Test Generation Games from Formal Specifications

Ansuman Banerjee, Bhaskar Pal, Sayantan Das
Abhijeet Kumar and Pallab Dasgupta

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
West Bengal, INDIA-721302

ABSTRACT

In this paper, we present methods for automatic test generation from formal specifications. These are used to create intelligent test benches that are able to cover corner case behaviors in much less time. We have developed a prototype tool for intelligent test generation within the layered test bench architecture proposed in RVM. We present results on verification IPs of standard bus protocols to show the effectiveness of our approach.

Categories and Subject Descriptors: B.5.2 [Hardware]: Register-Transfer-Level Implementation – *Verification*

General Terms: Verification

Keywords: Test Generation, Vacuity, Realizability

1. INTRODUCTION

In recent times, coverage driven dynamic Assertion-Based Verification (ABV) is assuming significance in the design validation flow of chip design companies. Active participation from the design and EDA industries have led to several formal languages for property specification, of which System Verilog Assertions (SVA) [13] is one of the most popular ones.

The success of dynamic ABV depends largely on the simulation coverage of the scenarios that are relevant to the properties that constitute the property suite. The mere existence of a property that guards against a given bug is not enough to detect the bug unless the simulation of the design-under-test (DUT) covers all input scenarios that may trigger the faulty behavior.

Designs often have correctness requirements under corner case scenarios that occur rarely, but are of considerable significance from the point of view of functional correctness. Experience shows that random test generators often fail to create these corner case scenarios in reasonable time because the probability of such scenarios is low. As a result, the validation engineer manually writes directed tests to create corner case scenarios to cover the behaviors targeted by the complex temporal properties. It is not easy to determine the complex input sequences that trigger a property by visual examination of the property. Therefore one of the main challenges in the current ABV framework is to develop formal methods for automatic test generation from temporal properties. This is the main focus of our work.

Automatic test generation may be viewed as a game between the DUT module and the test bench. The module attempts to satisfy a given property by setting appropriate values to its outputs, where as the test bench attempts to refute it by controlling the input signals. The module and the test bench alternate over time. The module has a bug if the test bench has a strategy to refute the property. An intelligent test bench must be online and adaptive, and should have the following two features:

- The test bench should avoid generation of input vectors that satisfy the specification *vacuously*.
- If at any point, the module produces an output that may lead to failure under some specific input scenario, then the test bench should drive that scenario to lead the module to failure, and expose the bug. This may happen over multiple cycles.

In order to adequately address the above issues, the test generation algorithm must be aware of the notions of realizability [9] and vacuity of open system specifications. In this paper, we address both these issues and propose an integrated intelligent test bench approach that can answer the above requirements.

The main contributions of this paper are as follows.

1. We show that the use of vacuity and realizability leads to intelligent test generation from formal properties in a game between the module and its test bench. We develop formal methods for generating non-vacuous tests.
2. For unreceptive [4] specifications, we create intelligent test benches that are capable of driving the simulation to a property refutation, thereby exposing a bug. This is important because system level specifications are often unreceptive, since cycle to cycle execution details are often unspecified.
3. We present a complete verification methodology on top of RVM [10] that implements our test generation algorithms and show results of higher simulation coverage in lesser time as compared to coverage driven randomized test benches on several verification IPs for standard Bus protocols.

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

2. RELATED WORK

There has been considerable research on automatic test generation from formal specifications. In one approach, a model checker is used as an oracle to compute the expected outputs and the counterexamples it generates are used as test sequences [2, 5]. This approach is offline and it suffers from capacity bottlenecks for large designs, whereas we have no such limitations. Another approach of automatically generating tests has been based on the specification of input constraints as a separate specification [3]. This requires the user to provide the input biases. However, our methodology is much more generic as we generate the input constraints from the property specification itself. In [6], a methodology is presented for automatically determining the appropriate vector constraints, based on the analysis of both the design and the property being checked. *ATPG based* techniques [1] have also been quite extensively used for automatic test generation. ATPG is offline and it compares the design implementation and the given properties to create the tests prior to simulation, where as we create them solely on the basis of the formal properties, and handle the comparison online during simulation. As a result, ATPG based approaches are limited by the capacity and efficiency of the ATPG engines, whereas we have no such limitations. Several other approaches [12, 14] have also addressed the problem of automatic test generation from specifications. Our work has two major differences with those above:

1. We address the issue of vacuity, which has not been addressed in previous test generation methodologies.
2. For unreceptive [4] specifications, we create test benches that can drive the simulation to a property refutation.

3. NON-VACUOUS TEST GENERATION

We first explain the issue of vacuity in test generation for verification of modules. Non-vacuous test generation is an advantage, since tests which satisfy properties vacuously lead to wastage of simulation cycles. Consider Example 1.

EXAMPLE 1. Consider an arbiter A, having two input request lines, r_1 and r_2, and three grant lines, g_1, g_2 and g_d. The functionality of the arbiter is described as below:

1. Whenever r_1 goes high, g_1 is asserted in the next cycle.
2. Whenever r_2 goes high, g_2 is asserted within 3 cycles.
3. g_d (default grant) is asserted at least once in every 3 cycles.
4. The grant lines are mutually exclusive.
5. The request lines r_1 and r_2 are mutually exclusive.
6. The request r_1 never arrives in successive cycles.
7. When r_2 arrives, it remains high until g_2 is asserted.

The last 3 properties are assumptions on the arbiter inputs. The properties expressed in Linear Temporal Logic (LTL) [8] are (X denotes the *next time* operator and G is the *always* operator):

1. P_1: $G(r_1 \Rightarrow Xg_1)$
2. P_2: $G(r_2 \Rightarrow Xg_2 \vee XXg_2 \vee XXXg_2)$
3. P_3: $G(g_d \vee Xg_d \vee XXg_d)$
4. P_4: $G(mutex(g_1, g_2, g_d))$
5. A_1: $G(mutex(r_1, r_2))$
6. A_2: $G(r_1 \Rightarrow X\neg r_1)$
7. A_3: $G(r_2 \Rightarrow (r_2 \, U \, g_2))$

The objective of ABV is to check whether the arbiter satisfies the properties $P_1, \ldots, P_4$ in those simulation runs that satisfy the *assume constraints*, $A_1, \ldots, A_3$. Simulations refuting one or more of these assume constraints are invalid, not because the arbiter has a bug, but because the test bench drives invalid input sequences. The first responsibility of a test bench is to drive input sequences that satisfy all assumptions.

Even when an input sequence satisfies all assume constraints, it may not *trigger* a given property. Consider an input sequence where r_2 is never asserted. This sequence satisfies all assumptions, but does not trigger P_2. In other words, to create the meaningful scenarios for which P_2 was written, the test bench must drive input sequences where r_2 is asserted. We treat non-vacuous interpretations of properties as coverage points. The challenge is in creating random tests which cover these coverage points. □

As demonstrated in the above example, the test generator must always generate tests that are non-vacuous with respect to the set of properties. The requirement of non-vacuity is not limited to a particular cycle, and may span multiple cycles as well, over different input vectors. Consider the following example specification for a IBM-CoreConnect compliant PLB Master device:

EXAMPLE 2. The PLB Master should assert ReadBurst signal for the secondary acknowledged burst read transfer in the following cycle after receiving ReadBurstTerm from the Slave device for the on-going primary burst read transfer. The LTL property can be written as:

$$\psi\text{: } G(((ReadBurst \wedge PAV \wedge S\_Ack) \wedge X(ReadBurst \wedge SAV \wedge S\_Ack) \wedge XX(ReadBurstTerm)) \Rightarrow XXX(ReadBurst))$$

To test the above property for a PLB master device in isolation, the signals PAV, SAV, S_Ack, and ReadBurstTerm (which are inputs to the PLB master) should be driven from the test bench. A purely random test bench in reasonable time would fail to create such a complicated pre-condition spanning over 3 cycles (in the first, $PAV \wedge S\_Ack$, in the next, $SAV \wedge S\_Ack$, followed by $ReadBurstTerm$ should be driven). However, to test this property, a test bench should create this corner case scenario. □

Formally, we define a module $\mathcal{J}$ as a RTL design block having a set of inputs $\mathcal{I}$, a set of outputs $\mathcal{O}$, an initial block $Init$, and a RTL description $\mathcal{B}$. The execution of the initial block produces the values of the output variables at the beginning of the simulation. Before defining the concept of non-vacuous test generation, we define the concepts of a *X-pushed formula* and a *X-guarded formula*.

DEFINITION 1. **[X-pushed formula:]**
A formula is said to be X-pushed if all the X operators in the formula are pushed as far as possible to the left.

DEFINITION 2. **[X-guarded formula:]**
A formula is said to be X-guarded if the corresponding X-pushed formula starts with an X operator whose scope covers the whole formula.

EXAMPLE 3. Let us consider the temporal property

$$\mathcal{P} = ((X \; p) \; U \; (X \; X \; q)) \wedge (X \; F \; r)$$

The X-pushed form of $\mathcal{P}$ is:

$$\mathcal{P}_X = X \; ((p \; U \; (X \; q)) \wedge (F \; r))$$

We can say that $\mathcal{P}$ is a X-guarded formula since the corresponding X-pushed formula $\mathcal{P}_X$ starts with a X operator whose scope covers the whole formula. □

The task of monitoring the truth of a given LTL property along a simulation run works as follows. If we are required to check a LTL property, φ, from a given time step, t, we rewrite the LTL property into a set of propositions over the signal values at time t and a set of X-guarded LTL properties over the run starting from time $t+1$. The rewriting rules are standard, and are as follows:

$F\varphi \quad = \quad \varphi \vee XF\varphi$
$G\varphi \quad = \quad \varphi \wedge XG\varphi$
$p\ U\ q \quad = \quad q \vee (p \wedge X(p\ U\ q))$

The property checker reads the signal values from the simulation at time t and substitutes these on the rewritten properties and derives a new property that must hold on the run starting from $t+1$, by dropping the leftmost X operator from each X-guarded term.

For example, to check the property $p\ U\ (q\ U\ r)$ at time t, we rewrite it as $(r \vee (q \wedge X(q\ U\ r))) \vee (p \wedge X(p\ U\ (q\ U\ r)))$. If the simulation at time t gives $p=0$, $q=1$, $r=0$, then by substituting these values, we obtain the property $X(q\ U\ r)$. Therefore at time $t+1$ we need to check the property $q\ U\ r$. We repeat the same methodology on $q\ U\ r$ at time $t+1$.

For automatic test generation, we may choose the values of the input signals at each time step t while monitoring the property. We first define a *vacuous input vector*.

Definition 3. **[Vacuous input vector]** *An input vector, $\widehat{I}$, is vacuous at a given state with respect to a property, φ, iff φ becomes true at that state on input $\widehat{I}$ regardless of the values of the remaining variables.* □

For each property, our target is to drive a sequence of tests such that a non-vacuous success/failure of the property is reported. Therefore, we define our coverage points as non-vacuous interpretations of *individual* properties and create tests that target each coverage point in turn. Consider the following example.

Example 4. Consider the specification $\mathcal{Q}$ for a priority arbiter having inputs r_1, r_2 and outputs g_1, g_2. $\mathcal{Q}$ contains the following properties:

1. Whenever r_1 remains high for two consecutive cycles, g_1 should be asserted in the next cycle.
   Q_1: $G(r_1 \wedge Xr_1 \Rightarrow XXg_1)$
2. Whenever r_2 goes high, with r_1 remaining low, g_2 must be asserted in the next cycle.
   Q_2: $G(r_2 \wedge \neg r_1 \Rightarrow Xg_2)$

We want to test both Q_1 and Q_2 non-vacuously. These are our coverage points. For Q_1, the test bench must drive $r_1=1$ and choose r_2 randomly for two consecutive cycles. For Q_2, it must drive $r_2=1$ and $r_1=0$. Note that in the first case, Q_2 is satisfied vacuously, while in the second case Q_1 is satisfied vacuously. □

3.1 Test Generation Algorithm

We first present the algorithm for test generation from LTL properties as a standalone procedure. We have used LTL for ease of presentation. However, the algorithms are generic and can be used with other property specification languages as well. We later integrate the algorithms into the verification framework of RVM [10].

To develop the formal algorithm for test generation, let us study the game between the module J and its test bench with respect to a given property, $\mathcal{L}$. The execution of the *initial*-block of the module is the first move of the module. If the initial state is sufficient to satisfy or refute $\mathcal{L}$, then we have a hit. Otherwise, there must exist some non-vacuous input vectors with respect to $\mathcal{L}$. The test generator must choose one such assignment. We now simulate the module with that input and study the response of the module (that is, the values of the outputs) in the next cycle. If $\mathcal{L}$ is now satisfied or refuted, then we have a hit, otherwise we repeat the process of test generation.

Procedure SimulateMain outlines our algorithm for test generation for a module J and a property $\mathcal{L}$. It calls the procedure GenStimulus to produce non-vacuous input vectors with respect to $\mathcal{L}$. Note that $\mathcal{L}$ can either be a single property or a conjunction of one or more properties.

Algorithm 1. **Procedure SimulateMain**

SimulateMain(module: J, property: $\mathcal{L}$)
Step 1: Set $\widehat{O}$ = the output vector obtained after execution of the initial block *Init* of J
Step 2: While (not end of simulation) begin
  2.1: Rewrite $\mathcal{L}$ in terms of present state Boolean propositions and X-guarded temporal properties
  2.2: Substitute the values of the outputs from $\widehat{O}$ in the non X-guarded terms of $\mathcal{L}$ to obtain $\widehat{L}$
  2.3: If $\widehat{L}$ = TRUE, return success
  2.4: If $\widehat{L}$ = FALSE, return failure
  2.5: $\widehat{I}$ = GenStimulus($\widehat{L}$)
  2.6: Obtain $\mathcal{L}'$ from $\widehat{L}$ by substituting $\widehat{I}$ in the non X-guarded terms of $\widehat{L}$ and dropping the leftmost X from each X-guarded temporal property
  2.7: Simulate J with $\widehat{I}$
  2.8: Set $\widehat{O}$ = the output vector after simulation
  2.9: Set $\mathcal{L} = \mathcal{L}'$
EndAlgorithm

Algorithm 2. **Procedure GenStimulus**

Input_Vector GenStimulus(property: $\mathcal{L}$)
// $\mathcal{L}$ is a property over $\mathcal{I}$ and X-guarded terms over $\mathcal{I} \bigcup \mathcal{O}$
Step 1: Rewrite $\mathcal{L}$ as a conjunction of clauses, where each clause is a disjunction of Boolean formulas and X-guarded terms
Step 2: Set $\mathcal{P}$ = the Boolean formula obtained from $\mathcal{L}$ after dropping the X-guarded terms
Step 3: If $\mathcal{P}$ is satisfiable, $\widehat{I}$ = a random input vector that refutes $\mathcal{P}$ else $\widehat{I}$ = any random input vector
Step 4: return $\widehat{I}$
EndAlgorithm

The working of SimulateMain is explained below:

Example 5. Consider the arbiter specification in Example 1. The coverage points include P_1 and P_2. Initially, g_d is high, while g_1 and g_2 are low. Substituting the values of g_d, g_1 and g_2, the specification does not evaluate to true or false. Let us assume P_1 is taken as the first coverage point. GenStimulus is called with P_1, re-written as:

$$((\neg r_1 \vee Xg_1) \wedge XG(r_1 \Rightarrow Xg_1))$$

as the argument, which returns a non-vacuous input vector, say $r_1=1$, $r_2=0$. The arbiter is simulated with this input vector. In response, the arbiter asserts g_1, and control goes back to Step 2 of SimulateMain, and a match of P_1 is found. P_2 is taken as the next coverage point. Substituting the values of g_1, g_2 and g_d obtained in the previous cycle, the specification does not evaluate to true or false. GenStimulus is called with

$$(\neg r_2 \vee Xg_2 \vee XXg_2 \vee XXXg_2) \wedge XP_2$$

as argument, and a non-vacuous input vector $r_2=1$, $r_1=0$ (due to A_1) is returned. The module is simulated with this input

vector but g_2 remains de-asserted. Control goes to Step 2 again and the value of g_2 is substituted. Since truth of P_2 cannot be concluded, the process is continued and r_2 is driven high for the next 2 cycles, but the arbiter fails to assert g_2. Hence P_2 fails. □

THEOREM 1. *GenStimulus never returns a vacuous input vector with respect to its argument $\mathcal{L}$.*

Proof: Since $\mathcal{L}$ can contain only Boolean propositions over the input variables and X-guarded temporal properties, the operation performed in Step 2 of *GenStimulus* can lead to one of the following two outcomes depending on $\mathcal{P}$:

1. $\mathcal{P}$ does not contain any input variable. This means that no assignment of input variables is sufficient to satisfy $\mathcal{L}$. In this case, any random vector $\widehat{I}$ over the present state input variables is non-vacuous with respect to $\mathcal{L}$.
2. $\mathcal{P}$ is satisfiable and contains a Boolean formula containing only input variables from $\mathcal{I}$. Thus any satisfying assignment for $\mathcal{P}$ can make $\mathcal{L}$ true regardless of the values of the remaining variables. In this case, *GenStimulus* returns an input $\widehat{I}$ vector which does not satisfy $\mathcal{P}$, and thereby does not satisfy $\mathcal{L}$ vacuously.

Thus, GenStimulus never returns a vacuous input vector with respect to a specification $\mathcal{L}$. □

THEOREM 2. *For an LTL specification $\mathcal{L}$ and a module J, if SimulateMain returns success (or failure), J satisfies (or refutes) $\mathcal{L}$ non-vacuously.*

Proof: At the start of every simulation step, $\mathcal{L}$ is rewritten in terms of present state Boolean propositions and X-guarded temporal properties. The values of the output variables obtained in the previous simulation (or the initial values) are substituted. Now it might be the case that $\mathcal{L}$ evaluates to true (or false) after this substitution, in which case, we report success (or failure). Otherwise, GenStimulus is called and a non-vacuous input vector is returned. This is used for the next simulation step and the output values are obtained. It follows from Theorem 1 that GenStimulus always returns a non-vacuous input vector with respect to $\mathcal{L}$. Thus, when procedure SimulateMain returns success (or failure), J satisfies (or refutes) $\mathcal{L}$ non-vacuously. □

4. REALIZABILITY GAMES

A common problem in dynamic ABV with temporal specifications is that the property refutation may happen several cycles after the actual fault. This is typically the case for unreceptive specifications [4]. Moreover, the refutation may depend on the inputs that are driven after the fault. Consider the following example:

EXAMPLE 6. Consider the scenario shown in Fig 1 for our running example. At time t, the grant was parked on g_d and r_1 was asserted. At $t+1$, the arbiter asserted g_1 (satisfying P_1), and the test bench raised r_2. At $t+2$, the arbiter asserted g_2, and the test bench raised r_1. At this point, a failure is imminent at $t+3$ because P_1 requires g_1 to be raised, P_3 requires g_d to be raised, and P_4 disallows both g_1 and g_d to be raised.
Here, the fault occurred at $t+2$ when the arbiter asserted g_2. The correct behavior would have been to assert g_d at $t+2$ and then assert g_2 at $t+3$. It is interesting to note that the fault would have passed undetected had the test bench not raised r_1 at $t+2$.

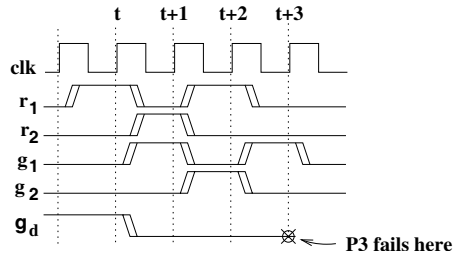

Figure 1: A Conflicting Scenario

Can the test bench intelligently detect such cases and drive the appropriate inputs to reach a refutation in some future state? Indeed it can by using the notion of *realizability* of specifications.

It is also interesting to note that though the fault occurred at $t+2$, the specification was not refuted at $t+2$. This is a typical scenario in *non-receptive* [4] specifications. Non-receptive specifications occur because identifying a property for every individual scenario is difficult in practice. For example, to detect this fault at $t+2$, we may add the property

$$G((r_1 \ \wedge \ Xr_2) \ \Rightarrow \ XXg_d)$$

This is non-intuitive, unless the above scenario is demonstrated. The property is also redundant formally, since it is possible to detect the fault at $t+2$ using a realizability checker. □

In order to adequately address the above issue, the test generation algorithm must be aware of the notions of realizability [9] of system specifications. Realizability is an important notion in the proposed test generation games. A specification may be realizable at the beginning of simulation, but when a fault occurs (that is, the module makes a wrong move), the specification may become unrealizable but not unsatisfiable. In other words, the property has not yet been refuted, but it is possible to drive the module to a state where the property is guaranteed to be refuted. This is a very important requirement, since our end goal is to expose bugs. In Example 6, the specification was realizable at time t. At $t+2$, when the arbiter asserted g_2, the effective specification contains the property

$$Xg_d \wedge (r_1 \Rightarrow Xg_1) \wedge mutex(g_1, g_2, g_d)$$

which is satisfiable (for $r_1 = 0$), but unrealizable, because the test bench has a winning strategy (for $r_1 = 1$).

Given an unrealizable specification, the goal of the test generator is to drive the module to a state where the specification is refuted. This may happen in a sequence of steps (with alternate moves between the module and the test generator). In each step, the test generator needs to make sure that the input generated by it does not make the specification realizable. This game between the module and its test bench is called a *realizability* game. When the module produces a fault that renders the specification unrealizable (but not unsatisfiable), then we start a *realizability game* to drive the module to a refuting state.

It may be noted that we could have simply reported a failure when the specification becomes unrealizable. The difficulty is in presentation of the bug. It is hard to demonstrate succinctly why a specification is unrealizable. On the other hand, if we play the realizability game then we have a real counter-example after reaching the refutation.

In order to adequately address the above issue, we propose to augment the test generation algorithm with a realizability

check at the start of every simulation step. Algorithms for realizability checking have been proposed in literature. In our flow, we have used the QBF based approach for checking realizability of LTL specifications presented in [11].

An obvious question which comes up is as follows : *Is it practical to invoke a realizability checker in every simulation step? Will it degrade simulation performance?* A realizability check is expensive, particularly when the specification is large. It is therefore not practical to invoke a realizability check on the entire specification at every step of the simulation. However in practice, when a fault occurs, typically a small subset of the properties become unrealizable, and as a result, the whole specification becomes unrealizable. A relatively experienced practitioner of verification can identify the unreceptive parts of the specification and the properties that are related to unreceptiveness. It is practical to use a realizability checker on such small collections of properties. We believe our approach will work well in practice since advances in SAT solvers have made a significant impact on performance of QBF solvers, used for realizability checking.

Thus, to detect failures as soon as they are imminent from the module responses, we have to verify after Step 2.4 of SimulateMain if the property $\widehat{L}$ is unrealizable. If so, we should drive the module to a refutation. In this case, Step 2.5 of SimulateMain is as follows (remaining steps are same):

If $\widehat{L}$ is unrealizable, $\widehat{I}$ = GenRefute($\widehat{L}$)
else $\widehat{I}$ = GenStimulus($\widehat{L}$)

We now present the procedure GenRefute.

ALGORITHM 3. **Procedure GenRefute**

Input_Vector GenRefute(property: $\mathcal{L}$)
Step 1: Set $\widehat{I}$ = a random input vector
Step 2: For each input variable $p \in \mathcal{I}$
2.1 Substitute p = 0 in the non X-guarded terms of $\mathcal{L}$
2.2 If $\mathcal{L}$ remains unrealizable, set the p^{th} bit of $\widehat{I}$ to 0;
else set the p^{th} bit of $\widehat{I}$ to 1;
Step 3: return $\widehat{I}$
EndAlgorithm

The working of SimulateMain is explained below:

EXAMPLE 7. As before, the coverage points include P_1 and P_2. Initially, g_d is high, while g_1 and g_2 are low. Substituting the values of g_d, g_1 and g_2, the specification remains realizable but does not evaluate to true or false. GenStimulus is first called with P_1, re-written as:

$$((\neg r_1 \ \vee \ Xg_1) \ \wedge XG(r_1 \ \Rightarrow \ Xg_1))$$

as the argument, which returns $r_1 = 1$, $r_2 = 0$. After simulation with this input, the arbiter asserts g_1, and control goes back to Step 2 of SimulateMain, and a match of P_1 is found. P_2 is the next coverage point. Substituting the values of g_1, g_2 and g_d obtained in the previous cycle, the specification remains realizable, but does not evaluate to true or false. GenStimulus is called with

$$(\neg r_2 \ \vee \ Xg_2 \ \vee \ XXg_2 \ \vee \ XXXg_2) \ \wedge \ XP_2$$

as argument, and a non-vacuous input vector $r_2 = 1$, $r_1 = 0$ is returned. When simulated with this input, the arbiter asserts g_2. Control goes to Step 2 again and the value of g_2 is substituted. At this point, the effective specification contains

$$Xg_d \wedge (r_1 \Rightarrow Xg_1) \wedge mutex(g_1, g_2, g_d)$$

which is unrealizable. GenRefute is called with this as the argument. Evidently, for $r_1 = 1$, the specification remains unrealizable, and hence $r_1 = 1$, $r_2 = 0$ is returned. When the module is simulated with this input vector, the module asserts g_1, and a refutation occurs on substitution of g_1 due to violation of P_3. □

THEOREM 3. *For an unrealizable LTL specification $\mathcal{L}$ and a module J, procedure GenRefute correctly returns an input vector $\widehat{I}$ such that $\mathcal{L}$ remains unrealizable after substitution of $\widehat{I}$ in $\mathcal{L}$. GenRefute performs $|k|$ realizability checks, where J has k inputs.*

Proof: Since $\mathcal{L}$ is unrealizable, it follows that we have a winning strategy for the test bench, and hence it can select some input vector $\widehat{I}$ for which the property remains unrealizable. $\widehat{I}$ must assume a value 0 or 1 for each input variable in $\mathcal{I}$. For each input variable $p \in \mathcal{I}$ of J, Step 2.2 of GenRefute constructs the input vector incrementally, by setting the value of p to 0 first and checking if the specification remains unrealizable. In case, the specification becomes realizable, we take the value of that input to be 1. Thus we perform $|k|$ realizability checks to construct the input at a particular cycle, when the module has k inputs. □

5. RVM INTEGRATION

We have developed a prototype tool for intelligent test generation within the layered test architecture of RVM [10]. In our tool, the specification is accepted in SVA [13]. The architecture of our tool is shown in Fig 2.

A RVM based test bench can have various layers. These include the test layer, scenario layer (generator), functional layer, command layer (Driver), and signal layer (interface). Each layer (except the signal-layer) models a transactor, each of which generates transactions at different levels of abstraction. Each layer can also have a coverage block associated with it. This constrains the respective transactor to generate stimuli so as to increase the overall coverage. In general, the generator layer does all the test scenario generation based on the current coverage requirement (constraints from the coverage block) and current state (current observed output) of the module. The monitor block samples the module outputs to keep track of the current module state. The Driver takes the generated transaction from the generator and drives it through the interface for simulation.

Functional coverage identifies which test conditions have been generated by the test bench. Instead of coding individual interesting conditions in individual directed test cases, they are coded as coverage points in a functional coverage model. Test cases are used to steer the test bench towards the uncovered points in the coverage model. We define our coverage goal as *the number of non-vacuous matches for each property.* This defines our coverage points.

The coverage block provides appropriate constraints at each cycle to the generator, which generates scenarios satisfying the constraints. The key to coverage driven constrained random test generation is in intelligently crafting the testbench, such that we can correctly identify the constraints at each cycle and generate scenarios that contribute to the overall verification goal. This is where, our algorithms can be best utilized. Figure 2 shows the complete RVM based verification framework. We explain the details below.

- The realizability checker RC is written in C and it is used through the DirectC interface.
- Data communication is performed using channels. For example, CB uses Ch_{cb2g} for communicating with G.

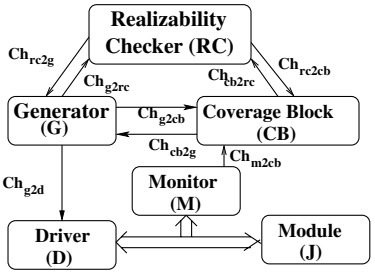

Figure 2: Integrated RVM-based Flow

- The Monitor block M samples the output values from the module at each simulation clock.
- The set of coverage points are specified in the Coverage Block CB in SVA. CB keeps track of the list of uncovered assertions. At each cycle, it samples the values from M and substitutes the values in the current property $\mathcal{L}$. If $\mathcal{L}$ becomes true (or false), it flags success (or failure) and takes the next assertion. Otherwise, it calls RC for the realizability check, and informs the generator G. It waits for an input vector from G, and on receiving that, substitutes the input values and prepares the property to be checked in the next cycle.
- The GenStimulus and GenRefute procedures are implemented in G. When G gets the realizability information of the current property from CB, it generates appropriate inputs (using GenRefute or GenStimulus) and sends the input vector to CB and the driver D.
- D simulates the module with the input from CB. VCS [15] of Synopsys is used for simulation.

Procedure SimulateCoverage outlines our top-level algorithm.

ALGORITHM 4. **Procedure SimulateCoverage**

SimulateCoverage
While (there are uncovered properties in CB)
Step 0: CB takes a new property $\mathcal{L}$
Step 1a: Set J to its initial state
Step 1b: Set $\widehat{O}$ = the initial block outputs of J.
Step 2: While (not end of simulation) begin
At each clock of the simulation do
2.1: M samples $\widehat{O}$ from J and puts $\widehat{O}$ in Ch_{m2cb}.
2.2: CB gets $\widehat{O}$ from M
2.3: CB rewrites $\mathcal{L}$ in terms of present state Boolean propositions and X-guarded temporal properties and substitutes $\widehat{O}$ in the non X-guarded terms of $\mathcal{L}$ to obtain $\widehat{L}$.
2.4: If $\widehat{L}$ = TRUE, report success and goto Step 0
2.5: If $\widehat{L}$ = FALSE, report failure and goto Step 0
2.6: Else CB sends $\widehat{L}$ to RC and waits for its feedback.
2.7: RC sends back the realizability of $\widehat{L}$ to CB.
2.8: CB puts the realizability of $\widehat{L}$ in Ch_{cb2g}.
2.9: G gets the realizability of $\widehat{L}$ through Ch_{cb2g}.
2.10a: If $\widehat{L}$ is unrealizable, G calls GenRefute($\widehat{L}$) to generate $\widehat{I}$
2.10b: Else G generates $\widehat{I}$ through GenStimulus($\widehat{L}$).
2.11: G puts the input vector $\widehat{I}$ in Ch_{g2d}.
2.12: D samples $\widehat{I}$ through Ch_{g2d} and simulates J with $\widehat{I}$.
2.13: CB samples $\widehat{I}$ through Ch_{g2cb}.
2.14: CB then obtains $\mathcal{L}'$ from $\widehat{L}$ by substituting $\widehat{I}$ in the non X-guarded terms of $\widehat{L}$ and dropping the left-most X from each X-guarded term .
2.15: CB updates: $\mathcal{L} = \mathcal{L}'$; goto Step 2
EndAlgorithm

6. RESULTS

We used our methodology on several verification IPs for standard Bus protocols. The test generation algorithms helped us to reach several complex coverage points in significantly less time as compared to a coverage driven constrained random verification approach. Table 1 shows the results of our tool for an industry standard ABV IP for the IBM CoreConnect bus protocol. It has 3 buses, namely DCR, OPB and PLB [7]. For each bus, we have a number of properties, a Bus functional model and a RVM-compliant coverage-driven randomized test bench. We present the results for a selected subset of the properties, which had complicated triggering conditions spanning multiple cycles.

Device	# Ip.	# Op.	#Prop.	IABV	CDR
DCR master	4	2	2	5000	7000
DCR slave	2	4	2	5400	8600
OPB master	6	6	2	6400	9200
OPB slave	5	4	2	6200	10800
PLB master	7	5	2	1000	6000
PLB slave	6	8	1	3000	4000

Table 1: Results on IBM CoreConnect

The first column of Table 1 refers to the device type, while the second, third and fourth columns are the number of inputs, outputs and properties respectively for each test. The last two columns of Table 1 compare the number of simulation cycles required to cover all properties non-vacuously using our tool (IABV) with that of the coverage-driven random (CDR) one. Results show the effectiveness of our approach in a coverage driven randomized test generation flow.

Acknowledgements

Pallab Dasgupta acknowledges the Department of Science & Technology, Govt. of India for partial support of this work.

7. REFERENCES

[1] Abraham, J.A., Vedula, V.M., and Saab, D.G., Verifying Properties Using Sequential ATPG; ITC'02, 194-202.
[2] Ammann, P.E. et al., Using Model Checking to Generate Tests from Specifications; ICFEM'98, 46-54.
[3] Clarke, E., German, S., Lu, Y., Veith, H., and Wang, D., Executable Protocol Specification in ESL; FMCAD'00, 197-216.
[4] Dill, D.L., Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits, ACM Distinguished Dissertations, MIT Press, 1989.
[5] Gargantini, A., Heitmeyer, C., Using Model Checking to Generate Tests from Requirements Specifications; SIGSOFT'99, 146-162.
[6] Gupta. A. et al., Property Specific Testbench Generation for Guided Simulation; VLSI'02, 524-534.
[7] IBM CoreConnect, http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs
[8] Pnueli, A., The temporal logic of programs; FOCS'77, 46-57.
[9] Pnueli, A., Rosner, R., On the Synthesis of a Reactive Module; In POPL'89, 179-190.
[10] Reference Verification Methodology for Vera, http://www.synopsys.com/products/simulation/pdf/va_vol4_iss1_vera.pdf
[11] Roy, S., Das, S., Basu, P., Dasgupta, P., Chakrabarti, P.P., SAT based solutions for Consistency Problems in Formal Property Specifications for Open Systems; ICCAD'05, 885-888.
[12] Shimizu, K., and Dill, D.L., Deriving a Simulation Input Generator and a Coverage Metric from a Formal Specification; DAC'02, 801-806.
[13] System Verilog, http://www.eda.org/sv/SystemVerilog_3.1a.pdf
[14] Tasiran, S. et al., Using a Formal Specification and a Model Checker to Monitor and Direct Simulation; DAC'03, 356-361.
[15] VCS, http://www.synopsys.com/products/simulation/