QUEBS: Qualifying Event Based Search in Concolic Testing for Validation of RTL Models

Alif Ahmed and Prabhat Mishra
Department of Computer & Information Science & Engineering
University of Florida, USA

***Abstract*—Input vector generation is an important step during validation and debugging of hardware designs. Validation using random and directed random tests are widely used today. However, these methods can lead to unacceptable functional coverage under tight deadlines. Concolic testing is a semi-formal method to address this issue. It combines concrete simulation guided by symbolic execution. Application of concolic testing in hardware domain is still in its infancy due to the lack of effective traversal strategies. In this paper, we present Qualifying Event Based Search (QUEBS) heuristic for concolic testing. During exhaustive concolic testing, same branch may be selected many times for traversal. Our heuristic limits the number of times a branch can be selected. By preventing repeated selection, it facilitates wider coverage within limited time. Also, whenever a previously uncovered branch is encountered, this limit is relaxed to permit thorough exploration of the newly reached area. Our experimental results demonstrate that this approach provides better branch coverage than state-of-the-art test generation methods in a given time budget. To further improve the performance of QUEBS, we provide two optimization techniques - unsolvable branch elimination and incremental solving by context reuse.**

I. Introduction

Verification is a major bottleneck in modern chip design life cycle. According to [1], more than 70% of the resources and engineering time is spent in verification efforts. Industry uses many forms of automation to speed up this process. One of them is random testing - where a large number of random input patterns are applied to the design, and corresponding outputs are checked for correctness. While random testing is scalable and can be applied to large designs, the random nature of produced tests tends to give low coverage and usually does not cover all corner cases. To cover these corner cases, directed tests are written manually by verification engineers. Writing directed tests are very difficult and time consuming, especially for large and complex designs. Some formal and semi-formal methods have been proposed to automate this test generation process [2]–[5]. However, formal methods such as model checking unrolls the whole design statically. This makes such methods prone to state explosion, and their applicability is limited to small designs [6].

On the other hand, semi-formal methods like concolic testing uses concrete simulation along with symbolic execution. It was first introduced in software domain [7] [8] and later adopted into hardware domain [9]. In concolic testing, the design is first simulated with an initial set of inputs. The path that is taken by the simulation is called an execution path. All the logical expressions in an execution path can be represented as path constraints. Now, one of the branches that is adjacent to the execution path is chosen to be explored next and path constraints leading to this branch are given to a constraint solver. If the constraint solver comes up with a solution, the generated input assignments can be used as a test case to activate that branch. If the constraint solver fails to come up with any solution, a different branch is chosen to be explored next. This procedure is repeated until all the branches are covered or some other terminating conditions are met. Concolic testing method avoids state explosion by exposing only a particular execution path to the constraint solver, not all possible paths at once. This advantage made concolic testing a widely used test generation technique in software domain.

How to select the next branch in concolic testing is an active research area. Many selection strategies have been analyzed in the literature, including random, depth-first search (DFS), breadth-first search (BFS), context guided, coverage guided etc [9]–[13]. DFS and BFS are exhaustive search strategy - they guarantee full coverage by exploring all possible paths. However, as number of paths are exponential to the number of branches, employing such exhaustive strategy is not practical even for small designs [9]. Other search heuristics tackle this path explosion issue by restricting which branches can be selected. However, once restricted, these heuristics do not consider these branches for exploration anymore, even when exploring them might lead to unexplored territory.

In this paper, we present QUEBS - QUalifying Event Based Search strategy for concolic testing. *QUEBS maintains a balance between the exhaustive and restrictive search techniques*. Our proposed approach uses bounded DFS for traversing the execution tree. However, unlike DFS, it sets a limit on how many times a branch can be selected for exploration. Intuition behind limiting is that unless the context is radically changed, exploring the same search region many times will not be beneficial. Events that indicate a change in context is defined as qualifying events. In case of such a qualifying event, the limit on branch selection is relaxed - permitting thorough exploration. Specifically, QUEBS use limiting to prevent path explosion and to quickly guide concrete simulation to a new search region; and it uses relaxing to allow careful exploration once that region has been reached. Additionally, this paper presents two optimization methods for QUEBS. First optimization method reduces failed calls to constraint solver by eliminating many unsolvable branches beforehand. The second

This work was partially supported by grants from NSF (CNS-1441667), SRC (2014-TS-2554) and Cisco.

optimization benefits by reusing overlapping contexts.

The primary contributions of this paper can be summarized as follows:

- Proposes Qualifying Event Based Search heuristic (QUEBS) for concolic testing. Proof of concept is provided by considering uncovering of new branch as an qualifying event.
- Presents a novel optimization method for unsolvable branch elimination using static transitive dependency analysis.
- Proposes incremental solving by constraint context reuse. We have considered two types of context reuse - intra simulation and inter simulation.
- Provides experimental data supporting effectiveness of QUEBS as well as the two optimizations methods.

The remainder of the paper is organized as follows. Prior works on test generation techniques are discussed in Section II. In Section III, necessary background on concolic testing is provided. Section IV describes QUEBS strategy in details. Section V gives insight into our proposed optimization techniques. In Section VI, experimental results compare our approach with state-of-the-art methods. Finally, we conclude our paper in Section VII.

II. Related Work

Idea of concolic testing was first introduced in software domain as DART [8]. Primary goal of DART was to automatically generate test cases for finding bugs in software. It was then extended by CUTE and jCUTE, which added more programming language constructs and support for multi-threading [7] [12]. Later, Liu et al. proposed STAR, which showed that same concept can be equally applicable for verifying hardware designs [9]. However, all these methods used DFS search strategy. Exhaustive strategies such as DFS or BFS are prone to path explosion problem and does not scale well with program size [14]. Also, as these strategies are very thorough and localized, given a limited time budget, they can explore only a narrow portion of the overall search space.

A wide variety of search heuristics have been proposed to overcome this path explosion problem faced by concolic testing. Authors of STAR recently proposed an enhanced version which counters path explosion by utilizing the concept of state caching [15]. In their approach, they cached the fully explored states using bitmap encoding of branches. Whenever a cached state is reached again, all the paths from that state is skipped. In software domain, Burnim et al. proposed heuristics based on random selection [10]. As these heuristics rely on randomness, they have shown varying effectiveness in different benchmarks. In the same article, authors proposed Control-Flow Directed Search. In this approach, distance to a target branch is calculated for all branches in the current execution path and the one with minimum distance is selected. Intuition behind this is - branch with minimum distance will most likely lead to that target branch. HYBRO is a similar approach for RTL designs, which uses static and dynamic analysis of CFG [11]. In HYBRO, control dependency of all branches are determined statically from CFG. During runtime, a branch can be selected only if it has uncovered control dependent child branch. Qin et al. proposed a method similar to HYBRO, but with added support for dynamic array references [16].

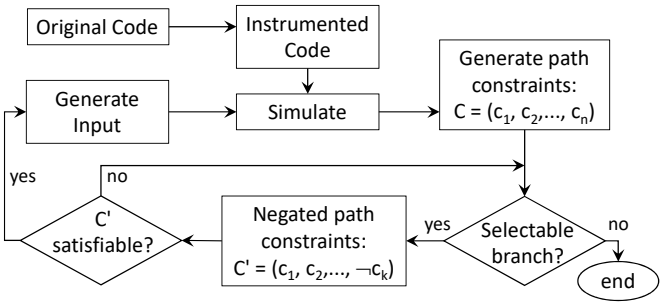

Fig. 1. Typical flow for a concolic testing engine

CarFast is another coverage guided search heuristic [17]. It selects branches based on incremental coverage gain - where coverage gain is determined by the number of uncovered statements that can be reached by selecting that branch. A similar technique is proposed by Li et al. [18]. In their approach, each branch stores how many times its *length-n subpath* is traversed. Here *length-n subpath* denotes the preceding n branches. Branch with least frequently traversed subpath is selected for exploration. An improved method is proposed by Seo et al. named Context Guided Search (CGS) [13]. It builds on the same concept of *length-n subpath*. However, unlike prior approach, which only considered fixed n, CGS dynamically increases n during runtime. Also, CGS does not allow a branch to be selected if its subpath is traversed before. CGS outperforms other search heuristics in software domain with higher coverage in less iterations. However, in case of hardware designs, same state can create different contexts due to concurrency. This limits the effectiveness of CGS for hardware designs.

III. Background: Concolic Testing

Figure 1 shows the standard flow for concolic testing engines. It starts by instrumenting the original code. Purpose of instrumentation is to trace the execution path during simulation. One of the main criteria of instrumentation is that it should not affect the functional behavior of the original code in any way. The instrumented code is then compiled and simulated with an initial input vector. As the compiled code is instrumented, simulation will dump a trace of the execution path. All of the execution paths together form the execution tree. Now, a previously unexplored path is selected from the execution tree - usually by negating one of the branch conditions. All the path constraints up to that selected branch along with the negated branch is then given to a constraint solver (shown as C'). If C' is satisfiable, same process is repeated with the solution input set. If C' is not satisfiable, a new branch is selected. The process is terminated if no branch is left to be selected. Sometimes other terminating conditions are also used, like a specific coverage goal, number of iterations or timeouts.

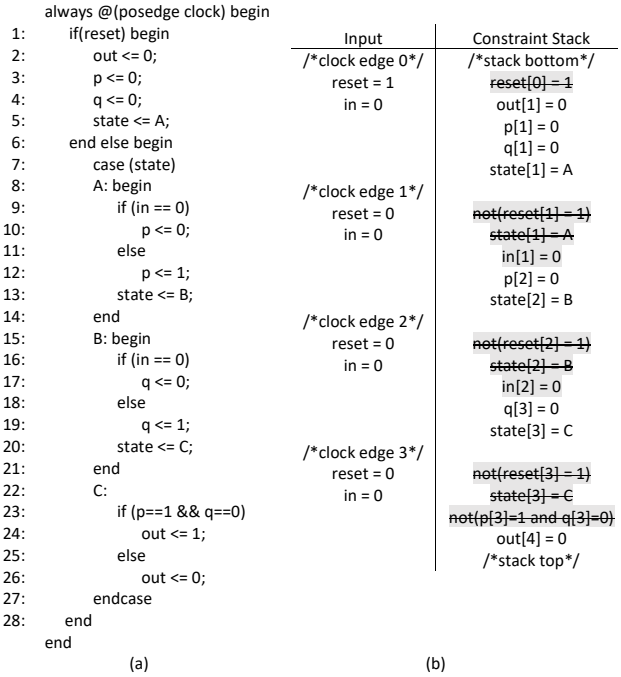

Fig. 2. An example showing constraint stack generation - (a) RTL code for program under test, (b) Constraint stack for a particular input vector. Grayed lines are branch constraints. Struck branches are pruned during optimization.

How the execution tree is explored defines the search strategy and is a crucial performance determinant. Our proposed method uses bounded DFS strategy at its core, like many others [10]–[12], [16]. In software domain, depth is bounded by number of branches in execution path. For RTL designs however, it is usually done by bounding the number of unrolled cycles. To facilitate depth-first search, a constraint stack is maintained. An example constraint stack is shown in Figure 2(b). This particular stack is generated when the design is unrolled for four clock cycles and executed with in set to zero. For selecting next branch, bounded DFS starts with the branch closest to the top and then proceeds towards the bottom. Search ends if it reaches the bottom of the stack without finding any explorable branch. QUEBS also follows bounded DFS, details of which is discussed in the next section.

IV. Qualifying Event Based Search (QUEBS)

QUEBS modifies the standard bounded DFS by introducing the idea of *qualifying event*, *limit* and *limit relaxation*. As we already know, bounded DFS is exhaustive within the bound. That means it will go through all possible execution paths. It guarantees that if there exists any input vector for which a branch can be reached within bounded clock cycles, bounded DFS will find it. However, exploring all path becomes infeasible when design size or bound increases. QUEBS avoids this path explosion issue by employing limits on branches. Limit can be defined as the maximum number of time a branch can be selected for exploration. Limiting forces change in search region. This limit can be changed to manipulate how thoroughly a region will be searched before moving onto a new region. However, employing limit can negatively affect the coverage. To ensure high coverage, QUEBS uses the idea of limit relaxation. Relaxation occurs when searching a previously explored region might be beneficial. In case of relaxation, all the branches are allowed to be selected again. Events that trigger relaxation are defined as qualifying events. In this paper, we have considered covering of an unexplored branch as a qualifying event.

A. QUEBS Algorithm

Conceptually, algorithm of QUEBS is very simple. It maintains a counter for each branch. Before selecting a branch, it checks whether the counter is less than the limit. If it is, then it tries to generate input by invoking constraint solver. In case solver also succeeds, counter value of that branch is increased and next iteration follows with generated input. Furthermore, if a qualifying event occurs, counter value of all branches except the last selected one is reset to 0.

Algorithm 1 Qualifying Event Based Search (QUEBS)

Input: Program under test, P
Output: Test vectors

Set of test vectors, $T \leftarrow \emptyset$
Set of unsolvable branches, $B_U \leftarrow branch\_elim(P)$
Last selected branch, $b_{last} \leftarrow \emptyset$
Input vector, $I \leftarrow random()$
repeat
  $T \leftarrow \{I\} \cup T$
  Execution path trace, $\pi \leftarrow simulate(I)$
  $update\_coverage(\pi)$
  if coverage goal is met **then**
    return T
  else if new branch covered in π **then**
    // qualifying event, reset branch selection count
    for all branch, $b \in P$ and $b! = b_{last}$ **do**
      $b.count \leftarrow 0$
    end for
  end if
  Constraints stack, $C \leftarrow build\_stack(\pi)$
  while C not empty **do**
    top constraint, $c \leftarrow C.pop()$
    if c type is branch **then**
      $b \leftarrow \neg c$
      if $(b \notin B_U)$ and $(b.count < K)$ **then**
        $I = constraint\_solver(C + b)$
        if I is valid **then**
          $b.count = b.count + 1$
          $b_{last} = b$
          break // execute with new input
        end if
      end if
    end if
  end while
until I is not valid
return T

Algorithm 1 shows this procedure. It takes the design as input and provides a set of test vectors (T) as output.

Initially, T is empty. A one-time static analysis is done at this stage to determine a set of unsolvable branches, B_U (line 2). These branches are skipped during selection process to reduce unnecessary solver calls. Details of this pruning technique is deferred to Section V.

After the initialization and static analysis is done, QUEBS goes through the standard procedure of bounded DFS concolic testing (line 5-32) until the coverage goal is met or no valid input set is found to continue iterations.

In each iteration, the design is simulated with current input vector I and execution path trace π is generated (line 7). Next, branch coverage is calculated for this path (line 8). If coverage goal is met, then test vector set T is returned (line 9-10). Otherwise, if a new branch is covered (indicated by increase in coverage), it is considered as a qualifying event and selection count of branches are reset (line 11-16). Last selected branch (b_{last}) is skipped from reset to prevent unnecessary exploration of same execution path. In the next step, constraint stack C is generated from π and searched in a depth-first manner for next candidate branch (line 18-31). After such a branch is found (b in line 21), two types of checking are done before giving it to the constraint solver. First, it is made sure that the branch under consideration (b) is not pruned during the optimization stage ($b \notin B_U$). Second, the selection count must be less than predefined limit ($b.count < K$). Once these criteria are met, path constraints leading up to b is given to the constraint solver (line 23). If the constraints are unsatisfiable, I becomes invalid. Otherwise, we found the input vector for the next iteration and it is assigned to I. In this case, selection count of b is increased, b_{last} is updated, and loop within constraints stack is terminated so that the next iteration can start (line 25-27). If no satisfiable branch is found, condition in line 32 becomes false and QUEBS ends by returning T.

B. Illustrative Example

Figure 3 explains how QUEBS systematically explore branches with limit K set to 1. The CFG shown in Figure 3(a) refers to the example RTL code of Figure 2, unrolled for three clock cycles after reset (total four). Each white node represents a branch of that example. Left path of these nodes are taken in case the branch condition is $true$ and right path is taken if $false$. This CFG does not contain reset branches and pruned case statements to increase clarity.

The design is first simulated with a random input vector, as shown in Figure 3(b). It goes through branch b_9, b_{16} and b_{25} (solid line). Here b_i refers to the i-th line in example code, and $count$ denotes how many times it is selected for exploration. Generated constraint stack for this particular path is shown in Figure 2(b). Now, as QUEBS uses DFS, it will start with the last branch b_{25} and will go up in the tree to find the next branch to explore. Opposite branch of b_{25} is b_{23}. However, b_{23} has been optimized (removed) during *'unsolvable branch elimination'* phase, and thus cannot be selected. Details of this optimization is discussed in Section V. Even if b_{23} was selected, it would have conflict with previous constraint $p[1] = 0$. Next branch in the path is b_{16} with opposite branch b_{18}. Since b_{18} has count less than limit K, it is selected for next iteration. To see if this path is feasible, following constraints are given to the solver: $(out[1] = 0) \wedge (p[1] = 0) \wedge (q[1] = 0) \wedge (state[1] = 0) \wedge (in[1] = 0) \wedge (p[2] = 0) \wedge (state[2] = 1) \wedge \neg(in[2] = 0)$. This is satisfiable and solver returns new input set for next iteration. Also, count of b_{18} is increased by 1. Execution path for this new input is shown in Figure 3(c).

Iteration 1 follows the same procedure as iteration 0. However, branch b_{18} is a previously uncovered branch. Going through it triggers relaxation of limit for all branches except b_{18}. As for iteration 2, branch b_{16} is selected and its count is increased. Execution path for this iteration is shown in Figure 3(d). During this iteration, b_{18} could not be selected, because b_{18}'s count value is no longer less than limit K of 1. Going further up, branch b_{11} does not cause any violation, and is selected for iteration 3. Execution path of this is shown in Figure 3(e). Here b_{11} is a previously uncovered branch - triggering a relaxation of limit for b_{16} and b_{18}. As count of b_{16} becomes 0, now it can be selected for iteration 4. This iteration is shown in Figure 3(f). Here branch b_{23} is finally covered. As all branches are covered, the QUEBS algorithm terminates at this point.

C. Complexity Analysis

Let n be the number of branches and K be the selection limit. In the worst case, n branches can be selected at most $(n.K)$ number of times before being reset. Also, in the worst case, there can be at most n resets. This simple analysis gives an upper bound to the number of iterations to be $O(n^2K)$. In each iteration, there can be $(n-1)$ unsatisfiable solver calls. This makes the upper limit of constraint solver call to be of $O(n^3K)$. An interesting point to note here is that the upper limit is independent of the unroll cycle. However, actual iterations will increase with unrolled cycles because previously unreachable branches will be accessible. Since the number of solver calls is polynomial in the worst case, QUEBS does not suffer from path explosion problem and can be applied on large designs.

V. Optimizations

A. Unsolvable Branch Elimination

This optimization method detects branches that, when given to constraint solver, will always return unsolvable. Many unsuccessful solver calls are thus avoided by skipping these branches during exploration.

For better explanation, two types of variables are defined. Variables that are transitively connected to functional input ports are $flexible$ variables. Others are $strict$ variables. Also, it can be observed that execution paths solely depend on input at different clocks cycles. Thus constraint solver can generate input for negated branch condition only by changing flexible variables. If the branch condition does not contain any flexible variables, constraint solver can not come up with a solution. This insight is used to statically prune branches that are not connected to any input variables. Note that non functional inputs such as $clock$ and $reset$ signals belong

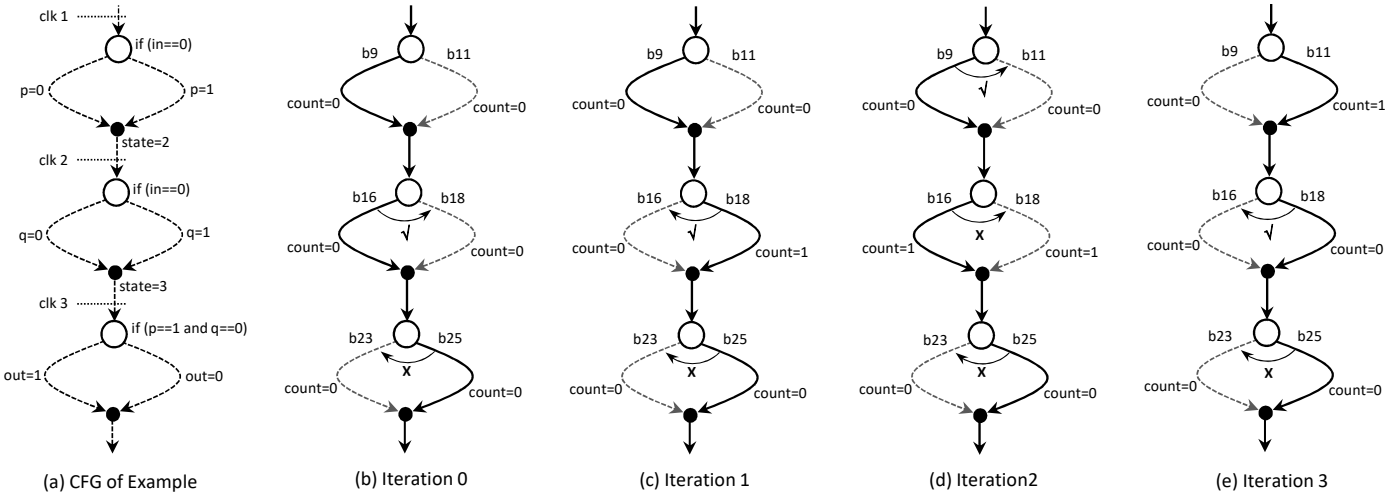

Fig. 3. Iterating through the example using QUEBS strategy. Pruned reset and case branches are not shown for clarity.

to strict category, because they are not used for exploring different paths. The basic idea of early pruning of reset branches already exists in literature [16].

Algorithm 2 describes the procedure for unsolvable branch elimination. It consists of several steps. Initially, all variables representing functional input ports are marked as flexible and all the other variables as strict (line 3 to 10). However, flexible property can propagate to strict variables via assignments. For example, in the assignment $x = y + 1$, if y is flexible, x also gets that property. For propagation, we need to know the dependencies of variables. Line 11 through 17 are used for generating this dependency list by going through all the assignments. After getting the dependency list, flexible property is propagated to all connected variables using BFS traversal (line 18 to 23). The last step is to check if a branch is unsolvable or not. If all the variables in a branch condition are strict, then it is marked as unsolvable (line 24 to 28).

For the example shown in Figure 2(a), initially in is marked as flexible. $clock$, $reset$, p, q, $state$ and out are marked as strict. As in is not assigned to any of them, they remain as strict variables after property propagation stage. This makes branches at line 1, 6, 8, 15, 22, 23, 25 to be marked as unsolvable. As shown in Figure 2(b), they are skipped during branch selection procedure.

There are several advantages of this optimization technique. First of all, this technique is generic and can be applied to other search heuristics as well. As this technique is not dependent on number of cycles unrolled, it is equally applicable for unbounded search heuristics. Furthermore, this method uses static analysis and only need to be done once during whole test generation process. Most importantly, as the number of possible execution paths is exponential with respect to the number of branches, even pruning a few branches will have significant impact on the number of solver calls and overall runtime.

Algorithm 2 Detection of Unsolvable Branch

Input: Program under test, P
Output: Unsolvable branches
// initialization
1: Set of unsolvable branches, $B_U \leftarrow \emptyset$
2: Queue of flexible variables, $Q_{flex} \leftarrow \emptyset$
3: **for all** variables, v used in P **do**
4: Set of variables that depends on v, $v.D \leftarrow \emptyset$
5: **if** v is an input port **then**
6: mark v as $flexible$ and insert it into Q_{flex}
7: **else**
8: mark v as $strict$
9: **end if**
10: **end for**
// variable dependency resolution
11: **for all** assignments, a in P **do**
12: $V_R \leftarrow$ Set of variables on right side of a
13: $V_L \leftarrow$ Set of variables on left side of a
14: **for all** $v \in V_R$ **do**
15: $v.D \leftarrow V_L \cup v.D$
16: **end for**
17: **end for**
// transitive propagation of $flexible$ property
18: **while** Q_{flex} not empty **do**
19: $v_{flex} \leftarrow Q_{flex}.pop()$
20: **for all** $v \in v_{flex}.D$ **do**
21: mark v as $flexible$ and insert it into Q_{flex}
22: **end for**
23: **end while**
// evaluate branches
24: **for all** branches, b in P **do**
25: **if** all variables used in b are $strict$ **then**
26: $B_U \leftarrow \{b\} \cup B_U$
27: **end if**
28: **end for**
29: **return** B_U

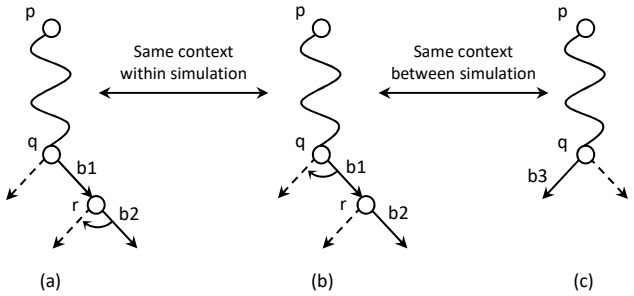

Fig. 4. Context reuse scenarios. (a)-(b) shows intra-simulation context reuse. (b)-(c) shows inter-simulation.

B. Incremental Solving by Context Reuse

Search strategies utilizing DFS have significant common path between consecutive solver calls and also between iterations. Based on this observation, we introduce two types of context reuse schemes: inter-simulation and intra-simulation. Figure 4 illustrates both scenarios. In Figure 4(a), a concrete simulation with execution path $(p \rightarrow q \rightarrow r)$ is shown. When applying DFS strategy on top of it, branch $b2$ will be negated first, and path constraints of $(p \rightarrow q \rightarrow r) \wedge \neg b2$ will be given to the constraint solver. Lets assume that it is unsatisfiable. Next, branch $b1$ will be negated and path constraints of $(p \rightarrow q) \wedge \neg b1$ will be given to the constraint solver (shown in Figure 4(b)). It is evident that path $(p \rightarrow q)$ is common in both cases. As all the constraints of path $(p \rightarrow q)$ is given to solver before, a model is already available and can be reused. This way all the previous learnings can be effectively utilized. We need to save the context before each branch, which is $(p \rightarrow q)$ for this example, and restore it when required. This is defined as intra-simulation context reuse.

Context reuse is applicable even between simulations. For example, consider the case in Figure 4(b). If negation of branch $b1$ was successful, next iteration will start with the generated input. Execution path for this input is shown in Figure 4(c). Since path $(p \rightarrow q \rightarrow r)$ is same for both (b) and (c), we can reuse the contexts that were created during (b). This idea is defined as inter-simulation context reuse. However, due to concurrent procedural blocks of RTL, the sequence in which they will be executed is non-deterministic. This introduces an issue where path $(p \rightarrow q \rightarrow r)$ in (c) is not exactly same as in (b). To solve this problem, trace of consecutive simulations are matched to make sure they are same up to the negated branch. Context of previous simulation is reused only when the path matches. Otherwise, path constraints are built from scratch. From now on, this recovery mechanism will be referred as $context\ recovery$. Less invocation of context recovery will result in more inter-simulation context reuse.

Algorithm 3 presents procedure for intra and inter-simulation context reuse. DFS portion of it is same as Algorithm 1. However, it omits the QUEBS specific details and focuses on context reuse part. In this algorithm, S_{ctx} is the solver context. It is the internal data structure maintained by the solver for all the added constraints. During building the stack, constraints are added to this context (build_stack(), line 7). Whenever a branch is encountered, a snapshot of the context before adding that branch is saved for later reuse (build_stack(), line 3-5). At the step where a negated branch is given to constraint solver, instead of rebuilding the whole context, previously saved context up to that branch is restored (main procedure, line 16). This is intra-simulation context reuse. On the other hand, for inter-simulation, position of the negated branch in trace file is saved (line 19). In the next iteration, trace up to that saved position is matched (line 8). In case of mismatch, context recovery mechanism is applied (line 9-10).

While context reuse could speed up the test generation process by significant amount, solver tools must support push/pop mechanism for saving and restoring contexts. We have used Yices 2.5.1 SMT solver in our experiments, which supports this function [19]. Other popular SMT solvers such as Z3 and boolector also supports this feature [20] [21].

Algorithm 3 Intra- and Inter-Simulation Context Reuse

Input: Program under test, P
Output: Test vectors

```
1:  Set of test vectors, T ← ∅
2:  Input vector, I ← random()
3:  Solver Context, S_ctx ← ∅
4:  Trace position marker, τ ← 0
5:  repeat
6:      T ← {I} ∪ T
7:      Execution path trace, π ← simulate(I)
8:      if π changed from previous iteration before τ then
9:          S_ctx ← ∅ // Context is changed. Recover context
10:         τ ← 0
11:     end if
12:     Constraints stack, C ← build_stack(π, τ, S_ctx)
13:     while C not empty do
14:         top constraint, c ← C.pop()
15:         if c type is branch then
16:             S_ctx ← pop_context()
17:             I ← constraint_solver(S_ctx + ¬c)
18:             if I is valid then
19:                 τ ← position of c in π
20:                 break // execute with new input
21:             end if
22:         end if
23:     end while
24: until I is not valid
25: return T
```

$build\_stack(\pi, \tau, S_{ctx})$

```
1:  Constraints stack, C ← ∅
2:  for i = τ to π.end do
3:      if π[i] is of branch type then
4:          push_context(S_ctx) // Save context for reuse
5:      end if
6:      C.push(π[i])
7:      add_assertion(S_ctx, π[i])
8:  end for
9:  return C
```

VI. Experiments

A. Experimental Setup

Experiments are carried out on a machine with Intel Core i7 6700k processor and 32GB RAM. Concolic testing framework is implemented using C++. Open-source Icarus Verilog target API is used for parsing, elaborating and flattening of Verilog RTL [22]. It removes some of the structurally unreachable branches during elaboration. Icarus Verilog is also used for simulation. Yices 2.5.1 SMT solver is used as the constraint solver [19]. We have evaluated our approach on several RTL benchmarks from ITC99 [23], OpenCore [24] and TrustHUB [25]. Most of these benchmarks contain hard to reach branches. As QUEBS is not exhaustive, coverage and runtimes might vary depending on the initial random input. All values reported in this paper are average of 10 simulations with different starting inputs.

B. Branch Coverage Evaluation

We have evaluated our approach against CGS [13], HYBRO [11] and [16]. CGS is one of the most prominent search heuristics in software domain. As demonstrated in [13], CGS outperforms other heuristics such as random, CarFast, generational, CFG-directed etc. However, being a software domain concolic testing method, CGS does not directly work on RTL benchmarks. For this reason, we have implemented CGS for comparison. On the other hand, HYBRO is a concolic testing platform for RTL designs. It uses branch coverage guided bounded DFS as search strategy. The last method we will be comparing against is proposed recently by Qin et al. [16], which uses a modified bounded DFS strategy.

Table I presents the results of branch coverage evaluation. The third, fifth, seventh, and ninth columns show the branch coverage obtained by CGS [13], HYBRO [11], Qin et al. [16], and our approach (QUEBS), respectively. We used the limit of 1 in QUEBS. The fourth, sixth, eighth and tenth columns indicate the runtime of the respective test generation methods. As it can be seen, coverage of QUEBS exceeds other approaches in most cases. [16] reports slightly higher coverage for b11 and b14, and HYBRO gives higher coverage for or1200 DCache. However, both required significantly more runtime. Overall, QUEBS provided high coverage (*avg.* 97.39%) with small runtime (*avg.* 2.77s) - demonstrating its applicability on a wide variety of designs.

C. Effect of Optimizations

1) Effect of Unsolvable Branch Elimination: Figure 5 shows number of branches before and after applying this pruning technique on seven benchmarks. On average, this method reduced effective branches by 30.62%. While some benchmarks like b14 have limited reduction in number of branches, it can lead to exponential reduction in runtime and solver calls. This will be more clear from Table II, which provides a comparison between QUEBS with and without branch elimination technique. Here, column 3 and 5 show the number of unsatisfiable calls to constraint solver - with and without applying branch elimination. Column 4 and 6 show corresponding runtime. Last two columns give the improvement factor.

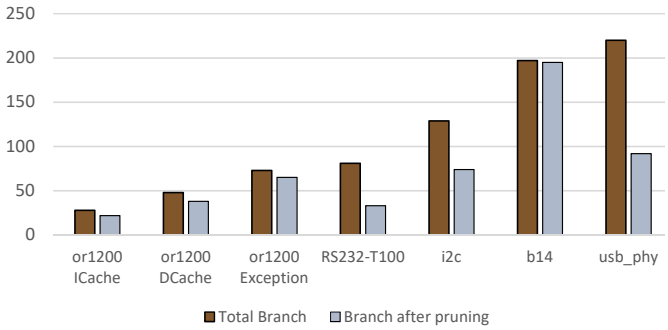

Fig. 5. Number of selectable branches before (left column) and after (right column) applying unsolvable branch elimination optimization.

As we can see, both unsatisfiable calls and runtimes are greatly benefited from this optimization. For b14 benchmark, which has the least branch elimination of only 1%, number of unsatisfiable calls to solver changes from over 20k to less than 2k. This effect is most prominent for RS232-T100 benchmark, where it reduced unsatisfiable solver calls from 26k to only 2. Similar improvement can be observed for runtimes also. Effectiveness of branch elimination technique depends on the rarity of eliminated branches. If these branches occur abundantly in execution paths, elimination will be much more effective than if they were rare.

2) Effect of Context Reuse: Table III presents the effectiveness of context reuse optimization. This table provides comparison before and after applying context reuse optimization on top of branch elimination. Column 3 and 5 gives total number of constraints given to solver over all iterations - before and after applying this optimization. This number is affected by both inter- and intra-simulation context reuse. Overall improvement factor is shown in last two columns. Column 6 shows context recovery percentage. Low recovery means that context between consecutive simulations are same most of the time. Thus low recovery increases inter-simulation context reuse. RS232-T100 benchmark have very high recovery rate, indicating that it does not benefit much from inter-simulation context reuse. For this benchmark, improvement in number of constraints comes mostly from intra-simulation. However, overhead of high recovery lead to worse runtime in this case. Selectively applying only intra-simulation context reuse for such designs with high recovery percentage might lead to better performance. Other benchmarks, even the ones with moderate recovery percentage, have profited from context reuse.

VII. Conclusion

We have proposed a novel search heuristic for concolic testing based test generation engines. Our search heuristic (QUEBS) prevents path explosion by limiting the number of times a branch can be selected. It also ensures high coverage by relaxing the limit whenever a qualifying event occurs. Overall, it combines the advantages of both exhaustive and restrictive approaches. We also presented two optimization

TABLE I
COVERAGE AND RUNTIME OF CONCOLIC TESTING METHODS

Benchmark	Unroll cycles	CGS, k=5 [13]		HYBRO [11]		Qin et al. [16]		QUEBS	
		Bran_cov	Time(s)	Bran_Cov	Time(s)	Bran_cov	Time (s)	Bran_cov	Time (s)
b01	10	**100.00%**	0.01	94.44%	0.07	96.30%	0.55	**100.00%**	0.01
b06	10	**100.00%**	0.02	94.12%	0.10	96.30%	0.46	**100.00%**	0.01
b10	10	93.02%	0.03	87.10%	4.56	-	-	**100.00%**	0.02
	30	**100.00%**	0.07	96.77%	52.14	96.67%	24.61	**100.00%**	0.14
	50	**100.00%**	0.18	96.77%	180.42	-	-	**100.00%**	0.38
b11	10	40.00%	0.01	78.26%	0.28	81.82%	0.67	**91.43%**	0.01
	50	91.43%	0.11	91.30%	326.85	**94.44%**	270.28	94.29%	0.19
	120	91.43%	0.78	-	-	-	-	**97.14%**	0.95
b14	15	95.38%	0.92	83.50%	301.69	**98.95%**	257.59	95.38%	0.98
or1200 I-Cache	50	89.29%	0.55	93.75%	37.73	-	-	**96.43%**	0.56
	100	89.29%	1.95	93.75%	191.82	-	-	**96.43%**	2.14
or1200 D-Cache	50	81.25%	9.13	**96.30%**	21.90	-	-	95.83%	3.88
	100	81.25%	64.16	**96.30%**	92.15	-	-	95.83%	23.05
or1200 Exception	10	98.63%	3.34	96.61%	287.62	-	-	**98.63%**	0.50

TABLE II
EFFECT OF UNSOLVABLE BRANCH ELIMINATION

Benchmark	Unroll cycles	Without optimization		With branch elimination		Improvement	
		Unsat	Time(s)	Unsat	Time(s)	Unsat	Time
b14	50	20877	18.81	1821	13.41	11.46	1.40
RS232-T100	100	17169	10.04	2	0.03	8584.50	334.67
	200	26228	41.13	2	0.34	13114.00	120.97
i2c	20	33194	13.55	9291	4.73	3.57	2.86
usb_phy	20	30349	11.68	2131	1.13	14.24	10.34
or1200 I-Cache	100	22040	8.44	1096	2.52	20.11	3.35
or1200 D-Cache	50	19857	29.42	2474	8.14	8.03	3.61
	100	84467	533.84	11122	113.35	7.59	4.71

TABLE III
EFFECT OF CONTEXT REUSE

Benchmark	Unroll cycles	With branch elimination		With branch elimination and context reuse			Improvement	
		Cnst	Time(s)	Cnst	Recovery	Time(s)	Cnst	Time
b14	50	4346k	13.41	267k	0.27%	8.71	16.28	1.54
RS232-T100	100	5k	0.03	4k	100%	0.05	1.25	0.60
	200	229k	0.34	215k	96.67%	0.40	1.07	0.85
i2c	20	7540k	4.73	120k	3.05%	1.00	62.83	4.73
usb_phy	20	3229k	1.13	172k	68.52%	0.32	18.77	3.53
or1200 I-Cache	100	1535k	2.52	585k	57.37%	2.14	2.62	1.18
or1200 D-Cache	50	2184k	8.14	445k	61.17%	3.88	4.91	2.10
	100	16130k	113.35	1829k	58.84%	23.05	8.82	4.92

techniques to further improve the performance of QUEBS. One of them is a static analysis to prune unsolvable branches. It statically detects branches for which constraint solver cannot generate satisfiable input set - which greatly reduces unsatisfiable solver calls. Another optimization is intra- and inter-simulation context reuse based on the similarity of execution path. Compared to state-of-the-art approaches, QUEBS provided higher coverage in most RTL benchmarks with significantly reduced runtime.

REFERENCES

[1] J. Bergeron, *Writing testbenches: functional verification of HDL models*. Springer Science & Business Media, 2012.
[2] X. Qin and P. Mishra, "Directed test generation for validation of multicore architectures," *TODAES*, vol. 17, p. 24, 2012.
[3] Chen et al., "Automatic rtl test generation from systemc tlm specifications," *TECS*, vol. 11, p. 38, 2012.
[4] F. Farahmandi and P. Mishra, "Automated test generation for debugging arithmetic circuits," in *DATE*, 2016, pp. 1351–1356.
[5] Farahmandi et al., "Exploiting transaction level models for observability-aware post-silicon test generation," in *DATE*, 2016, pp. 1477–1480.
[6] M. Chen et al., *System-level validation: high-level modeling and directed test generation techniques*. Springer Science & Business Media, 2012.
[7] K. Sen et al., "Cute: a concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, 2005, pp. 263–272.
[8] P. Godefroid et al., "Dart: directed automated random testing," in *ACM SIGPLAN Notices*, vol. 40, 2005, pp. 213–223.
[9] L. Liu and S. Vasudevan, "Star: Generating input vectors for design validation by static analysis of rtl," in *HLDVT*, 2009, pp. 32–37.
[10] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *ASE*, 2008, pp. 443–446.
[11] L. Liu and S. Vasudevan, "Efficient validation input generation in rtl by hybridized source code analysis," in *DATE*, 2011, pp. 1–6.
[12] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *CAV*, 2006, pp. 419–423.
[13] H. Seo and S. Kim, "How we get there: A context-guided search strategy in concolic testing," in *FSE*, 2014, pp. 413–424.
[14] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, pp. 82–90, 2013.
[15] L. Liu and S. Vasudevan, "Scaling input stimulus generation through hybrid static and dynamic analysis of rtl," *TODAES*, vol. 20, p. 4, 2014.
[16] X. Qin and P. Mishra, "Scalable test generation by interleaving concrete and symbolic execution," in *VLSID*, 2014, pp. 104–109.
[17] S. Park et al., "Carfast: Achieving higher statement coverage faster," in *FSE*, 2012, p. 35.
[18] Y. Li et al., "Steering symbolic execution to less traveled paths," in *ACM SIGPLAN Notices*, vol. 48, 2013, pp. 19–32.
[19] B. Dutertre, "Yices 2.2," in *CAV*, 2014, pp. 737–744.
[20] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, 2008, pp. 337–340.
[21] R. Brummayer and A. Biere, "Boolector: An efficient smt solver for bit-vectors and arrays," in *TACAS*, 2009, pp. 174–177.
[22] S. Williams, "Icarus verilog," *On-line: http://iverilog.icarus.com/*, 2006.
[23] F. Corno et al., "Rt-level itc'99 benchmarks and first atpg results," *IEEE Design & Test of computers*, vol. 17, no. 3, pp. 44–53, 2000.
[24] "Opencores website," *On-line: https://www.opencores.org*, 2017.
[25] "Trusthub website," *On-line: https://www.trust-hub.org*, 2017.