STAR: Generating input vectors for design validation by static analysis of RTL

Lingyi Liu and Shobha Vasudevan
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
{liu187, shobhav}@illinois.edu

Abstract—We introduce STAR, an automatic technique for functional input vector generation for design validation. STAR statically analyzes the source code of the Register Transfer Level (RTL) design. The STAR approach is a hybrid between RTL symbolic simulation and concrete simulation, that offsets the disadvantages of both the techniques. It allows deeper as well as wider exploration of the design space by varying the extent of concrete and symbolic simulation in a given run. STAR follows a region-wide notion of coverage, where the concrete simulation navigates to a region of the design space and the symbolic simulation explores it systematically. We demonstrate that preliminary results of using STAR are promising by showing high path coverage on benchmark RTL designs.

I. INTRODUCTION

Design validation/verification is widely accepted as responsible for the bottlenecks that plague modern system design. Directed application of input vectors captures only known behavior, and therefore, known flaws in the Register Transfer Level (RTL) design. Random vectors capture unintentional and often surprising design behavior, consequently they are more valuable. The process of deciphering the reaction of the design to the random stimulus, however, can take many man-months. This means that making the test suite mature enough to incorporate random stimulus behavior checking is an iterative and resource-intensive process.

It is desirable to have an input vector generation strategy that lies in the spectrum between directed and random input validation—one that can explore unknown territory, but in a controlled manner. In this work, we introduce STAR (Static Analysis of RTL), a technique for automatic generation of functional vectors in RTL using static analysis of the Hardware Description Language (HDL) source code. STAR uses symbolic simulation of the RTL in conjunction with concrete simulation to form a practically feasible, efficient input vector generation strategy.

Symbolic simulation at the gate level is an effective method to generate input vectors. In symbolic simulation, a design is simulated using symbolic values of inputs, in place of concrete values. Symbolic simulation in RTL can be done by static analysis of the HDL source code. Static analysis of RTL deciphers all possible executions of the Verilog program by traversing its source code. When used for reasonably sized systems, this exhaustive analysis can quickly get intractable. An input vector generation technique that uses purely symbolic simulation can therefore be practically infeasible. Instead of using a purely symbolic approach, we use an approach that directs the symbolic simulation to a certain region of the input vector space. The principle behind this technique is to explore regions of the input space, where a region is a structural conglomerate of many paths. Within a region, STAR exhaustively analyzes all the paths. The symbolic simulation process is navigated to a particular region using a concrete simulation trace.

The “divide-and-conquer” strategy of STAR can be described as follows for a combinational (or a single timeframe) design. A random concrete stimulus is applied to the design and an execution trace is obtained. An RT-Level symbolic simulation of the concrete trace is extracted. The symbolic expression that corresponds to the concrete execution trace is extracted. The expression is composed of guard (branch) conditions as well as assignment statements that use RTL operators. The conjunction of guards provides the path constraints under which the concrete trace is executed. One or more of these constraints are now inverted (or toggled). The resulting symbolic expression corresponds to another path in the design. A constraint solver is used to solve for the new constraints, and produce an input vector pattern that is a test for the new path. The inversion

1 We use Verilog programs in the same sense as used in [1]
of constraints can be done systematically to cover all paths in a region in a depth-first manner or a breadth-first manner. This process is repeated until the desired coverage objective is achieved. A new iteration of STAR will correspond to a different region of the design space.

We extend the STAR approach to sequential input vector generation over multiple time frames. We use a notion of RT-level symbolic simulation of combinational and sequential designs as presented in. We demonstrate that STAR’s approach provides high path coverage on benchmark RTI designs, thereby showing a lot of promise in abridging the chasm between directed and random testing. We use MathSAT, a SAT-based decision procedure for linear arithmetic logic [2], solving our path constraints.

A very interesting aspect of STAR is that it provides a “knob” with the degree of concrete and symbolic simulations per run. In a given run, the region of exploration can be varied by the number of timeframes (cycles) of the concrete simulation, as well as by directing it to a given unexplored unit. Once inside a target region, the region can be explored deeply, widely or in a zigzag manner, depending on the constraint inversion strategy. STAR can therefore be made very sensitive to the validation objectives.

Our contributions in this work are as follows:

- We introduce STAR, a functional input vector generation strategy in RTL that uses static analysis of the HDL source code.
- STAR uses hybrid concrete simulation traces as well as RT-Level symbolic simulation to generate input patterns.
- We extend the STAR technique to sequential input pattern generation.
- STAR follows a region-wise coverage strategy. The boundaries of a region are fluid, and can be varied along many degrees of freedom like depth, width or zigzag exploration tracks.

II. RELATED WORK

In software testing, the idea of combining concrete and symbolic executions has been explored extensively in recent times [3]–[5]. Static analysis has typically been used in the context of software. We view the RTI design expressed in an HDL as a “program,” and are thereby able to perform a similar static analysis on the HDL source code. Prior work has analyzed RTI statically for formal verification [11], [16]–[18]. Lightweight static analysis has also been used for functional testing [9], [10]. To the best of our knowledge, this class of techniques have not been applied to design validation.

Design validation using RT-I level has been researched before. An RT-level algorithm is proposed in [11], integrating linear-programming techniques for datapath modules and a 3-SAT solver for control logic. The technique targets statement coverage by sensitizing a set of paths that cover inter-module interfaces. In [12], each potential execution path is enumerated through the analysis of behavioral HDL programs using annotations. In [13], the HDL model is translated into a set of integer constraints automatically. Scenario or context constraints are also generated for a given verified function point. The integer constraint solver is employed to generate the functional patterns.

Concrete and symbolic executions have been used before to perform a hybrid verification at the gate level. [14] leverages the dynamic simulation method to guide the state-space exploration. Input vectors are generated at the gate-level with combination of BDD and ATPG engines.

III. ALGORITHM OF STAR

Figure 3 shows the block diagram corresponding to the different phases in the STAR algorithm. We describe them below.

A. Code Instrumentation and Concrete Simulation

A set of concrete inputs is applied to the target module in the design using a commercial simulator like Synopsys VCS. The HDL source code is instrumented with annotations that act as a communication channel between the concrete and symbolic simulations. As the concrete simulation proceeds, for every statement in the RTL that is executed, a function call is made to the symbolic simulator. The symbolic simulator now extracts the symbolic expression that caused the most recent concrete simulation to occur. For every concrete execution in the RTL, the symbolic execution follows the concrete path, forming an expression for the executed statement. This closely coupled combination between the concrete and symbolic simulations is due to the instrumentation.

In certain cases, a symbolic expression need not be generated for the corresponding concrete execution. For example, in the case of a rare event like the reset signal, a concrete value of 1 can be assigned to it for the sake of simplifying the symbolic expressions. This, and other such constants, concrete value assignments may lead to transitive forward propagation of these constants. This
may also reduce the size or complexity of the symbolic expression generated from the concrete execution. If a bug is hit during the concrete execution, it terminates the current execution of the algorithm and a new run is started. The failing input pattern is recorded.

B Symbolic Simulation and Constraint Generation

The symbolic simulation of STAR is at the RTI level. This means that the symbolic expression has operators at the RTI level of abstraction, instead of at the bit level. This is in contrast to traditional gate level symbolic simulation techniques [15]. Although word-level symbolic simulation techniques [16] achieve the data hiding advantage, our symbolic simulation achieves more scalability due to a higher level of abstraction. Our level of abstraction is the term [17], where every RTI variables are treated as integers, and the operations like integer operations. The operations are left uninterpreted, but term level rewriting is performed on them, as the symbolic simulation proceeds. This term level rewriting helps in reducing the complexity of the symbolic term (expression). Finally, the symbolic expression representing a design variable is given a Boolean interpretation.

A function computes the symbolic expression at a given cycle of simulation i.e., an executed RTI statement. For every assignment to a given variable, it substitutes the right hand side of the definition for the variable for an assignment statement, along with the control signal information required for executing the assignment. We call an individual control signal as a guard. The Boolean expression that includes all the guards required to execute an assignment is called a path constraint, or simply a constraint. For every executed statement, constraints are extracted in this manner.

C Sequential Unrolling

In the case of combinational or single frame designs, a single concrete input pattern is applied. In the case of sequential designs, the design needs to be unrolled to simulate its future behavior. The concrete input patterns are applied for a pre-determined number of cycles. Every concrete simulation is annotated with the "cycle number" at the relative time when it is executed. These annotations are used during the symbolic expression generation. A symbolic expression corresponding to a concrete execution is annotated with the cycle in which it holds true. This means that every variable \( v \) in the design is annotated with the cycle \( t \) at which it is executed in the concrete simulation, making \( v(t) \), the complete definition of a sequential variable. Every instance of \( v(t) \) is regarded as a separate variable by the constraint solver. Theoretically, the unrolling needs to be performed until a fixpoint is reached. However, practically, the extent of unrolling is determined by some design-specific heuristics. This is because practical designs are not intended to span an unlimited number of cycles. A typical module of interest would have repetitive behavior after a few cycles. This implies that the unrolling until a designer specified cycle will capture a sizable portion of the design behavior.

D Path Exploration and Constraint Solving

The constraint generated by the symbolic simulation step has many guards in it, representing the concrete execution path that was just taken. At this step, one or more of these guards is inverted. The resulting constraint with inverted guards corresponds to another path in the design. The constraint inversion step, therefore, enables the exploration of different paths in the design. Depending on the number and type of guards chosen for inversion, different paths in the neighborhood or region of the concrete execution can be explored.

The concrete simulation is used as the directive that navigates the search to a region of interest. Within that region, the constraint inversion process allows an exhaustive exploration. In the current implementation of STAR, we perform a bounded depth first search and choose the last guard in the constraint to negate. The inverted constraint is then given to a constraint solver.

The constraint solver used in this work is MathSAT [18] [2], an efficient DPLL-based decision procedure. The constraint solver searches a satisfiable input pattern.
for the given constraints.

The inverted constraint, if satisfiable, will produce a set of assignments that can be considered a valid input test pattern for the execution path that it represents. If the constraint is unsatisfiable due to a conflict in the guards, another guard is inverted and the new constraint is passed to the solver.

When all paths are enumerated on a coverage requirement is satisfied, the algorithm terminates.

IV. EXAMPLE

We illustrate the effectiveness of our algorithm by explicitly walking the reader through a sequential RTL design, that involves multiple processes. The source code is shown in Figure 2. We call the first process P1 and the second process P2.

The comments following each statement compose the instrumented code for communication between the concrete and symbolic simulations. The variables values in the instrumented code are monitored during simulation. If a variable value changes, it is likely to have been touched by the concrete execution process. The symbolic simulator will try to track the value of the changed variable.

```plaintext
module (d,e,f,g,clk)
  1. input d, e, f, g, clk;
  2. output c;
  3. reg a,b;
  4. always@ (posedge clk):
  5.   if (d > b)
  6.     c <= d; //i1=i1+1;
  7.   else e <= c; //i2=i2+1;
  8.  always@ (posedge clk):
  9.   f(d > e)
 10. begin
 11.   a <= [f & g] //i3=i3+1
 12.   b <= [f | g] //i4=i4+1
 13. end begin
 14. else
 15.   begin
 16.     a <= [f | g] //i5=i5+1
 17.     b <= [f & g] //i6=i6+1
 18. end begin
 19. endmodule
```

Fig 2. An example run through STAR.

In the above example, we unroll the RTI model for a depth of 2! Here, the index i for a[i] is the annotation corresponding to the cycle in which that variable is assigned a concrete value. This can be thought of as a “cycle number”. The concrete random inputs applied to each input are as follows. The initial state of the system is assumed to be a[0] = 0, b[0] = 0. In cycle 0, the inputs are:

- d[0] = 0, e[0] = 0, f[0] = 0 and g[0] = 0. In cycle 1, the inputs are:

We identify the executed paths by their line numbers. Due to the concrete input, the path 15-17 is executed in P1 and 19-14 is executed in P2 and 19-14 is executed in P2. Both are executed. After symbolic simulation, the generated constraint is:

\[(a[0] \leq b[0]) \land (d[0] \leq e[0]) \land (a[1] \leq b[1]) \land (d[1] \leq e[1]) \land (a[1] = f[0] | g[0]) \land (b[1] = f[0] & g[0]).\]

The next step is the inversion of a guard in the constraint to generate the new path constraints. The guard \((d[1] \leq e[1])\) is selected for inversion since it is the last guard in the constraint. The new path constraint then is:

\[(d[0] \leq e[0]) \land (a[1] \leq b[1]) \land (d[1] > e[1]) \land a[1] = f[0] | g[0]) \land (b[1] = f[0] \land g[0]).\]

Solving this new constraint with MathSAT solver will produce the next input pattern:

\[d[0] = 0, e[0] = 1, d[1] = 1, e[1] = 0, f[0] = 1, g[0] = 1\]

Simulation with the new input patterns will cover the if-branch of P2.

From the example above, there is a noteworthy point. The inversion of one guard in the path constraint will always guide the next concrete path toward a different branch of the negated guard that has not been explored before. Comparing with the random input vector generation method, this algorithm can make sure that each path in the RTI is explored exactly once with one pattern and therefore largely remove the redundancy in the input patterns. The guards in the design serve as the criterion for dividing the input space into several sub-regions of equivalence. Each combination of these guards represents one equivalence class. An input pattern satisfying one combination of guards can be selected to represent the corresponding class.

VI. PRELIMINARY RESULTS

We have implemented a prototype of STAR in C++. We present a set of experimental results on some examples of RTL-level model. The Synopsys VCS simulator
<table>
<thead>
<tr>
<th>Design Name</th>
<th>Number of patterns (STAR)</th>
<th>Statement Coverage (STAR)</th>
<th>Branch Coverage (STAR)</th>
<th>Unrolled Depth (STAR)</th>
<th>Number of patterns (Rand)</th>
<th>Statement Coverage (Rand)</th>
<th>Branch Coverage (Rand)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b01</td>
<td>55</td>
<td>100%</td>
<td>100%</td>
<td>6</td>
<td>100</td>
<td>85.94%</td>
<td>70%</td>
</tr>
<tr>
<td>b01</td>
<td>17</td>
<td>65.7%</td>
<td>55.4%</td>
<td>4</td>
<td>100</td>
<td>85.94%</td>
<td>71%</td>
</tr>
<tr>
<td>b06</td>
<td>8</td>
<td>88%</td>
<td>88.8%</td>
<td>10</td>
<td>100</td>
<td>84.42%</td>
<td>78.95%</td>
</tr>
<tr>
<td>b1U</td>
<td>54</td>
<td>96%</td>
<td>87.5%</td>
<td>10</td>
<td>100</td>
<td>71.05%</td>
<td>57.58%</td>
</tr>
</tbody>
</table>

Fig 3 Experimental results STAR vs Random

is employed to do concrete simulations. Our preliminary results are shown on ITC99 benchmarks.

We demonstrate the effectiveness of STAR with respect to two aspects. First, we compare structural coverage (such as branch and statement coverage) acquired by applying input patterns generated by STAR against those generated by random input generation. In addition, we consider the number of patterns as a parameter to quantify the quality of the input vectors. Covering more statements or branches with fewer input patterns is the most ideal case. It should be noted that our method prevents redundancy or repetition of each input patterns.

From the coverage report and number of patterns in the experimental results shown in Table 3, STAR can efficiently cover more statements and branches in the RTL model with fewer input patterns as compared to the random generation method.

Theoretically, STAR can achieve complete statement or branch coverage if all statements in the design are reachable and all branches are independent. However, limited by the constraint of time and space, it is impractical for STAR to unroll the sequential design for hundreds or thousands of frames. Interestingly, from b01’s results, coverage is very low when unrolling the RTI for only four frames from the designated initial states. This is because many states cannot be reached from the initial states within only four frames. However, when the sequential depth is increased to 6 cycles, the coverage by STAR is very comparable to that of random input generation.

VI. CONCLUSION AND FUTURE WORK

In conclusion, STAR can be viewed as a bridge between random and directed testing. It leverages the power of two technologies—concrete and symbolic simulation, to provide a notion of region-wise coverage. This enables systematic exploration of regions of the design space that might have gone unaddressed, in an automatic manner.

In STAR, the designated execution cycle numbers also influence the path exploration since the algorithm always starts from an initial state and executes for a pre-determined number of cycles. If the sequential depth of some variable assignments is more than the depth of the concrete simulation, the statement including those variables will not be covered. One solution is to increase the executed cycles at the expense of longer constraint solver runtimes. Another solution is to replace the initial state with new reached states in the designated cycles. These two methods for improving coverage will be included in our future work.

In future work, we will use different degrees of hybridization of the concrete and symbolic simulation in STAR and analyze the point of maximum practical benefit in this hybrid spectrum.

REFERENCES