full path coverage is a more comprehensive criterion, where every path in the HDL code is exercised by the input stimuli. Since the number of paths in the HDL model can grow exponentially with the size (number of lines) of the model, full path coverage is generally not attainable. Design-specific strategies for computing coverage have been proposed. Very often, designers manually generate functional vectors with full statement coverage, and which exercise all "interesting" paths in the model. The generation of input stimuli that satisfy coverage requirements is a tedious process. However, the automatic generation of functional vectors is a difficult computational problem.

We assume that we are given an HDL model as input which describes interconnections of arithmetic, logic and memory modules. We have chosen selective path coverage as a metric for functional vector generation. Given a set of paths in the HDL model, for each path, we solve a satisfiability problem to obtain the input stimuli that exercise the chosen path. Path sensitization in HDL models can be converted into a standard satisfiability checking problem by expanding the arithmetic modules into logic-gates. However, this approach is not very efficient.

We present a satisfiability algorithm that works directly on the HDL model. The primary feature of our HDL-satisfiability algorithm is a seamless integration of linear-programming techniques for arithmetic modules and 3-satisfiability (3-SAT) checking for logic modules. This integration is critically important to efficiency, since it avoids module expansion and allows us to work with logic equations and arithmetic equations that track the size of the HDL model. We use 3-SAT checking to solve the logic equations and linear-programming to check the feasibility of the arithmetic equations. Interactions between Boolean variables in the 3-SAT equations and integer variables in arithmetic equations are handled by increasing the number of integer variables while reducing their range.

We have chosen path coverage as our metric because exercising computation paths in a circuit is a tried-and-trusted way of design debugging. Since the number of paths in the circuit may be too large to enumerate, we select a subset of paths. Other coverage metrics, such as statement coverage, can also be used in our framework. This will only impact the specifics of the generated satisfiability problem.

Representative approaches for gate-level satisfiability include [8] and [13]. An excellent survey of satisfiability checking methods at the gate level that includes empirical comparisons is presented by Stephan *et al.* in [14]. Test generation for stuck-at faults given hierarchical or register-transfer level descriptions has been a subject of considerable attention. We review relevant previous work in hierarchical test generation in Section II.

We describe our overall strategy toward functional vector generation targeting selective path coverage in Section III.

The primary focus of this paper is the HDL-satisfiability checking algorithm. We first describe the basic 3-SAT formulation due to Larrabee [8] in Section IV. The HDL-satisfiability algorithm is described in detail in Section IV.

The public-domain 3-satisfiability (3-SAT)-based test pattern generator from [14] incorporates many different heuristics for fast 3-SAT checking. We have augmented this 3-SAT framework, and used a commercial linear-programming (LP) solver to implement our algorithm. Experimental results which show significant speedups over gate-level satisfiability checking are included in Section V.

II. Previous Work

A. Hierarchical Test Generation

Roy and Abraham [11] [12] use data flow descriptions to perform hierarchical test generation. Data flow graphs are recursively selected for propagating or justifying signals. The assumption of precomputed test sets for modules is adopted. However, conflicts due to reconvergence are not handled for fault effect propagation.

The use of nonlinear equation solving has been proposed in [7] and [9]. Both approaches work significantly better than gate-level approaches in some cases, but suffer from some drawbacks. Finding the solution to the assembled system of equations is not guaranteed, even if a solution exists. Further, the approaches have not been shown to work for circuits with complicated control and large datapaths.

B. Functional Test Generation for HDL Models

The extended finite state machine model is used to automatically generate functional tests in [6]. The extended finite state machine of [6] allows update transformations that include increment, and

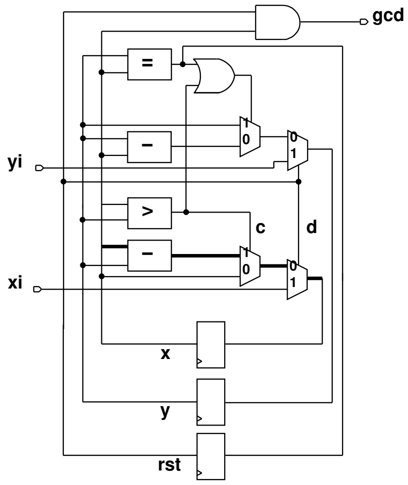

Fig. 1. Module-level circuit for a greatest common divisor example

comparison with a constant. It does not allow addition of several variables, or comparison between several variables.

Constraint solving for test generation is a well-researched problem in the software test area (e.g., [1], [10]). These systems typically attempt to solve nonlinear constraint systems heuristically. In many such systems the logic corresponding to the constraint system is undecidable. The high-level design validation system of [5] uses augmented linear-program solving to solve certain types of constraints. Nonlinear operators in constraints require exhaustive enumeration, random guesses, or the use of heuristics. Bultan *et al.* [4] present a method that uses Presburger formulas as a symbolic representation to perform model checking of infinite state systems. A Presburger solver is used to manipulate these formulas.

III. Functional Vector Generation Strategy

A. Introduction

In this section, we will describe our overall strategy toward functional vector generation given an HDL model. Our strategy involves choosing a coverage metric, setting up search problems given the coverage metric, and solving the search problems to obtain input stimuli. For the purposes of this paper, we have chosen selective path coverage as our metric.

B. HDL Models

We view an HDL model as a structural interconnection of modules. The modules can be comprised of combinational logic, and registers. The combinational logic can correspond to Boolean operators (e.g., AND, OR), or arithmetic operators (e.g., $+$, $>$).

We give an example of a module-level sequential circuit corresponding to a greatest common divisor in Figure 1.

C. Path Testing

We will first focus on combinational circuits. Our strategy of generating functional vectors is to **sensitize** paths from the circuit inputs to circuit outputs. The sensitization of a path can mean many things, and we will define it in the next section. Roughly speaking, sensitizing a path implies that the value at the input to the path should affect the value at the output of the path. An important point to note is that we do not expand busses or word-level variables into bit-level variables. Therefore, a path can be a collection of electrically-distinct signals.

Since there may be exponentially many paths in an HDL model, we will choose a subset. This subset is chosen such that at least one path in the subset passes through each module input to each module output. That is, if we have a module with two inputs A and B, and two outputs C and D, we will choose paths that pass through $A \rightarrow C$, $A \rightarrow D$, $B \rightarrow C$, and $B \rightarrow D$.

There are other criteria in choosing paths, and the quality of the generated vector set is dependent on these criteria. We will target full statement coverage in the HDL model, which can be achieved by sensitizing a set of paths that cover each module-input module-output subpath.

D. Path Sensitization

A path is a set of alternating modules and signals, call it $P = \{s_1,\ m_1,\ s_2,\ m_2,\ \cdots,\ s_n\}$. The first signal, s_1, is a circuit input, and the last, s_n, is a circuit output. For each i, s_i is an input to module m_i and s_{i+1} is an output of module m_i. Sensitizing a path simply corresponds to sensitizing each module m_i.

We now need to define the sensitization of a subpath through a module m_i, namely s_i, m_i, s_{i+1}. Sensitization of a subpath through a module will require values at the module side-inputs, i.e., the inputs other than s_i to m_i.

For logic-gates, we can say the following:

- INVERT: no values required.
- AND: Require side-inputs to be at 1.
- OR: Require side-inputs to be at 0.

The above rules can be used to determine sensitization for arbitrary collections of logic-gates, as well as logic macros such as multiplexors, and decoders.

For word-level operators, sensitization needs to be defined on the basis of the logic function of the operator. Arithmetic operators are easy to sensitize, we merely need to check for overflow or underflow. In the sequel, assume we wish to sensitize the path from A to the module output. Sensitization should reflect the condition that a (large enough) change in the value of A should result in a change in the module output.

- Adder $A + B$: If we are not concerned with overflow, there is no requirement. If we are, then we require $A + B < MAX$, where MAX is the largest represented number.
- Increment $A + k$: To avoid overflow $A < MAX - k$.
- Subtractor $A - B$: Similar to an adder, except we are concerned with underflow.
- Comparator $A > B$: We require $B < MAX$. Similarly, for other types of comparators.
- Scalar multiplication $A * k$: $A < \frac{MAX}{k}$ to avoid overflow.

While the above sensitization requirements for logic-gates and word-level arithmetic operators may appear simple, the reader should note that we are asking for requirements on intermediate signals in the circuit. Therefore, we may ask for a Boolean signal c to be a 1, which is the output of a comparator $A > B$, where $A = C + D$. (C and D may be circuit inputs or intermediate signals themselves.)

Our method of finding input values that sensitize paths is to:

1. Write sensitization requirements on (intermediate) signal values as described above.
2. Write module-input module-output relationships for **every** module in the circuit (cf. Section IV), and
3. Solve a satisfiability problem that corresponds to 1 and 2 above (cf. Section IV). This produces an input assignment that satisfies the sensitization requirement and is consistent with the behavior of the logic circuit.

We could easily choose different sensitization criteria without changing the overall framework described immediately above.

E. Sequential Circuits

We have considered only combinational HDL models thus far. In general, we will have sequential models such as the one in Figure 1, where registers are intermixed with Boolean and arithmetic operators. Our approach to handling sequential circuits is to use the conventional time-frame expansion strategy of sequential test generation [2].

A path may be selected from circuit inputs/register outputs to circuit outputs/register inputs. Once the satisfiability problem for path sensitization is set up and solved, we will require the circuit inputs to be at particular values, and we may require particular $\{0, 1\}$ values at the register outputs as well. (It may be that certain circuit inputs and register outputs can be left undefined.)

We now need to **justify** the required values at the register outputs. This is done using a time-frame expansion strategy. We check to see what the initial values of the registers are. If the initial values are consistent with the required values, there is no need for justification. Otherwise, we attempt to determine circuit input values that produce

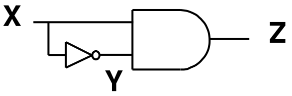

Fig. 2. Simple Circuit

the required values at the register *inputs*, so, in the next cycle, the register outputs have the right values. This is done by solving a new satisfiability problem. If no satisfying assignment can be found, we time-frame expand the circuit over 2 cycles, and attempt to find a sequence of input values of length 2 that produces the required values at the register inputs, solving a (larger) satisfiability problem. This process continues until a user-defined limit on the number of time-frames is reached.[1]

The vector generation method we currently use does not attempt to propagate values at register inputs to circuit outputs. Given an HDL model the registers are all assumed to be observable during functional simulation. If the user wishes to only observe circuit outputs, then additional steps of satisfiability are required to accomplish propagation.

F. Satisfiability Problems

As was described above, to generate vectors that sensitize a single path, we have to solve several satisfiability problems for a sequential HDL model. Further, the satisfiability problems encountered in the justification (and propagation) steps may result from a time-frame expanded circuit, which is larger than the original, since combinational logic is replicated. Thus, it is very important that the satisfiability algorithm be as powerful as possible. We describe our algorithm in the next section.

IV. A Hybrid Algorithm for Satisfiability Checking

A. Introduction

We will describe our hybrid algorithm for satisfiability checking in this section. This algorithm seamlessly integrates linear-programming feasibility and 3-satisfiability checking. Integration would be easy if there was no correlation between word-level variables and Boolean variables. But, in general, a word-level variable A may appear in an arithmetic operation (e.g., $C = A + B$), and a bit-masked version may appear in a Boolean operation (e.g., $d = a_{n-1} \vee b_0$, where $\vee$ denotes logical OR).

B. The 3-Satisfiability Approach

The 3-Satisfiability (3-SAT) approach to stuck-at fault test generation was pioneered by Larrabee [8]. Here, we are not interested in stuck-at fault testing, but rather in justifying values on a collection of signals. That is, given a logic circuit, we wish to find an input assignment that produces appropriate signal values at circuit outputs, or intermediate signals. The 3-SAT approach converts this problem into one of satisfying a conjunctive normal form (CNF) expression. Each of the clauses in the CNF have at most 3 variables in them, hence the name 3-SAT.

Consider the circuit of Figure 2. Assume we wish to find an input assignment that sets the output to a 1. We will write clauses for each gate in the circuit. These clauses model the input-output relationship for each gate, and are shown below.

$$(\overline{X} \vee \overline{Y}) \cdot (X \vee Y) \cdot (\overline{Z} \vee X) \cdot (\overline{Z} \vee Y) \cdot (\overline{X} \vee \overline{Y} \vee Z). \quad (1)$$

In order to produce a 1 at the output of the circuit we need to satisfy the CNF above with Z set to 1. This simplifies to

$$(\overline{X} \vee \overline{Y}) \cdot (X \vee Y) \cdot X \cdot Y, \quad (2)$$

which is clearly not satisfiable.

[1]The theoretical upper bound on the number of time-frames that may be required grows exponentially with the number of register bits in the circuit.

C. New Algorithm for Satisfiability

C.1 Problem

We assume that we are given an arbitrary interconnection of combinational modules which include Boolean and word-level operators:

- AND, OR and NOT gates
- Comparison: $>, \geq, <, \leq$
- Addition/subtraction: $+/-$
- Increment/decrement: $+/-\ k$
- Scalar multiplication: $*k$
- Left and right shift: $<<\ k$, $>>\ k$

Any other word-level or Boolean operator that is not one of those listed above has to be converted into a collection of the listed word-level and Boolean operators. Note that the word-level operators are all linear.

We are given a required set of values for an arbitrary subset of module outputs, and we may be given additional constraints as described in Section III-D. Our task is to find an input assignment that produces the required output values, or prove that none exists. We call this the **HSAT** problem.

C.2 Variables

There are two kinds of variables, Boolean variables, and word-level variables or integers. The word-level operators operate on word-level variables, whereas the Boolean operators work on Boolean (single-bit) variables.

It is possible that Boolean operators are applied to word-level variables. For example, a word-level variable A may operated upon as $A + B$, and also as $A\ \&\ C$, where & corresponds to a bitwise Boolean AND. To represent the bitwise AND operator we will write $O(n)$ **Boolean clauses** over the bits in A and C, namely, $a_{n-1}, a_{n-2}, \cdots, a_0$, and $c_{n-1}, c_{n-2}, \cdots, c_0$. Here a_{n-1} is the most significant bit in A, and a_0 the least significant bit.

We will use the word A in a **linear arithmetic constraint** (LAC) to model the word-level operations on A. We will "remember" the correlation/equivalence between A and the a_i, but we will represent the two separately throughout the algorithm.

The **range** of a variable is the set of values that the variable can take. For example, an integer variable A may take on the values -15 to $+15$. A Boolean variable a_i can only take on the values $\{0, 1\}$.

Given the set of combinational modules listed in Section IV-C.1, for each module we will write a set of Boolean clauses defining the relationship between the inputs and outputs of that module. In the next two sections, we describe the clauses for Boolean operators, and linear constraints for word-level operators, respectively.

C.3 Boolean Clauses for Boolean Operators

We write 2-SAT and 3-SAT clauses for each logical gate as described in Section IV-B. For example,

$$z = AND(x, y): \quad (\overline{z} \vee x) \cdot (\overline{z} \vee y) \cdot (\overline{x} \vee \overline{y} \vee z) \quad (3)$$

D. Linear Constraints for Word-Level Operators

We write linear constraints for each of the word-level operators.

$$C = A + B: \quad A + B - C \leq 0 \ \ and\ \ A + B - C \geq 0 \quad (4)$$

$$C = A + k: \quad C - A \leq k \ \ and\ \ C - A \geq k \quad (5)$$

Similarly for $-$, and $-k$. For scalar multiplication, we have:

$$C = A * k: \quad C - kA \leq 0 \ \ and\ \ C - kA \geq 0 \quad (6)$$

The shift operators $<<$ and $>>$ can be viewed as scalar multiplication. In the case of shifting in 1's we may need to add a constant to the above equations. We can also model cyclic shifts, by adding the most significant bit a_{n-1} to the equations.

Now consider $c = A > B$. Note that c is a Boolean variable. In the sequel, U refers to 2^n where n is the maximum number of bits in A or B. We will write

$$A - B + U(1 - c) \geq 1. \quad (7)$$

In the above equation, if c is 1, then the constraint becomes $A - B \geq 1$, i.e., $A > B$. If c is 0, then we have $A - B \geq 1 - U$, which is always true, given that U is large enough. We also write,

$$A - B - Uc \leq 0. \tag{8}$$

If c is 0, then the constraint becomes $A - B \leq 0$. If c is 1, then the constraint is $A - B \leq U$, which is always true.

Thus, the two constraints above will express the input-output relationships of a $>$ comparator. We can write similar pairs of constraints for $\geq$, $\leq$, and $<$ comparators.

Nonlinear operators such as integer multiplication can be decomposed into linear operators. We can write $4n + 2$ linear constraints for an n-bit multiplier, $Z = X * Y$, by expanding the bits of one of the operands as shown below.

$$Z \geq \sum_{i=0}^{n-1} 2^i p_i, \qquad Z \leq \sum_{i=0}^{n-1} 2^i p_i$$

$$p_i - Ux_i \leq 0, \qquad 0 \leq i < n$$

$$p_i + U(1 - x_i) - Y \geq 0, \qquad 0 \leq i < n$$

$$0 \leq p_i \leq Y, \qquad 0 \leq i < n \tag{9}$$

Consider the modulus operator $Z = X \ mod \ m$, where m is a constant. We will model it using the following equations and variables.

$$X - Z - mY \leq 0$$

$$X - Z - mY \geq 0$$

$$0 \leq Z \leq m - 1$$

$$0 \leq Y \leq \lfloor \frac{maximum \ value \ of \ X}{m} \rfloor \tag{10}$$

We can handle integer division in a similar manner.

E. Transformation of LAC

Given an arbitrary set of modules, we have three types of constraints:

- 2-SAT clauses
- 3-SAT clauses and
- Linear arithmetic constraints (LAC).

The 2-SAT and 3-SAT clauses are over the Boolean variables. The LAC are over the word-level variables, and some Boolean variables.

HSAT = 2-SAT + 3-SAT + LAC.

In this section, we describe a transformation that allows the integration of linear arithmetic constraints and Boolean clauses in our satisfiability algorithm.

Note that there is correspondence or correlation between the word-level variables and the Boolean variables. For example, we may have A appear in an LAC, and a_i, a bit in A, appearing in a 2-SAT clause. We have to make sure that values of word-level variables in the LACs are **consistent** with the values of constituent bits in the Boolean clauses.

Initially, we have all the variables **unbound** or set to unknown values. During our search (cf. Section IV-E.1), we may set Boolean variables or word-level variables. However, if a word-level variable W is correlated to a Boolean variable w_i, we will only set the Boolean variable to $\{0, 1\}$, and transform the LACs that contain W (cf. Section IV-E.1). We describe the transformation below.

Assume that $0 \leq W \leq 2^n - 1$, i.e., W is a positive integer n bits wide. Suppose w_0 is set to 1 in W. Then, we introduce a new variable, call it, V, and we write $2V + 1$, instead of W. The range of V is smaller than that of W, and it is $2^{n-1} - 1$. In general, given a word-level variable W, and an arbitrary number of Boolean variables (bits) in W set to 1 or 0, we can always write a **linear expression** for the arithmetic value for W using

$$W = 2^{n-1} w_{n-1} + 2^{n-2} w_{n-2} + \cdots + 2w_1 + w_0$$

If W is of n bits, and we set r bits in W, then we will need in the worst case, $r + 1$ new word-level variables, assuming none of the r bits are adjacent. As an example, consider a four-bit W, with w_2 set. We may transform it to $8X + 4 + Y$, where $X \leq 1$, $Y \leq 3$. X is effectively a Boolean variable, and Y is a two-bit integer. We no longer need W, so the increase in the number of variables is at most r.

For sign/magnitude or two's complement integers, we merely use the appropriate equation that corresponds to the arithmetic value of the integer based on the individual bits. For example, the two's complement equation for an $n + 1$-bit number W, whose most significant bit is the sign bit, would be:

$$W = -2^n w_n + 2^{n-1} w_{n-1} + 2^{n-2} w_{n-2} + \cdots + 2w_1 + w_0$$

Thus, given an arbitrary setting of Boolean variables that correlate to word-level variables in an LAC, we can modify the LAC into another LAC using the method outlined above. The left-hand-side of the LAC is modified and the right hand side remains the same. Simple algebraic manipulation is applied to normalize the LAC, i.e., to convert it to a standard form $\sum_i a_i w_i \leq C$ or $\sum_i a_i w_i \geq C$. The modified LAC will typically have more variables than the original LAC, however, the ranges of the variables will be smaller. Further, there is no exponential increase in the number of variables or constraints; the total number of constraints and the total number of variables in the LAC will grow no more than the number of Boolean variables that are set to 0 or 1.

The above transformation is key to the seamless integration of Boolean clauses and LAC in our hybrid satisfiability algorithm. We next describe the satisfiability search which uses the above transformation, so word-level operators are only expanded when needed, resulting in greater efficiency.

E.1 Satisfiability Search

If the LAC is empty, we can check for various special cases in the 3-SAT clauses.

Essential variable : If a clause contains a single variable in true form (complemented form), clearly the variable has to be set to 1 (0) for the clause to be satisfied.

Dominating Clause : If two clauses $(a \vee b)$ and $(a \vee b \vee c)$ exist, we can delete the latter, since the former implies the latter.

Unate Variable : If a variable a appears only in true form, (complemented form) in all the clauses, we can immediately set the variable to 1 (0).

There are many other optimizations that can be applied for the 3-SAT problem, see [14].

In the general case, the LAC will be non-empty. We can still check for essential variables, and delete dominating clauses. However, we cannot set variables that are unate with respect to the Boolean clauses without checking for correlation in the LAC. For example, if we have

$$(w_0 \vee a \vee b)$$

$$W + X \leq 5$$

with w_0 only appearing in the one clause shown, we still cannot assume that $w_0 = 1$, because w_0 is correlated with W. In order to satisfy the clauses and LAC, we might need $w_0 = 0$, $a = 1$.

We will perform the standard branching search to find a satisfying assignment. We will only set variables corresponding to inputs to the circuit, and not intermediate signals. (The intermediate signals may be set by implication on the clauses.) We first perform special case pruning, such as the checks outlined above. There is a lot of structural information that can be exploited to simplify the 3-SAT clauses. The reader is referred to [8] and [14] for details.

Regardless of correlation among bit-level and word-level variables, we can assert the following:

- Polynomial-time checks using cycle detection can be performed to see if 2-SAT + 3-SAT is infeasible [8]. If 2-SAT + 3-SAT is infeasible, so is HSAT. Note that for 3-SAT even if cycle detection fails, the problem may still be infeasible.
- We can run full-fledged CNF satisfiability algorithms with a limit on the number of backtracks in search, to possibly prove infeasibility of 3-SAT.

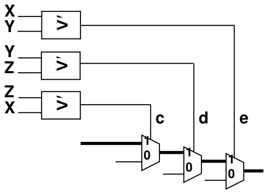

Fig. 3. A False Path

- The linear program relaxation (LPR) of the LACs corresponds to relaxing the assumption that the word-level variables are integral. (That is, they can be real.) If the LPR corresponding to HSAT = 2-SAT + 3-SAT + LAC is infeasible, then the HSAT problem is infeasible.

If the problem is not infeasible, we will heuristically select a Boolean variable, or a word-level variable. If we select a Boolean variable, we set it to 1, modify the clauses and LACs appropriately, and recursively call the satisfiability procedure. If we fail in finding a satisfying assignment, we will set the Boolean variable to 0, and recur. The selection of the variable is dependent on how many clauses and LACs it appears in. In the case of selecting a word-level variable, we check to see that no unknown Boolean variable correlated with that word-level variable exists. (If so, we will select another word-level variable, or the Boolean variable.) We split the range of the word-level variable and recur on the two smaller ranges. The split is based on the value of the variable returned by the LPR. That is, if the word-level variable's range is $[-7, 7]$, and the LPR returns a value of 4.5 for the variable, we will recur on the $[-7, 4]$ and $[5, 7]$ ranges. The only modification we need to perform, in this case, is to the LAC.

It is possible that LPR may return a feasible solution, but no integer solution may exist. If the LPR is feasible, the solution is integral, and is consistent across the word-level and bit-level variables, then we have a feasible solution to HSAT, and we can stop. When LPR returns a feasible, non-integral solution, we perform a simple perturbation of the the solution returned by LPR to see if we can make it integral, while satisfying the consistency requirements. If so, we have found a feasible solution to HSAT.

The CPU time allowed for search is limited by limiting the number of backtracks. Given a certain number of backtracks, three things may happen:

1. We find an input assignment that produces the required output values.
2. We determine that no input assignment that satisfies the required values exists.
3. We may run out of backtracks before 1 or 2 occurs above. In this case we have failed.

F. Examples

Consider the bold path in Figure 1. Assume we wish to sensitize this path. There are three requirements:

- Assuming unsigned numbers, to avoid underflow in the subtractor, we will require that `x >= y`.
- We will require `c = 1`. This will translate to `x > y` when the module-input module-output relationships are included.
- We will require `d = 0`. This directly translates to `rst = 0`.

Another example is depicted in Figure 3, where LPR quickly detects infeasibility, but gate-level satisfiability typically requires a large number of backtracks to prove infeasibility. The bold path in the figure is **false**, i.e., there is no assignment to the word-inputs X, Y, and Z that sensitizes it. One call to LPR will determine this. If we use gate-level satisfiability, the number of backtracks will depend on the order in which the bits of X, Y, and Z are assigned to 0/1 values. If the words are n bits wide, and we begin with the least significant bits, we will require $O(2^{3n})$ backtracks to establish infeasibility. If the most significant bits of X, Y, and Z are set first, gate-level satisfiability will still require $O(2^n)$ backtracks, because infeasibility will not detected for the $x_i = y_i = z_i = 1$, or $x_i = y_i = z_i = 0$ cases. It is possible that bits of word X or Y or Z appear in Boolean clauses due to additional circuitry. However, since we take correlation into account during the search, appropriately modifying the LAC will allow us to use LPR and outperform gate-level satisfiability.

If we use ordered Binary Decision Diagrams [3], under the right ordering, the BDDs for the outputs of the comparators will grow linearly with bit-width. To determine infeasibility we have to compute the intersection of the three BDDs, and the time required for this will grow super-linearly with bit-width. Further, for more complex sets of (linear) arithmetic constraints, it is difficult to find an ordering (assuming one exists) that results in BDDs of reasonable sizes.

Satisfiability problems generated during the justification step are similar to those generated in the path sensitization step. We have the module-input, module-output relationships, and additionally a requirement of particular values on register inputs.

V. Experiments and Comparisons

We present two sets of results on some example HDL models. The HDL models correspond to a parallel port for an embedded processor (`pport`), various finite state machines with some data operations, notably comparisons (`schsm`, `ctla`, `ctlbc`), and two protocols (`protrisc` and `protmpp`). The HDL models are in both VERILOG and VHDL and their size varies between 100-500 lines of code. We also present an example that corresponds to checking if the output of a 16-bit multiplier can be a prime number (`mult16`), restricting the operands to be different from 1.

A. Satisfiability

The first set of results compares our new HSAT algorithm to gate-level 3-SAT(GSAT). Our HSAT algorithm is implemented using the 3-SAT package from [14] and a commercial LP solver CPLEX. We compare the performance of HSAT to vanilla 3-satisfiability using the software from [14], adapted to read CNF expressions. During the process of vector generation, many satisfiability problems are solved. For the different HDL examples, we give data on the **hardest** satisfiability problem that was successfully [2] solved by HSAT.

In Table I we give the quantitative comparison between HSAT and GSAT. For HSAT, we report the number of clauses (2-SAT or 3-SAT), the number of LACs, the number of backtracks, and the amount of CPU time required for vector generation. For GSAT, we report the number of clauses, number of backtracks, the CPU time required to expand the LACs into Boolean clauses, and the CPU time required to solve the satisfiability problem. To compare overall performance the reader should compare the sum of the GSAT times against the HSAT time, since in order to run GSAT we have to perform the expansion of the LACs into clauses.

Just comparing the time for satisfiability checking, HSAT performs significantly better than GSAT for all the satisfiability problems. The improvement is due to the significant reduction in the number of backtracks, which directly translates to faster execution time. The reduction in backtracks is due to the LP feasibility check which prunes unsatisfiable branches of the tree early in the recursion. The CPU time required **per backtrack** is significantly higher for HSAT as compared to GSAT because we are performing an LP solve at each step. Note that Table I corresponds to the hardest satisfiability problems we encountered in vector generation. Many of the satisfiability problems encountered in vector generation are simple, and both HSAT and GSAT find satisfying assignments without backtracking. We did not encounter any satisfiability problems where GSAT was successful, and HSAT failed.

Other satisfiability approaches such as GRASP [13] may outperform GSAT in some cases; however, these approaches still have to deal with a substantially larger problem than HSAT, and do not use LPR feasibility checking. We were unable to create ordered Binary Decision Diagram [3] representations for the circuits corresponding to the large examples. Finally, one can envision using commercial ILP solvers once the problem has been set up, but this is also inefficient because any integer variable that appears in a Boolean clause has to be expanded in the initial problem specification; there is no notion of remembering correlation between different representations of a variable in ILP formulations.

[2] By successfully we mean that HSAT either found a satisfying assignment, or proved that none existed within the allotted number of backtracks.

Model	HSAT				GSAT			
	#clauses	#LACs	#backtracks	Sat. time	#clauses	#backtracks	Exp. time	Sat. time
pport	17	18	0	< 1 s	685	2,604	1.2s	4.7s
schsm	141	24	0	< 1 s	841	35,102	3.1s	261s
ctla	1	24	1	< 1 s	1355	> 100,000	12.0s	> 1000s
ctlbc	1	60	1	< 1 s	1341	17,477	4.9s	142s
mult16	0	104	4158	16 s	-	-	-	-

TABLE I
COMPARISON BETWEEN SATISFIABILITY APPROACHES

Model	Satisfiability Test			Random Test	
	#Vec	Cov %	Time	#Vec	Cov %
pport	23	90.9	34s	2000	90.9
schsm	24	88.1	56s	2000	85.6
ctla	66	100.0	146s	2000	91.2
ctlbc	32	100.0	82s	1157	100.0
protrisc	46	94.7	246s	2000	78.2
protmpp	76	100.0	140s	2000	64.0

TABLE II
COMPARISON BETWEEN TEST GENERATION APPROACHES

B. *Directed Test Generation*

We next report results on comparing the vector generation strategy against random pattern generation. The data is reported in Table II. For each HDL example, we give the number of vectors in the vector sequence generated by the HSAT algorithm, the amount of CPU time required, and the percentage of HDL statements exercised by the vectors. For random vector generation, we give the **best** coverage obtained by simulating 10 sets [3] of 2000 random vectors. CPU time for random vector generation and simulation grows proportionately with the number of vectors, and is not reported.

The directed vector generation method produces as good, or higher coverage, using a much smaller number of vectors in all cases. The circuits were initially assumed to be in a known state, since this makes it easier to generate random patterns. (If the circuit were in an unknown state, random pattern generation gives significantly worse results. However, using HSAT we can put the circuit into a known state and then perform vector generation.)

The CPU times in both tables correspond to seconds on an Ultra-Sparc.

VI. ONGOING WORK

The two primary reasons why the hybrid satisfiability algorithm may fail for a particular problem are nonlinear operations and large sequential depth. In our methodology, nonlinear operators have to be decomposed into linear arithmetic or Boolean operators. We have choices in the decomposition process. For example, we may choose to decompose nonlinear multiplication $A * B$ into a large set of Boolean clauses. However, it is usually more efficient to transform $A * B$ into linear constraints as described in Section IV-D. Being able to perform the "optimal decomposition" is desirable.

Time-frame expansion works well for circuits with small sequential depth such as datapaths, and small finite state machines. One case where time-frame expansion may become prohibitively expensive is when there are timers or counters in the circuit. For example, we may have a 16-bit counter whose output is compared to 65535 to generate a Boolean signal. It is possible that we have to unroll the counter 65535 times in order to find the appropriate input stimuli. Unrolling the next state logic k times can be done efficiently by turning a $+1$ increment operator into a $+k$ increment operator. However, any reset logic in the counter may have to be explicitly unrolled k times to find the input conditions that force reset in the appropriate clock cycle. Methods to alleviate this computational expense are a subject of ongoing work.

[3] These sets were generated using different seeds.

ACKNOWLEDGEMENTS

F. Fallah and S. Devadas were supported in part by a grant from Schlumberger Foundation.

REFERENCES

[1] G. S. Avarunin, L. K. Dillon, and J. C. Wileden. Experiments with Automated Constrained Expression Analysis of Concurrent Software Systems. In *Proceedings of ACM SIGSOFT 89, 3^{rd} Symposium on Software Testing, Analysis, and Verification*, pages 124–130, December 1989.

[2] M. A. Breuer and A. D. Friedman. *Diagnosis and Reliable Design of Digital Systems*. Computer Science Press, 1976.

[3] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] T. Bultan, R. Gerber, and W. Pugh. Symbolic Model Checking of Infinite State Systems Using Presburger Arithmetic. In *Proceedings of the Workshop on Computer-Aided Verification*, pages 400–411, 1997.

[5] A. K. Chandra and V. Iyengar. Constraint Solving for Test Generation: A technique for high-level design verification. In *Proceedings of the Int'l Conference on Computer Design*, pages 245–248, October 1992.

[6] K-T. Cheng and A. S. Krishnakumar. Automatic Functional Test Generation Using the Extended Finite State Machine Model. In *Proceedings of the 30^{th} Design Automation Conference*, pages 86–91, June 1993.

[7] A. Ghosh, S. Devadas, and A. R. Newton. *Sequential Logic Testing and Verification*. Kluwer Academic Publishers, Boston, Massachusetts, 1992.

[8] T. Larrabee. Test Pattern Generation Using Boolean Satisfiability. *IEEE Transactions on Computer-Aided Design*, 11:4–15, January 1992.

[9] J. Lee. *Architectural Level Test Generation and Fault Simulation*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[10] K. M. Olender and L. J. Osterweil. Cesar: A Static Sequencing Constraint Analyzer. In *Proceedings of ACM SIGSOFT 89, 3^{rd} Symposium on Software Testing, Analysis, and Verification*, pages 66–74, December 1989.

[11] K. Roy. The Use of RTL Descriptions in Accurate Timing Verification and Test Generation. *IEEE Journal of Solid-State Circuits*, 26(9):1230–1239, September 1991.

[12] K. Roy and J. Abraham. High Level Test Generation Using Data Flow Descriptions. In *Proceedings of the European Design Automation Conference*, pages 480–484, March 1990.

[13] J. Silva and K. Sakallah. GRASP: A New Search Algorithm for Satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[14] P. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. Combinational Test Pattern Generation Using Satisfiability. *IEEE Transactions on Computer-Aided Design*, 15(9):1167–1176, September 1996.