GreyConE: Greybox Fuzzing + Concolic Execution
Guided Test Generation for High Level Designs

Mukta Debnath†
Indian Statistical Institute
Kolkata, India
Email: mukta_t@isical.ac.in

Animesh Basak Chowdhury†
New York University
New York, USA
Email: abc586@nyu.edu

Debasri Saha
Calcutta University
Kolkata, India
Email: debasri_cu@yahoo.in

Susmita Sur-Kolay
Indian Statistical Institute
Kolkata, India
Email: ssk@isical.ac.in

Abstract—Exhaustive testing of high-level designs poses an arduous challenge due to complex branching conditions, loop structures, and the inherent concurrency of hardware designs. Test engineers aim to generate quality test cases satisfying various code coverage metrics to ensure minimal presence of bugs in a design. Prior works in testing SystemC designs are time inefficient which obstructs achieving the desired coverage in a shorter timespan. We interleave greybox fuzzing and concolic execution in a systematic manner and generate quality test cases for accelerating test coverage metrics. Our results outperform state-of-the-art methods in terms of number of test cases and branch-coverage for some of the benchmarks, and runtime for most of them.

Index Terms—SystemC, High-level Synthesis, Greybox fuzzing, Symbolic Execution, Concolic Execution, Code Coverage

I. INTRODUCTION

SYSTEMC is the latest de-facto standard for early-stage development of hardware designs having a complex design architecture. Detecting functional and security bugs at early stage design development is crucial to eliminate functional inconsistencies and vulnerabilities. A SystemC design provides cycle-accurate behavioural modeling of a given specification. Therefore, testing SystemC design focuses more on functional level correctness, inconsistency in specifications and unknown (possibly erroneous) behaviour rather than defects arising from low-level hardware. Hardware engineers prefer to check logical inconsistencies in early-stage of design as defects caught post manufacturing stage are costly. This motivates verification and test engineers to develop a good test generation framework for detecting bugs for designs at higher abstraction level.

Verification and testing of SystemC designs (and in general, high level designs) for bug detection have a rich literature [1], [2], [3], [4]. Prior works adopt two mainstream approaches to tackle the problem: (a) formal techniques, e.g. model checking [2] and symbolic execution [3]; and (b) simulation-based techniques such as black-box testing, white-box [5], [6] and grey-box [7], [8] fuzzing. There are a plethora of works that have critically identified the issues of scalability and exhaustive testing, and addressed them by proposing a hybrid approach – concolic testing. A very recent work [9] has applied concolic testing to alleviate the problem of scalability arising from symbolic execution. Although the approach is quite promising, we ask the following two unaddressed questions:

1) Concolic testing relies heavily on initial test cases. How can one generate better quality initial test cases (in a minimalistic time frame) such that concolic execution can focus only on “hard-to-reach” state of design?

2) Guided fuzzing-based techniques explore the design space quickly but often get stuck in complex branch conditions. How can one mitigate this problem in fuzzing?

In this work, we propose GreyConE, an end-to-end test-generation framework for high-level designs to generate high-quality test cases and quickly achieve better coverage metrics. In short, we make the following contributions:

1) We developed GreyConE which leverages the power of greybox fuzzing by quickly covering "easy-to-reach" states first and passing on “hard-to-reach” state(s) to concolic engine in an alternating manner.

2) GreyConE outperforms state-of-the-art Greybox fuzzing (AFL [7]) and concolic test generation (S2E [10]) in terms of branch coverage achieved and runtime for many of the benchmark SystemC designs.

The rest of the paper is organized as follows: Section II outlines the background and prior related works. In Section III, we present our proposed framework and show the efficacy of results in Section IV. Section VI has the concluding remarks.

II. BACKGROUND

A. Test-case generation for SystemC Designs

Recent works such as SESC [11] and CTSC [9] use symbolic and concolic execution respectively to test SystemC designs. SESC adapted symbolic execution with significant code coverage but was limited to only a subset of SystemC features. Later, CTSC covered most SystemC semantics but failed to improve efficacy in terms of performance and scalability compared to SESC. Further, both the works depend on the availability of initial test cases for the symbolic engine.

B. Greybox fuzzing

Fuzz testing is a well-known technique for detecting bugs in software. Greybox fuzzing [7], [12] generates interesting test cases using genetic algorithm on instrumented executables. Instrumentation injects markers in the code at every basic-block which can track post compilation and execution whether a test-case has reached the marker location. A fitness function
C. Symbolic and Concolic Execution

Symbolic execution is a formal technique for generating quality test cases. It typically assigns program inputs as symbolic values (as opposed to concrete values) and forks two threads at each condition; a boolean formula evaluating to true and its negation to false. Thus, symbolic execution forms an execution tree (Figure 1(b)). At each branching condition, a SAT/SMT solver is invoked to generate a test-case which finds a set of concrete values assigned to the symbolic variables reaching the branching condition. The size of the execution tree grows exponentially in terms of branching conditions resulting in state-space explosion.

One major drawback of symbolic execution is treating all program inputs as symbolic variables. This results in complex boolean formulae within a few nested conditions resulting in longer runtime. Concolic execution solves this by assigning concrete values to a set of program inputs for quicker search space exploration. This helps covering the branching conditions which are “easy-to-reach”, and pass on the simplified formulae for “hard-to-reach” conditions to SAT/SMT solver. Post executing the program with concrete values, constraints at each condition are negated and solved to generate a new test case covering the unexplored path. The path constraints generated have reduced the number of clauses and variables aiding solvers to penetrate deeper program states quickly.

We demonstrate concolic execution with a simple example (Figure 1a). A set of program inputs are treated as symbolic variables; the inputs i and j have symbolic values i = a and j = b. We choose a random concrete input (i = 2, j = 1) and obtain the execution trace (dotted lines, Figure 1b). The alternative path constraints to be explored symbolically are collected along the execution path guided by concrete inputs, forking the side branches. At each condition, constraints are negated and solved to generate new test cases. The concolic engine terminates after all conditions are covered. In the next section, we present Greycone, which interleaves concolic execution and greybox fuzzing to accelerate test-case generation.

Fig. 1: (a) Example code snippet. (b) Symbolic and concolic execution flow

III. PROPOSED GreyConE FRAMEWORK

GreyConE (Figure 2) comprises: (1) Greybox fuzzing, (2) Concolic execution, and 3) Coverage evaluator.

A. Generating instrumented binary

First, we convert a high-level hardware design to a low-level intermediate representation (IR) in the form of LLVM bitcode [13]. The compiler inserts a marker at the top of every basic block in the IR to generate an instrumented executable which is passed on to both greybox fuzzing engine and concolic engine to generate test cases and track uncovered basic blocks.

B. Greybox fuzzing by AFL

We outline the overall flow of Greybox fuzzing by American fuzzy lop (AFL) [7] in Algorithm 1. The instrumented executable DUT$_{ins-exec}$ and a test-set T$_{initial}$ are fed to the fuzzing framework. The function CALCULATE-ENERGY assigns energy to every test-case of T$_{initial}$ based on runtime behavior, e.g. execution time and obtained coverage. It assigns more energy to a test-case with faster execution time, covering more branches and penetrating deeper code segments. AFL uses T$_{initial}$ to perform deterministic and havoc mutations to generate newer test cases (MUTATE-SEED). AFL uses branch-pair coverage as a fitness metric to determine the quality of a test-input. For each branch-pair, it maintains a hashtable entry to record hit counts. It retains a test-case for further exploration if it covers an unseen new-branch pair, or has unique hit-counts on an already covered branch-pair (IS-INTERESTING). The fuzz engine retains interesting test cases for further exploration. The algorithm terminates after either covering all the branching conditions or reaching a user-defined time cutoff. AFL maintains all interesting test cases in the queue T$_{fuzzed}$.
C. Concolic testing by S2E

For concolic testing, we employ symbolic executor (S2E) [10] which has two main components: (a) a virtual machine based on QEMU [14] and (b) a symbolic execution engine based on KLEE [15]. The interesting inputs in the design can be marked as symbolic using an S2E API function. The CONC-EXEC executes the DUT with test cases from $T_{\text{initial}}$ to generate concrete execution traces in $DUT_{\text{execTree}}$, and S2E identifies the uncovered branch-pairs. The $\text{COND-PREDICATE}$ constructs the path constraints for the uncovered edge of a condition from $DUT_{\text{execTree}}$, forks a new thread and invokes SAT-solver ($\text{CONSTRAINT-SOLVER}$) to generate a test-case. The concolic engine terminates either after a user-defined timeout is reached or covers all branch-pairs of $DUT_{\text{execTree}}$. The test cases generated by the concolic engine are stored in the queue $T_{\text{concolic}}$. We outline a typical concolic execution approach in Algorithm 2.

D. Interleaved fuzzing and concolic testing

We interleave greybox fuzzing and concolic execution to extenuate the problems associated with their standalone modes. Concolic engine usually performs depth-first search for search-space exploration. A set of random test cases leads to invoking the SAT/SMT solver frequently for generating test cases to cover unexplored conditions in $DUT_{\text{execTree}}$. We perform fuzzing of the DUT to avoid this huge slow-down and invoke concolic engine for “hard-to-cover” scenarios with the fuzzer generated test cases. As shown in Figure 2, we first perform lightweight instrumentation on all conditions of design-under-test (DUT) and generate an instrumented executable. We start our fuzz-engine (FUZZER) with a set of initial test cases. The fuzz-engine generates interesting test cases using genetic algorithm and explores various paths in the design. Once branch-pair coverage stops growing in a user-defined time period $time_{\text{cutoff}}$, we invoke the concolic engine ($\text{CONCOL-EXEC}$) with fuzzed test cases. $\text{CONCOL-EXEC}$ identifies uncovered conditions in $DUT_{\text{execTree}}$ and forks new threads for symbolic execution on such conditions using depth-first search. The concolic engine generates new test cases satisfying complex conditional statements. We limit the runtime of concolic execution engine to avoid scalability bottleneck by $time_{\text{cutoff}}$, which monitors the time elapsed since the last test case was generated. Test cases generated by concolic engine are fed back to the fuzz engine for quicker exploration once the “hard-to-cover” conditions are explored. This process halts when either a user-defined target coverage is achieved or a user-defined time limit $time_{\text{cutoff}}$ is reached.

![Figure 2: GreyConE test generation framework](image-url)

**Algorithm 2: CONCOL-EXEC($DUT_{\text{ins-exec}}, T_{\text{initial}}$)**

- **Data:** $DUT_{\text{ins-exec}}, T_{\text{initial}}$
- **Result:** $T_{\text{concolic}}$
- **Variables:**
  - $\phi$: Path predicate
  - $\tau$: Test-case
  - $T_{\text{execTree}}$: Test-case tree
  - $T_{\text{initial}}$: Initial test case


code:

```
for $\tau \in T_{\text{initial}}$ do
    $Prace \leftarrow \text{CONCOL-EXEC}(DUT_{\text{ins-exec}}, \tau)$
    $DUT_{\text{execTree}} \leftarrow DUT_{\text{execTree}} \cup Prace$
end

for uncovered cond $c \in DUT_{\text{execTree}}$ do
    $pc \leftarrow \text{COND-PREDICATE}(c)$
    $t_i \leftarrow \text{CONSTRAINT-SOLVER}(pc)$
    $T_{\text{concolic}} \leftarrow T_{\text{concolic}} \cup t_i$
end
return $T_{\text{concolic}}$
```

496
TABLE I: GreyConE execution phases

| Benchmarks LOC | # Test cases | Branch cov. (%) \(f_{\text{con}}\) | Time (in s) |
|----------------|-------------|---------------------------------|-------------|
| ADPCM          | 270         | 5 6 -                            | 93.3 100    | 9 39         |
| AES            | 429         | 3 4 4                            | 79.2 83.3 91.7 | 76 29 50    |
| FFT_fixed      | 334         | 3 3 -                            | 81.2 96.9   | 29 137      |
| IDCT           | 450         | 62 137 -                         | 64.8 100    | 7 222       |
| MDSC           | 467         | 5 3 2                            | 87.5 90.6 100 | 9 34 7     |
| Filter_FIR     | 176         | 3 2 7                            | 76.8 87.5 93.8 | 7 23 37    |
| Interpolation   | 231         | 44 -                             | 100           | 3 -         |
| Decimation     | 422         | 3 2 -                            | 96.8 100    | 12 22       |
| Kasumi         | 415         | 31 23 -                          | 93.3 100    | 10 46       |
| UART           | 160         | 2 15 4                           | 81.2 84.1 88.5 | 334 136 248 |
| Quick_sort     | 204         | 7 -                              | 100          | -           |

IV. EXPERIMENTAL SETUP AND DESIGN

A. Experimental setup

We implement GreyConE with state-of-the-art software testing tools: AFL (v.2.52b)[7] and S2E[10]. For robust coverage measurements, we cross-validated our results with coverage measurement tools: lcov-1.13 [16], and gcov-7.5.0 [17]. Experiments were on a 3.20 GHz 16 GB RAM i5 linux machine.

B. Benchmark characteristics

We evaluate GreyConE on a wide spectrum of available SystemC benchmarks: SCBench[18] and S2CBench[19], selected from a variety of application domains covering many open-source hardware designs. The benchmarks considered have diverse characterization: ADPCM, FFT, IDCT (all image processing cores), AES, MDSC (cryptographic cores), Quick_sort (data manipulation), Decimation (filters) and UART (communication protocols).

C. Design of experiments

We evaluate the efficacy of GreyConE in two aspects: (a) coverage improvement and (b) run-time speedup, and compare our results with the two state-of-the-art testing techniques – (i) AFL and (ii) S2E.

Baseline 1 (Fuzz testing): We run AFL on the above-mentioned SystemC benchmarks using default algorithmic settings. Initial seed inputs are generated randomly.

Baseline 2 (Concolic execution): Similarly, we run S2E on the SystemC benchmarks with default configurations and randomly generated concrete seed inputs.

GreyConE: We run GreyConE on the SystemC benchmarks starting with randomly generated input test cases. \(t_{\text{threshold}}\) and \(t_{\text{threshold}}\) set as 5s and 10s respectively, excluding time to generate the first seed for each engine.

For our experiments, we set \(t_{\text{cutoff}}\) to 2 hours and compare baseline methods and GreyConE. Next, we discuss the performance of GreyConE in terms of branch-coverage improvement and run-time speedup.

V. RESULTS AND EVALUATION

GreyConE invokes a fuzz-engine and a concolic engine interchangeably. We have annotated each phase of run incrementally, where \(f_{\text{con}}\) denotes the \(k\)th execution phase of fuzz-engine. In order to show the effectiveness of GreyConE, we present branch coverage achieved in each phase along with the number of test cases generated in Table I. In Table II, we report the number of test cases generated, achievable branch coverage within \(t_{\text{cutoff}}\) and the earliest time taken to reach that coverage. We compare our results with AFL [7] and S2E [10] which are open-sourced implementations. Due to unavailability of implementations of SESC [11] and CTSC [9], we used S2E [10] to demonstrate the performance of concolic execution on SystemC designs by adopting necessary changes.

1) Coverage achieved by GreyConE: Higher branch coverage indicates greater probability of test cases detecting bugs hidden in deeper program segments. As depicted in Table I, for certain designs, GreyConE does not require concolic engine at all. This indicates that such benchmarks lack complex “branching” conditions where fuzzer can get stuck. For every design, GreyConE achieves maximum possible coverage with one call to concolic engine. We claim that GreyConE achieved “maximum” coverage as we independently validated that “uncovered” branches are unreachable codes in GreyConE for certain designs, does not require concolic engine at all. This indicates that such benchmarks lack complex “branching” conditions where fuzzer can get stuck. For every design, GreyConE achieves maximum possible coverage with one call to concolic engine. We claim that GreyConE achieved “maximum” coverage as we independently validated that “uncovered” branches are unreachable codes in AES, FFT_fixed, Filter_FIR and UART. We compare the branch coverage obtained by GreyConE with other techniques in Table II and observe that it outperforms baseline techniques AFL and S2E in terms of coverage achieved (Figure 3). As illustrated, the test cases produced by GreyConE significantly improves the branch coverage by 3%-25.9% compared to AFL and 2.8%-30% compared to S2E within the two hours’ time limit. Figure 4 provides a detailed coverage analysis over the entire time period of two hours.

2) Analyzing run-time speedup: We measure the time taken to obtain the best achievable coverage by baseline techniques
and compare run-time speedup of GreyConE to achieve the same. From Figure 4, we observe that the time taken to achieve a certain branch coverage is lower bounded by GreyConE compared to AFL and S2E. In Figure 3, we show the runtime speedup of GreyConE over AFL and S2E for the designs where every technique has achieved the same branch coverage within the time-limit. We observe that GreyConE is significantly faster to reach a certain branch coverage. The speed-up achieved by GreyConE are in line with our design approach: (1) GreyConE quickly identifies the region where fuzz engine gets stuck and invoke concolic engine to solve the complex conditions; (2) GreyConE avoids expensive path exploration by concolic execution by using fuzzer generated seeds leading to faster exploration and test-case generation.

3) Analyzing test-case quality: We report in Table II the number of test cases preserved by each technique until it reaches the user-defined timeout, or till the maximum coverage is achieved. A closer analysis reveals that the number of test cases generated by GreyConE is the same as by AFL where AFL alone sufficed in reaching the target coverage without getting stuck within the time $t_{threshold}$. But, for cases where AFL crossed $t_{threshold}$, GreyConE invokes concolic engine for generating quality test cases. Similarly, when S2E gets stuck for $t_{threshold}$, GreyConE invokes the fuzz engine. Finally, GreyConE needs fewer test cases than both AFL and S2E indicating good quality test-case generation.

VI. CONCLUSION

We have proposed GreyConE here an end-to-end test-generation framework penetrating into deeper program segments of SystemC designs. Our results show scalable generation of test cases with better branch coverage and accelerated design space exploration compared to state-of-the-art techniques. GreyConE has alleviated the drawbacks of fuzzing and concolic execution by interleaving them systematically.

Future works may include enhancing GreyConE to low-level netlist (RTL/gate-level) and uncover hardware specific bugs in the designs.

REFERENCES

[1] D. Karlsson et al., “Formal verification of systemc designs using a petrinet based representation,” in Proc. DATE, 2006, pp. 1228–1233.
[2] P. Herber et al., “Model checking systemc designs using timed automata,” in Proc. CODES+ISSS, 2008, pp. 131–136.
[3] A. Vafaee et al., “Symba: Symbolic execution at c-level for hardware trojan activation,” in Proc. ITC, 2021, pp. 223–232.
[4] V. Herdt et al., “Verifying systemc using stateful symbolic simulation,” in Proc. DAC, 2015, pp. 1–6.
[5] P. Godefroid et al., “Dart: Directed automated random testing,” in Proc. PLDI, 2005, pp. 211–223.
[6] P. Godefroid, “SAGE: Whitebox fuzzing for security testing,” Queue, vol. 10, pp. 1–8, 2012.
[7] M. Zalewski, “American fuzzy lop,” http://lcamtuf.coredump.cx/afl.
[8] T. Trippel et al., “Fuzzing hardware like software,” ArXiv, vol. abs/2102.02308, 2021.
[9] B. Lin et al., “Concolic testing of systemc designs,” in Proc. ISQED, 2018, pp. 1–7.
[10] V. Chipounov et al., “The S2E platform: Design, implementation, and applications,” ACM TCS, vol. 30, pp. 2:1–2:49, 2012.
[11] B. Lin et al., “Generating high coverage tests for SystemC designs using symbolic execution,” in ASPDAC, 2016, pp. 166–171.
[12] A. B. Chowdhury et al., “Verifuzz: Program aware fuzzing,” in Proc. TACAS, 2019, pp. 244–249.
[13] “LLVM,” https://llvm.org/.
[14] F. Bellard, “QEMU, a fast and portable dynamic translator,” in Proc. USENIX FREENIX, 2005, pp. 41–46.
[15] C. Cadar et al., “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs,” in Proc. OSDI, 2008, pp. 209–224.
[16] “LCOV,” http://ltp.sourceforge.net/covtool/lcov.php.
[17] “Gcov,” https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[18] B. Lin et al., “SCBench: A benchmark design suite for systemc verification and validation,” in Proc. ASPDAC, 2018, pp. 440–445.
[19] B. C. Schafer et al., “S2CBench: Synthesizable systemc benchmark suite for high-level synthesis,” IEEE ESL, vol. 6, pp. 53–56, 2014.