

RACE: A WORD-LEVEL ATPG-BASED CONSTRAINTS SOLVER SYSTEM FOR SMART RANDOM SIMULATION

Mahesh A. Iyer

Synopsys, Inc. Mountain View, CA, USA

Abstract

Functional verification of complex designs largely relies on the use of simulation in conjunction high-level verification languages (HVL) and test-bench automation (TBA) tools. In a constraints-based verification methodology, constraints are used to model the environmental restrictions of the Design Under Verification (DUV), and are specified using HVL constructs. The job of a constraints solver is to produce multiple random solutions to these constraints. These random solutions are used to drive legal random stimulus to the DUV using procedural HVL constructs.

This paper presents RACE (**R**andom **A**TPG for solving **C**onstraint **E**xpressions), a new word-level engine for solving combinational constraint expressions. RACE builds a high-level netlist model to represent the constraints and implements a branch-and-bound algorithm to solve them. Advanced interval arithmetic concepts and algorithms are used to propagate word-level values across a wide variety of high-level operators. RACE is architected to produce multiple random solutions for the same constraints problem, and has been successfully used for random stimulus generation within a commercial TBA tool [14] for Register Transfer Level (RTL) verification of complex designs using simulation.

1. Introduction

Functional verification is typically performed at the RTL to ensure that the behavior of the RTL design matches the specifications of the design. Functional verification is widely acknowledged as the bottleneck of the hardware design cycle [3]. To keep pace with today's design verification challenges, only a few iterations for debugging are acceptable as time-to-market windows shrink.

Simulation is the most prevalent and proven technique that is used in the industry today for functional verification. However, simulation alone does not solve the functional verification problem. Complex designs require higher levels of abstraction to model their input/output temporal behavior; this is accomplished through the use of HVL and TBA tools that implement and support these languages. A high-level view of this commonly used verification methodology is depicted in Figure 1.

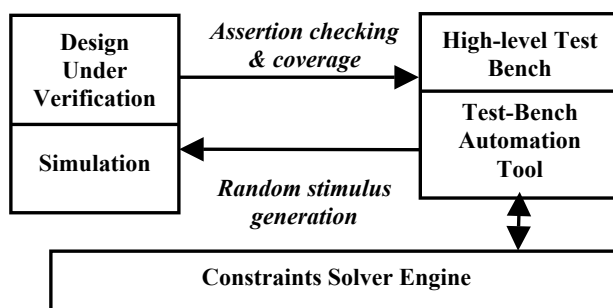


Figure 1 Functional Verification Methodology

The high-level test-bench is written in the HVL that is supported by the underlying TBA tool. While RTL test-benches are also commonly used, HVL test-benches provide the high-level abstraction that is needed to model complex behaviors of the DUV. These could include transactors that reason about the behavior of the design at the transaction level or bus functional models that deal with bus protocols as opposed to analyzing individual bits of a bus. In essence, a high-level test-bench models the behavior of the DUV at the system level and has two main goals.

- Apply random stimulus and/or directed tests to the DUV by interfacing to a simulator.
- Analyze results from the simulation and perform assertion checking and measure functional coverage as defined in the test-bench.

Most designs have assumptions on how they interact with their environment. A goal of the test-bench is to generate random stimuli to the DUV that satisfy these environmental restrictions. Consequently, most commercial and industry-standard HVL provide means to specify constraints on certain variables (that are declared within the test-bench) and the ability to randomize a certain set of these variables upon demand. The constraints typically consist of arbitrary expressions on signed or unsigned variables with varying bit-widths using a wide variety of high-level operators that the HVL supports. The results of this randomization are eventually translated (through procedural code within the test-bench) as legal random stimuli to the DUV. When control is handed over to the simulator, the newly generated stimulus is simulated until there are no more events for the simulator in the current clock cycle. At this

point, the control is handed back to the test-bench, which asserts on certain properties being true (based on the current signal values of the DUV) and measures its view of functional coverage (as defined in the test-bench). It then generates the next set of random stimuli to the DUV and the process continues, until the test-bench decides to stop.

In order for this verification methodology to reveal corner-case bugs, the random stimuli to the DUV should have good distributions. Consequently, most HVL also provide means to specify desired distributions on random variables. This functional verification methodology is most commonly used for system-level verification. (Formal and semi-formal verification techniques are more popular at the block-level.)

Smart constrained-random simulation has been identified as a key solution to meet functional verification challenges. The TBA tool thereby needs an efficient constraint solver system, which has the following key requirements.

- Ability to deal with large number of complex intertwined constraints. Modern designs tend to be fairly complex with complex input environmental restrictions.
- Ability to deal with a variety of high-level operators.
- Ability to produce good random distributions.
- Ability to deal with wide bit-widths. Most HVL are highly object-oriented. Verification engineers very commonly use wide bit-widths in their test-bench code, because the test-bench typically tends to model and verify very high-level behaviors of the DUV. Another subtle reason is that it is natural for engineers to (mis)use the power of the HVL. For example, a particular constraint may need only 4 bits, but for coding convenience, a 32-bit integer may have been used for this constraint instead.
- Seamless solution for signed and unsigned semantics that matches RTL semantics.
- High performance with high capacity.

To address these requirements, this paper presents RACE, a new word-level Automatic Test Pattern Generation (ATPG)-based combinational constraints solver system. RACE builds a high-level netlist model to represent the constraints and implements a branch-and-bound algorithm to solve them. RACE compactly represents values on variables with large bit-widths and uses advanced interval arithmetic concepts and algorithms to propagate values across a wide variety of high-level operators in the netlist. Several deterministic heuristics guide the search-space exploration algorithm in RACE, while maintaining the ability to generate random solutions with good distributions. The ability to generate multiple random solutions with good distributions for the same constraint problem is a key requirement in constrained random

simulation, and is also a key feature that differentiates RACE from some state-of-the-art constraints solvers, boolean satisfiability (SAT) solvers and gate-level ATPG tools.

RACE has been successfully used for random stimulus generation within a commercial high-level TBA tool [14] for RTL Verification of complex designs using simulation. Experimental results on industrial test cases demonstrate the effectiveness of RACE.

The rest of the paper is organized as follows. Section 2 formulates the combinational constraints problem. Section 3 reviews related prior art. Section 4 describes the RACE algorithm. Experimental results are presented in Section 5. Section 6 highlights some limitations of RACE and future directions in the area of constrained-random simulation. Finally, Section 6 concludes the paper.

2. Problem Formulation

The combinational constraints solving problem can be formulated as follows. Consider a set of variables, $V = \{v_1, v_2, \dots, v_n\}$, and a set of relations or constraints, $C = \{c_1, c_2, \dots, c_m\}$, such that each constraint is a relation defined between expressions over a subset of the variables in V . The variables in V have defined bit-widths and sign, which dictate range of values that could possibly be assigned to them. Every variable in V could be either a random or a state variable. The constraints solving problem comprises finding legal assignments to all random variables in V for particular values of state variables in V , such that all the constraints in C are satisfied.

Example 1: Consider the following constraints problem.

```

rand bit[3:0] a, b;
rand integer c;
constraint c1 {
    a + b == c * 2;
}
constraint c2 {
    c > 0;
}

```

Here a and b are 4-bit unsigned random variables and c is a 32-bit signed random variable. Thus, a , b , and c can possibly take 16, 16 and 2^{32} values respectively. $c1$ and $c2$ represent constraints over variables a , b , and c . The constraints solving problem comprises finding legal values for a , b , and c , that satisfy $c1$ and $c2$. One such solution is $a = 1, b = 3, c = 2$. \square

3. Related Work

In the last decade, several techniques have been proposed to solve the combinational constraints problem. This Section attempts to review some prominent works in this area as applied to functional verification, without

necessarily exhausting all prior art in constraint logic programming.

Yuan *et al.* [15] have proposed a constraints solving system, SimGen, that uses Binary Decision Diagrams (BDDs) [5]. Their method unified the handling of constraints and user-defined biasing through the use of constrained probabilities. BDDs are built for the constraints and an algorithm is applied to bias the branching probabilities in the BDD. During simulation, this annotated BDD is used to generate input vectors whose distribution matches their predetermined constrained probabilities. This technique has several advantages with respect to simulation performance, because the BDDs essentially represent the entire solution space. Consequently, during simulation, substituting the state variable values and traversing the BDDs is a very efficient operation. Furthermore, the knowledge of the entire solution space enables this technique to generate desired distributions accurately. However, it is important to point out that the technique hinges upon the ability to build the BDDs using reasonable compute resources. Building BDDs for complex, intertwined constraints with large bit-widths can be a daunting task. This fact makes this constraints solving technique attractive for smaller number of constraints that do not use certain types of operators like multipliers and use variables with smaller bit-widths.

Huang and Cheng [9] have proposed an interesting hybrid technique for assertion checking in RTL design verification. Their contribution combines the use of word-level ATPG for control logic and modular arithmetic techniques for data-path logic in the DUV. Their technique is sequential in nature and it analyzes the entire DUV and the specified property assertions to determine if the property holds true for all states from the initial state or if a counter-example exists that violates the property. The technique also uses a 3-valued logic value system and has separate treatment of control and data-path operators. Their technique is applied in the context of assertion checking, which only requires the generation of a single vector sequence to violate each property. This is one aspect that RACE differs in. RACE is used in the context of constrained-random simulation, thereby requiring it to produce multiple random solutions for the same constraints problem.

Another distinguishing factor for RACE is that it uses a simple 2-valued data representation for values using word-level ranges and eliminates the need to represent the don't-care "x" value through the use of over-approximation and efficient value assignment algorithms. RACE does not differentiate between data-path or control operators, and performs word-level ATPG on all supported operators. RACE also supports biasing of random variables through appropriate modeling. This helps control the distribution of

solutions produced by RACE. Another difference is that RACE has a unique netlist modeling technique to model RTL precision and sign semantics.

Other test program generation systems proposed in [2, 6, 7] implement dynamic-biased instruction generation and utilize constraint-solving techniques tailored for specific instructions. Their techniques rely on backtracking and heuristic search techniques to recover from bad decisions.

Bin *et al.* [4] formulated the random test program generation problem in verification as a constraint satisfaction problem and combined several techniques to deal with hard constraint problems that have huge variable domains and to produce good distributions. Their work is of importance because it was applied for specific test generators targeted at various parts of a design and stages of a verification process.

4. The RACE System

This Section describes the main contribution of this paper – **RACE** (**R**andom **A**T**P**G for solving **C**onstraint **E**xpressions).

Gate-level ATPG is a well-understood concept that has become an industry standard for testing manufacturing defects [1]. ATPG techniques have also been successfully used for other Electronic Design Automation (EDA) problems like logic synthesis, equivalence checking, formal verification, and SAT. These techniques deterministically analyze a gate-level netlist to derive tests for faults under a fault model using a branch-and-bound procedure.

The constraints solving problem can be formulated as a SAT problem. The BDD-based technique described in [15] builds the BDDs from a bit-blasted gate-level representation of the constraints problem. So, in principle, we could try solving the constraints problem using ATPG techniques on the gate-level representation of the problem. However, we would encounter several shortcomings with this approach.

- Traditional gate-level ATPG systems do not have the ability to produce many random solutions for the same problem.
- For complex, intertwined constraints, the gate-level netlist could be vastly complex for gate-level ATPG. The complexity explodes considering the need to generate several solutions for the same problem.
- For constrained variables with large bit-widths, each bit of each variable could become a decision variable for gate-level ATPG. The branch-and-bound procedure in gate-level ATPG would very soon become intractable for such situations.
- For a SAT problem, traditional techniques like fault simulation [1] that are used to significantly improve the

performance of gate-level ATPG would not be applicable.

Despite these shortcomings, some core concepts of gate-level ATPG, like backtracking, learning and heuristic search are highly applicable to the constraints solving problem, and precisely what RACE leverages. In recent years, high-level ATPG has gained interest [11] in the context of manufacturing test. However, not many successful solutions have been advertised as yet. One of the key problems with high-level ATPG for manufacturing test is that of determining a good fault model.

RACE is a word-level ATPG-based system that works on a high-level description of the constraints problem. The high-level netlist that RACE creates essentially presents the constraints problem as a SAT problem. Consequently, RACE does not have to deal with any fault model.

4.1 Value Representation

RACE represents word-level values using 3 data structures.

- *DataType* – represents large signed/unsigned numbers. RACE supports large bit-widths up to $(2^{32} - 1)$ bits.
- *Interval* – a pair of $\{low:high\}$ *DataType* members to represent a contiguous interval of values from *low* to *high*, inclusive.
- *Range* – a canonical list of intervals to represent disjoint contiguous intervals of values.

RACE supports a wide variety of operators, which are listed in the next sub-section. For all these operators, RACE implements several algorithms for value propagation using the above data structures. The above data structures are also attractive for set operations like union and intersection. A key feature of this data representation is that it is 2-valued; i.e. there is no notion of a don't-care "x" value. This allows unified support of range and interval propagation algorithms across all the different kinds of operators. Of course, with some operators, the absence of the "x" value results in an over-approximation of values. RACE resolves this problem during ATPG by using efficient value assignment algorithms or by resorting to simple backtracking.

4.2 Operators

RACE supports the following operators that are part of most HVL.

- *Arithmetic*: +, -, *, /, %
- *Relational*: >=, >, <=, <, ==, !=
- *Boolean*: &&, ||, !
- *Bit-wise*: ~, &, &~, |, |~, ^, ^~
- *Shift*: <<, >>
- *Concatenation*: {}, rep{}
- *Unary*: &, ~&, |, ~|, ^, ^~
- *Conditional*: ?:

- *Bit-slice*: []
- *Implies*: =>

For all these operators, RACE implements RTL-Verilog [10] semantics. To help in modeling RTL-Verilog precision, sign and wrap-around semantics, RACE also uses some precision and cast operators.

For each of these operators, RACE implements algorithms to forward and backward propagate range values. These algorithms tend to be pessimistic in the sense that when a compact range value representation cannot be derived (because of algorithmic difficulties or because the number of intervals in the range is huge), the value propagation algorithms would conservatively return the complete range for that node's precision and sign. Refinement of these ranges into singleton values happens during the ATPG process in RACE. However, when a smaller range is computed, RACE guarantees that for the current state of the system, the values that are not in the computed range are definitely not possible. RACE also ensures that propagation of singleton values is accurate and not pessimistic. This is required for correctness of the solution and helps in detecting conflicts in value assignments, so that backtracking can be invoked during ATPG. In practice, conflicts are detected much earlier in the process when all variables don't necessarily have singleton values.

Example 2: Consider a 4-bit unsigned random variable, *a*. The possible values for *a* are $\{0:15\}$. Consider a constraint $(a > 2 \ \&\& \ a < 7) \ || \ (a \geq 11)$. The propagation rules in RACE would shrink the range for *a* to $\{3:6\}, \{11:15\}$. □

4.3 High-Level Netlist Model

From the constraint expressions, RACE builds a high-level netlist model on which it performs its analysis. This netlist is a directed acyclic graph (DAG), similar to the one used in gate-level ATPG, except the nodes in the netlist are high-level operators listed above. Each constraint is independently modeled and all active constraints are simultaneously solved for. No implicit variable ordering or topology is imposed in the netlist structure based on the order or style of the constraints. Variables that participate in multiple constraints have multiple fanouts in the netlist. The netlist model also has precision and cast operators appropriately inserted to support RTL-Verilog semantics. All constraints are conjoined using a Boolean AND gate, whose output has to be justified to value $\{1:1\}$, through assignment of all random variables. A nice property of the RACE netlist model is that its size is linear in the size of the constraint problem. The netlist modeling in RACE is best illustrated with an example.

Example 3: Consider the following constraints.

```
rand bit [3 : 0] a, b;
constraint c1 {
```

```

a + b == 10;
a - b == 6;
}

```

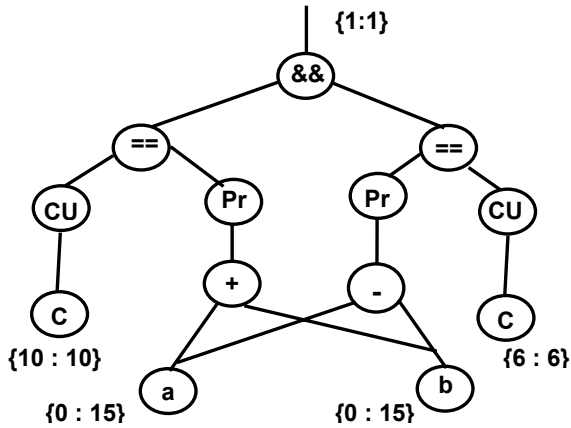


Figure 2 RACE Netlist Model for Example 3

Here, the two constants, 10, and 6, are interpreted as 32-bit signed numbers. The result of the adder and the subtractor are to be interpreted as 32-bit unsigned numbers. This is modeled using Precision operators (*Pr*) on the outputs of the adder and subtractor. The output of the adder and the subtractor are computed using unbounded precision and the *Pr* operators appropriately convert the result to 32 bits. RTL-Verilog interprets mixed signs on the equality operator as unsigned. This is modeled using cast unsigned operators (*CU*) on the output of the constant nodes. As part of the model building process in RACE, the primary input and primary output nodes in the netlist are also initialized with range values. For example, the primary input node, *a*, is initialized with a range {0:15}. Similarly the output node is initialized with a range {1:1}, to model the constraint satisfiability problem. The resultant model and initial value assignments are shown in Figure 2. □

4.4 Implication Engine

A core sub-system within RACE is a fast implication engine. The implication engine essentially propagates range values across the netlist. It manages all the node evaluation scheduling to recursively determine the consequence of a value assignment on a node on other nodes in the netlist. During the implication process, when a node evaluates to a new value, the new value is intersected with the old value of the node. This ensures that the range values will only decrease in size as the implication proceeds. An empty intersection means that an inconsistent state has been reached, typically referred to as a conflict.

While there is a lot of similarity between bit-wise implications across logic gates and range value implications across high-level operators, there is also a distinct difference in terms of implication convergence and node evaluations. With bit-wise implications across logic gates, there is no opportunity for cyclic implications; i.e. if

$a \Rightarrow b$, and there is no conflict on *b*, the new value on *b* has no new implications on *a*. However, implications of range values can have such cyclic behavior, because range values keep shrinking with each new implication. This is demonstrated in the following example.

Example 4: Consider the following constraints.

```

rand bit [3 : 0] a, b;
constraint c1 {
(a + b) > 4'd14;
a < (4'd15 - b);
}

```

For simplicity, assume that all node evaluations are performed with 4-bit unsigned precision semantics. Figure 3 illustrates the high-level netlist model for these constraints.

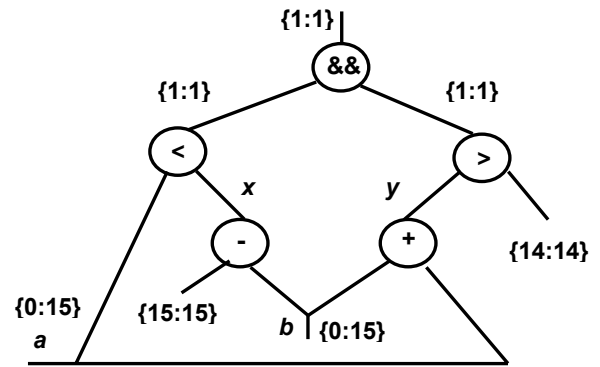


Figure 3 Example to Illustrate Implications

Since $y > 14$, we conclude that $y = \{15:15\}$. Forward evaluating the node *x*, we get $x = \{0:15\}$ (using RTL-Verilog wrap-around semantics). This implies $a = \{0:14\}$ through backward propagation across the < comparator. Backward evaluation to node *b* from the output of the adder using the new value of *a* gives $b = \{1:15\}$. This in turn implies $x = \{0:14\}$. This in turn implies $a = \{0:13\}$. This implication cycle will continue as $x \Rightarrow a \Rightarrow b \Rightarrow x \Rightarrow a \dots$. Note that in every iteration, as nodes get re-evaluated, their range values keeps shrinking, thereby giving rise to new implications. If we follow this implication through, we will discover that the range values on *a* and *b* will become empty, thereby representing a conflict in the constraints. In this particular example, it is also possible to discover such conflicts through static learning heuristics. □

It can be shown that for a particular value assignment, the implication process will converge. However to deal with cyclic implications (as above), the RACE implication engine has some pessimistic criteria to either detect these as conflicts early on or to declare a bail out. The RACE ATPG engine then appropriately interprets the results of the implication engine, depending on the context of the call to the implication engine.

4.4.1 Node Evaluation

Forward evaluation of an operator node is the process of determining a range value for that node's output using the range values of all its input nodes and the semantic functionality of that operator. In RACE, forward evaluation is done using unbounded precision. For example, consider an adder with two 4-bit unsigned inputs that have range values $\{0:15\}$ and $\{5:10\}$. The output of this adder is evaluated and represented in 5-bit precision with the range value $\{5:25\}$. Note that for many operators (for example *boolean*, *relational* and *unary reduction* operators), the range value of the output node can be either $\{0:0\}$, $\{1:1\}$, or $\{0:1\}$. Following RTL-Verilog semantics, this can be represented in 1-bit precision. For example, if we are forward evaluating a $<$ comparator with two input range values $\{0:3\}$, $\{5:6\}$ and $\{9:12\}$ respectively, the result would be $\{1:1\}$. On the other hand if the second range value was $\{6:12\}$ instead of $\{9:12\}$, the result would be $\{0:1\}$.

Backward evaluation to an input of an operator node is essentially the process of determining a range value on that input using the range values of all other inputs of that node, the range value of the output node and the semantic functionality of that operator. To illustrate a simple case of backward evaluation, consider a comparator node c (a node is identified with its output signal), $a < b$, where a and b are 4-bit unsigned inputs to the comparator. Assume that $c = \{1:1\}$, $a = \{0:15\}$, and $b = \{0:15\}$. A backward evaluation to node a implies that $a = \{0:14\}$. Similarly, a backward evaluation to node b implies that $b = \{1:15\}$.

Although RACE has many complex algorithms for backward propagation across different types of operators, as one might imagine, for some operators such backward propagation is limited either because of algorithmic complexities or because of range explosion. For example, using our data representation, it would be hard to perform backward propagation across a bit slice operator. This is illustrated using the following example.

Example 5: Consider a 32-bit integer variable a and let the constraint on a be $a[7:3] == 9$. RACE uses a ternary bit slice operator to represent this constraint, the three inputs being the variable a , a constant node with value $\{7:7\}$ and a constant node with value $\{3:3\}$ respectively. With implications, the output of the bit slice operator would have the range $\{9:9\}$. However, as can be clearly observed, it would be a challenge to backward propagate this range to the input variable a . This would require enumerating all split ranges with those 5 bits to have value 9. The root cause of the problem in this case is the absence of the don't-care "x" value in RACE's range value representation. Consequently, the implication engine simply returns a conservative implication; i.e. the maximum range for a intersected with the current range value on a . □

As much as the absence of the "x" value seems like a limitation, it is actually a strength because it helps in unified treatment of all different operators using the same word-level ATPG technique. This problem of absence of "x" value (as in the example above) is mitigated by having a good value assignment algorithm within the core ATPG engine. For example, in the above case of Example 5, when variable a is ready to get assigned, its *random* value will be determined in a fashion that satisfies the above constraint, thereby minimizing backtracking. In some cases, where the number of possible values is large (for example all even numbers in a large range are the only legal values), no intelligent analysis is done and simple backtracking is invoked when a wrong value is assigned (like an odd number in the above example).

Given the pessimism that is built into RACE's value propagation algorithms, one could argue that this pessimism may lead to an incorrect solution because conflicts will go undetected during a pessimistic implication process. However, this is not true because RACE's algorithms are only pessimistic when dealing with large range values. For smaller range values and in the degenerate case when all variables have a singleton range value, RACE would perform accurate implications and in the worst case detect a conflict late in the ATPG process (if one exists). This property guarantees that RACE will always produce a legal solution (if one exists).

4.4.2 A Note on Precision Semantics

In general, expression evaluation can give different results depending on what precision semantics are followed. For example, consider a constraint expression $a == (4'd8 << 1)$, where a is a 32 bit unsigned variable. There are 2 possible precision semantics for this expression evaluation.

1. Left shift a 4-bit unsigned number 8 by 1 bit using 4-bit precision and interpret the result in 32 bits before assigning to variable a . In this case, a would get the value 0.
2. Sign extend the 4-bit (unsigned) constant 8 to 32 bits and left shift the 32-bit result by 1 bit and assign the result to variable a . In this case, a would get the value 16.

RTL-Verilog semantics impose the second precision semantics for this example. RACE appropriately inserts precision operators during its model building process to ensure such precision semantics.

4.4.3 Modeling Biasing Constraints

A key requirement in constrained-random simulation is the ability to generate stimulus (that satisfy the input constraints) to the DUV that have good coverage over the solution space for those constraints. In general, this requires a-priori knowledge of the entire solution space. To allow the user better control of distributions of random

variables, most state-of-the-art HVL support some form of biasing constraints. The underlying constraints solver is then expected to respect these biasing constraints. RACE supports biasing constraints through appropriate modeling. For example, consider the following biasing constraint for a 4-bit unsigned variable a .

```
a dist {5:7 :/ 60, 9 :/40}
```

This constraint imposes the solver to generate a value for variable a in the range $\{5:7\}$ with 0.6 probability and a value 9 with 0.4 probability. One modeling technique that can be used to model this constraint is as follows. Introduce a new random variable, s , with the following constraints.

```
0 <= s < 100;
(0 <= s <= 59) => a in {5:7};
(60 <= s <= 99) => a in {9:9};
solve s before a;
```

Note the use of an explicit variable ordering constraint to appropriately pick a value for variable s (before variable a) with uniform distribution from the range $\{0:99\}$.

4.5 The RACE Algorithm

After creating a netlist model for the constraint problem, as described above, RACE attempts to solve the problem using the following basic approach.

1. Statically imply initial range values
 - a. If conflict, return FAILURE
2. Set state variable values
3. Imply state variable values
 - a. If conflict, return FAILURE
4. Pick decision variable and random value for variable
5. Imply decision variable value
 - a. If conflict, backtrack by picking different random value
 - b. If conflict and all values of variable exhausted, backtrack to previous variable.
 - c. If all values for all decision variables exhausted, return FAILURE
6. Repeat from Step 4 as long as there are unassigned random variables.
7. If all random variables are assigned, return SUCCESS

Although this is a simplistic view of the algorithm, RACE implements several heuristics to guide the search process. The next sub-sections describe some of these heuristics. The algorithm also implements several bailout criteria that are functions of the constraint expressions themselves.

4.5.1 Word-Level Recursive and Adaptive Learning

Recursive learning has been successfully used with gate-level netlists for several EDA problems [12]. Recursive learning essentially performs case analysis on Boolean

gates to learn implications that would not have been learnt otherwise. RACE extends the notion of recursive learning (for a pre-determined recursion depth) to word-level implication learning. This concept is illustrated in the following example.

Example 6: Consider the following constraint expression.

```
rand bit[31 : 0] a;
constraint c1 {
  a == 10 || a == 15 || a == 20;
}
```

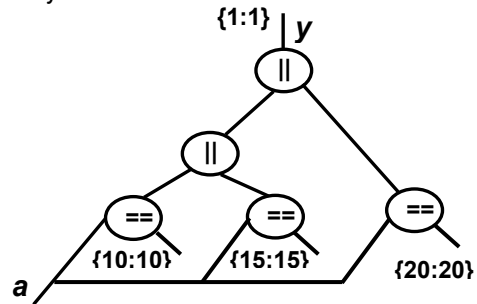


Figure 4 Example Netlist to Illustrate Recursive Learning

Without any analysis, the variable a can take 2^{32} values. The corresponding RACE netlist model is shown in Figure 4. RACE performs recursive learning on all unjustified Boolean nodes. By analyzing the OR node y , and the two possibilities for justifying it, RACE will conclude that node a must take value 10, 15 or 20, independent of how node y gets justified. The implication results on a variable node for each case analysis are accumulated and the union represents the legal possible values for that variable. □

4.5.2 Picking Decision Variables

Similar to the PODEM [8] gate-level ATPG algorithm, RACE picks only primary input random variables as decision variables. RACE also supports user-defined variable priorities; nevertheless there could be several unassigned variables at the same priority. RACE uses a dynamic cost function mechanism to decide which variable to pick first for assignment. Some factors that are considered to compute the cost function include the number of constraints and types of operators the variable participates in and the range value of the variable.

There is one distinct difference between RACE's decision process and a gate-level PODEM-like ATPG's decision process. Unlike gate-level ATPG where a justification or propagation requirement is backtraced to a primary input decision (possibly using controllability/observability cost functions) [1], RACE does not use any backtracing mechanism. (Note that backtracing and backtracking are independent concepts.) It simply picks a variable to assign (based on a cost function mentioned above) from the list of all unassigned random variables at the current priority level. This is because, for RACE to satisfy the constraints,

there is no need to generate the most intelligent vector, and all random variables have to get assigned with random values.

4.5.3 Picking Values – Iterative Relaxation

Once a variable has been picked as a decision variable, the next problem is to pick a value for this variable. From the range value of a decision node, an interval is randomly selected. If the cardinality of the interval is large, RACE uses a relaxation procedure to reduce the interval by splitting it into a pre-determined number of smaller intervals. Then, iteratively a smaller interval is picked randomly. Eventually, when the interval is small enough, a random value is picked. This main benefit of this technique is to eliminate invalid intervals from some ranges (that are discovered through implications), thereby making the random values picked from the remaining intervals more likely to be valid. In essence, it helps recover from bad decisions early on in the search process.

4.5.4 Backtracking

During ATPG, if a conflict is encountered, RACE backtracks on the last decision variable for which all values have not yet been exhausted. As part of the backtracking mechanism, the conflicting decision value is excluded before picking a new random value for the backtracked variable. This is illustrated in the following example.

Example 7: Consider again the constraints in Example 3 and the netlist model in Figure 2. It can be verified that static implications reduce the range values of variables a and b to $\{6:10\}$ and $\{0:4\}$ respectively. Assume, we pick a value for variable $a = \{7:7\}$. This value assignment backward propagates through the adder to result in $b = \{3:3\}$. This in turn implies that the output of the subtractor is $\{4:4\}$. This represents a conflict because the output of the subtractor is $\{6:6\}$. Consequently, we backtrack on variable a , by excluding the value $\{7:7\}$. Thus $a = \{6:6\}, \{8:10\}$. This implies $b = \{0:2\}, \{4:4\}$. Now ATPG continues its process of picking another random value for variable a . Eventually, ATPG will conclude that $a = 8$ and $b = 2$ is the only solution to this problem. \square

4.5.5 Random Behavior

Unlike BDD-based solutions, RACE does not have knowledge of the entire solution space. So, in order to produce random solutions with good distributions, RACE employs ample randomness within its process, while maintaining the formalism to guide the search process. Some of the contributors to random behavior include

- random variable ordering for assignment (subject to user-defined variable priorities),
- random interval selection for a variable,
- random splitting of large intervals, and
- random selection of values from intervals.

RACE ensures that it is random stable, in that repeating the process will produce the same random results. This is especially important for test-benches that are run in regression mode.

5. Experimental Results

RACE was implemented using the “C” programming language and is integrated as part of a commercial TBA tool [14]. RACE has been successfully used with several real-life test-benches. In this section we compare simulation results using RACE and a state-of-the-art BDD-based constraints solver implementation, hereinafter referred to as the *S Solver*, similar to the SimGen constraints solver [15].

1. Characteristics of Real Test Cases

Test Case	# Constraints (C)	# Random Variables (RV)	# State Variables (SV)	Max Bits
Test1	20	12	0	10
Test2	14	20	1	28
Test3	14	20	1	28
Test4	4	4	8	32
Test5	21	19	0	16
Test6	26	24	3	5
Test7	12	12	0	32
Test8	357	119	0	17
Test9	16	16	6	28
Test10	724	197	0	32
Test11	17	6	13	32
Test12	215	83	178	32
Test13	23	9	120	32
Test14	393	35	43	144
Test15	1	11	0	32

Table 1 shows characteristics of some real test cases that were processed using RACE and S Solver, either as a standalone test-bench or in conjunction with a commercial RTL-Verilog simulator [13]. Here “Max Bits” refers to the maximum bit-width of variables participating in the corresponding constraints. These test cases are fairly complex test-benches and are used for functional verification of state-of-the-art complex designs. For example, Test10 is a test-bench to verify designs for cellular video streaming, and Test25 in Table 3 is an ethernet verification IP.

Table 2 summarizes the simulation results for these test cases. The simulations were run on a 900 MHz SUNW, UltraSPARC-III+ machine with 4 Gb of RAM. All CPU numbers are for the entire simulation run. For *S Solver*, the “Total” time includes the time to compile the BDDs, and the “Evaluation” column refers to the total simulation time, assuming the BDDs are already compiled. The product infrastructure on which both solvers are implemented includes a custom memory manager. Consequently, it becomes hard to measure the exact absolute memory

consumed by either solver. The “Memory” column shows the incremental increase in memory consumed by the simulation process when either solver is used (as would be observed by an user).

2. Simulation Results for Real Test Cases

Test Case	Simulation with RACE		Simulation with S Solver		
	CPU (secs)	Memory (Mb)	CPU (secs)		Memory (Mb)
			Total	Evaluation	
Test1	16	0.23	15	0.2	26.6
Test2	99	0.09	40	14	42.3
Test3	121	0.08	41	17	42.3
Test4	122	0.06	75	62	18.4
Test5	116	0.07	79	51	22.7
Test6	110	0.07	98	80	36.8
Test7	113	0.05	103	93	22.2
Test8	90	0.62	119	22	203.7
Test9	133	0.05	135	113	54.5
Test10	86	2.84	144	74	132.4
Test11	72	0.17	260	231	123.0
Test12	99	1.45	615	10	272.8
Test13	66	0.73	751	145	366.9
Test14	60	3.95	754	2	284.2
Test15	124	0.04	2873	29	323.1

All these test cases run for several simulation cycles and invoke the core solver engines several times. For test cases Test1 through Test9, where the BDD compilation and evaluation time for *S Solver* is less than or comparable to the RACE evaluation time, *S Solver* would likely be the preferred solution. In such cases, the one-time price for building the BDDs would get amortized as a very low cost over long simulation runs and *S Solver* would beat any runtime solver like RACE.

However, for test cases Test10 through Test15, *S Solver* spends a lot of time in BDD compilation and/or evaluation. RACE would be the preferred solution in these cases. In all test cases, the incremental memory consumption by RACE is very small compared to that of *S Solver*.

3. Results for Real Test Cases Solved Only By RACE

Test Case	# C	# RV	# SV	Max Bits	Simulation with RACE	
					CPU	Mem
Test16	17	14	13	32	89	0.34
Test17	21	16	5	32	47	0.17
Test18	9	9	7	32	140	0.03
Test19	546	215	6	32	83	0.48
Test20	549	215	6	32	117	0.50
Test21	400	129	0	32	52	1.35
Test22	412	129	0	32	24	1.39
Test23	630	193	0	32	71	2.20
Test24	752	176	0	32	32	2.74
Test25	138	34	68	48	220	0.55
Test26	191	16	27	32	64	0.99
Test27	1	1	15	32	113	0.05

During our benchmark evaluations with customers, we also encountered several real test-benches, whose constraints could be solved *only* by RACE. The results for these test

cases are summarized in Table 3. For these test cases, *S Solver* could not compile the BDDs after consuming significant compute resources. Factors contributing to this include large bit-widths on certain variables, complex intertwined constraints, use of certain arithmetic operators, and poor static BDD variable ordering. In some of these evaluations, RACE also outperformed some other commercial constraints solvers in the industry.

Next we demonstrate the scalability of the RACE solver by formulating a constraint problem for the well-known N-Queen problem. The N-Queen problem is to find legal positions for N Queens on an $N \times N$ chessboard, such that no 2 queens can kill each other. This problem can be formulated using 3 sets of linear constraints for elements in an $N \times N$ matrix whose values can be either 0 or 1.

- The sum of each row should be exactly one.
- The sum of each column should be exactly one.
- The sum of each diagonal should be at most one.

4. RACE results for N-Queen Problem

Test Case	Simulation with RACE	
	CPU (secs.)	Memory (Mb)
4-Queen	0.2	0.09
8-Queen	0.6	0.30
15-Queen	2	0.98
30-Queen	10	3.77
50-Queen	26	9.99
100-Queen	169	39.30

The number of constraints for the N-Queen problem is $6N - 2$ and the number of random variables is N^2 . Table 4 summarizes the results of running RACE to generate 10 random solutions for each queen problem. *S Solver* could not handle any more than the 10-queen problem.

6. Limitations and Future Directions

RACE, like most runtime solution techniques has its own limitations. For constraint problems where the search-space is large and the solution-space is small, one could imagine RACE getting into trouble. This situation can occur when a test-bench is architected mostly with directed tests, with minimal random stimulus generation under very tight constraints. Another limitation of RACE is that it attempts to provide good random distributions, without the knowledge of the entire solution-space. The sub-optimality of the RACE netlist model may also limit the number of implications learnt, thereby affecting its performance.

RACE, being used in the simulation context, needs to be highly performance-sensitive, especially considering that it is a runtime solution technique that has the potential to slow down the simulation when dealing with tight constraints. Although this problem is somewhat mitigated, by caching enough search-space exploration and learning information across randomization requests, continuous

improvements are required through new and improved heuristics.

Another area that needs attention is sequential constraints solving. This problem will become more relevant as designers start adopting the emerging SystemVerilog standard, where assertions and constraints could be used interchangeably to efficiently support an assume-guarantee type of SOC verification methodology.

7. Conclusions

Functional verification of complex SOCs presents numerous challenges. Complex designs have complex operating environments, thereby making functional test generation a big bottleneck. A key goal of functional test generation is to expose corner-case design bugs. Constrained-random simulation and ability to solve constraints efficiently plays a key role in functional verification to accomplish these goals.

This paper has presented RACE, a new word-level constraints solving system. RACE provides several advantages, most notably its ability to analyze and solve the constraints problem at a much higher level of abstraction. Experimental results on real test-benches demonstrate the effectiveness of RACE in its simulation performance, as well as in its ability to solve complex constraints. As reported by our customers, RACE has shown to be a highly predictable, robust and scalable solver, that produces consistent results across a wide variety of constraint problems.

Experience shows that constraints in state-of-the-art RTL test-benches are fairly complex and any single solution technique is probably not going to suffice. With many emerging technologies in this area, a natural approach would be to combine the complementary strengths of different constraint solvers with appropriate orchestration.

8. Acknowledgements

The author thanks William Nicholls and Carl Pixley for providing the *S Solver* used in the experiments. Special thanks to Scott Davidson for reviewing an earlier version of this paper and providing valuable feedback. Thanks also to Bharat Kalyanpur, Sushrut Karanjkar, Mukul Khandelia, Stephen Meier, Amir Mottaiez, Carsten Mueller, Vikram Saxena, and Shashidhar Thakur for useful discussions and/or for helping integrate RACE into a TBA tool.

9. References

[1] M. Abramovici, M. Breuer, and A. Friedman, "Digital Systems Testing and Testable Design", *IEEE Press*, New York, 1995.

- [2] A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, and V. Schwartzburd, "Verification of the IBM RISC System/6000 by a Dynamic Biased Pseudo-Random Test Program Generator", *IBM Systems Journal*, Vol. 30, No. 4, 1991, pp. 527-538.
- [3] J. Bergeron, "Writing Testbenches: Functional Verification of HDL Models", *Kluwer Academic Publishers*, Boston, MA, 2000.
- [4] E. Bin, R. Emek, G. Shurek, and A. Ziv, "Using a Constraint Satisfaction Formulation and Solution Techniques for Random Test Program Generation", *IBM Systems Journal*, Vol. 41, No. 3, 2002, pp. 386-400.
- [5] R. Bryant, "Graph-based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computers*, C-35, August 1986, pp. 677-691.
- [6] A. K. Chandra and V. S. Iyengar, "Constraint Solving for Test Case Generation", *Proceedings of International Conference on Computer Design*, 1992, pp. 245-248.
- [7] A. K. Chandra, V. S. Iyengar, D. Jameson, R. Jawalekar, I. Nair, B. Rosen, M. Mullen, J. Yoon, R. Armoni, D. Geist, and Y. Wolfsthal, "AVPGEN – A Test Case Generator for Architecture Verification", *IEEE Transactions on VLSI Systems*, Vol. 3, No. 2, June 1995, pp. 188-200.
- [8] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits", *IEEE Transactions on Computers*, Vol. C-30, No. 3, March 1981, pp. 215-222.
- [9] C.-Y. Huang and K.-T. Cheng, "Assertion Checking by Combined Word-level ATPG and Modular Arithmetic Constraint-Solving Techniques", *Proceedings of the Design Automation Conference*, June 2000.
- [10] IEEE Std. P1364-2001, "IEEE Standard Hardware Description Language Based on Verilog Hardware Description Language", *The IEEE, Inc.*, New York, March 2001.
- [11] M. A. Iyer, "High Time for High-Level ATPG", *Panel position statement, Proceedings of the International Test Conference*, 1999, pp. 1112.
- [12] W. Kunz, and D. Pradhan, "Recursive Learning: A New Implication Technique for Efficient Solutions to CAD-problems: Test, Verification and Optimization", *IEEE Transactions on Computer-Aided Design*, Vol. 13, No. 9, September 1994, pp. 1143-1158.
- [13] VCS 7.0 Reference Manual, *Synopsys, Inc.*, 2003.
- [14] Vera 6.0 Reference Manual, *Synopsys, Inc.*, 2003.
- [15] I. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz, "Modeling Design Constraints and Biasing in Simulation Using BDDs", *Proceedings of the International Conference on Computer-Aided Design*, November 1999, pp. 584-589.