High Coverage Concolic Equivalence Checking

Pritam Roy
pritam_roy@mentor.com

Sagar Chaki
sagar_chaki@mentor.com

Pankaj Chauhan
pankaj_chauhan2@mentor.com

Abstract*—A concolic approach, called SLEC-CF, to check sequential equivalence between a high-level (e.g., C++/SystemC) hardware description and an RTL (e.g., Verilog) is presented. SLEC-CF searches for counterexamples over the possible values of a set of "control signals" in a depth-first lexicographic manner, avoiding values that are unrealizable by any concrete input. In addition, SLEC-CF respects user-specified design constraints during search, thus only producing stimuli that are of relevance to users. It is a superior alternative to random simulations, which produce an overwhelming number of irrelevant stimuli for user-constrained designs, and are therefore of limited effectiveness. To handle complex designs, we present an incremental version of SLEC-CF, which iteratively increases the search depth, and set of control signals, and uses a cache to reuse prior results. We implemented SLEC-CF on top an existing industrial tool for sequential equivalence checking. Experimental results indicate that SLEC-CF clearly outperforms random simulation in terms of coverage achieved. On complex designs, incremental SLEC-CF demonstrates superior ability to achieve good coverage in almost all cases, compared to non-incremental SLEC-CF.

I. Introduction

Hardware equivalence checking [1] has emerged as one of the most promising areas of formal verification in practical settings. With the increasing complexity of hardware designs, there is a clear trend toward high-level synthesis (HLS) [2] – designing hardware using a high-level language (such as C++/SystemC[1]) and compiling it down to RTL (such as Verilog). In this context, equivalence checking has assumed even greater urgency. The synthesizer performs non-trivial optimizations and transformations over large designs, and it is critical that we assure that it produces functionally equivalent RTL. A number of tools [3] have emerged to tackle this problem.

In practice, equivalence checking is often applied to a large system on a per-module basis. Since a module will eventually operate in the context of other modules (its environment) in the final system, its verification must reflect the appropriate environment constraints. This is typically specified by a set of assumptions on legal values of "internal" module variables (such as branch conditions or intermediate results) rather than module "inputs". To be effective, verification techniques must also respect these environment constraints. In particular, any technique that generates a large number of "invalid" stimuli (which violate environment constraints) for inspection is quickly discarded by users as ineffective.

Random testing (fuzzing) [4] is the most prevalent technique used today for generating interesting stimuli for inspection by hardware developers. However, it not only has poor coverage for complex designs, but also ignores environment constraints and deluges the user with invalid stimuli. This work was inspired by the need for a better alternative which achieves not only high coverage, but also produces only high-quality valid stimuli. Concolic testing has emerged in recent years as a promising direction for generating testcases with high coverage. Taking this cue, we propose an approach, called SLEC-CF, which to our knowledge is the first to use concolic exploration for producing valid stimuli in hardware equivalence checking. Specifically, we make three contributions.

First, we present SLEC-CF formally. SLEC-CF uses the set of all branch conditions (in the C++) and MUX-select signals (in the RTL) as control signals, rather than the inputs. This is intended to produce higher-quality stimuli since branch conditions and MUX-selects represent design corner cases where bugs typically lurk. The core of SLEC-CF is a lexicographic exploration of all control signal (CS) values, using a depth-first search (DFS) strategy. Constraint solving is used to guide the DFS, so that we never explore (unrealizable) values which cannot be reached by any concrete execution. SLEC-CF outputs all encountered equivalence-falsifying input values (i.e., stimuli) that lead to different outputs between the C++ and the RTL. We show how to incorporate environment constraints into SLEC-CF so that only valid stimuli are generated. We also define a suitable notion of coverage that measures the fraction of CS values explored by SLEC-CF, taking into account both the realizable and unrealizable ones.

Second, we present an incremental version of SLEC-CF that iteratively increases the search depth, and the CS set. We also present a caching mechanism to improve the scalability of this incremental approach. In essence, the cache stores results from constraint solver calls that indicate whether a specific CS assignment is realizable. The cache is not a direct lookup since the prior solver invocations involve either a different search depth, or a smaller CS. In addition, the cache works asymmetrically for SAT and UNSAT answers from the solver. For example, if a CS assignment is unrealizable, then so are all its "extensions", but not necessarily all its "restrictions". For realizable assignments, the situation is reversed. We use two caches (for SAT and UNSAT answers) that leverage this idea, using an appropriate notion of extensions and restrictions for CS assignments, as the search depth and CS size changes.

Finally, we implement SLEC-CF on top of an existing HLS and equivalence checking tool, and evaluate it against random testing using a benchmark derived from industrial examples. Our results indicate that SLEC-CF provides orders

[1]We'll henceforth use C++ to mean both.

of magnitude better coverage than random simulation. On a set of 100 hard industrial testcases, SLEC-CF achieves full coverage on 16 testcases, and at least 10% coverage on 78 testcases, while random simulation cannot achieve this level of coverage on even one testcase. Also, SLEC-CF finds CEXs in 10x more examples than random simulation. Since incremental SLEC-CF restarts the search whenever the depth or CS set is changed, it performs some level of redundant work. Therefore on simpler testcases, non-incremental SLEC-CF performs better. However, for harder testcases, incremental SLEC-CF achieves much better coverage. The combination of the two SLEC-CF variants therefore provides a good strategy for addressing a wide range of equivalence checking scenarios.

Related Work. Concolic testing [5] aims to improve testcase generation for software. It combines symbolic simulation with constraint solving to generate testcases that provide better coverage than random fuzzing [4], while being more tractable than MC/DC testing [6]. It has been applied in both industrial [7] and academic [8], [9] settings. We use the key ideas behind concolic testing, but: (i) target hardware equivalence checking, and (ii) develop an incremental version.

Several projects have applied concolic verification to hardware testing. Liu and Vasudevan [10] propose the STAR technique for concolic testcase generation of RTL. They use static analysis of the RTL to mine conditions which can be "flipped" during concolic exploration. They also propose the HYBRO [11] technique, where a branch coverage metric is used to decide the next condition to be flipped. Their goal is increased test coverage, not verification of a specific property. Their techniques are complementary to ours. In addition, we target hardware equivalence checking, not testing.

Zhang et al. [12] propose to generate testcases for RTL by first compiling it to C++, and then using concolic tester that supports C++. They implement an RTL-to-C++ translator to perform the first step. For the second step, they augment KLEE to support bit-level operations, which are more critical for hardware analysis. Our approach works directly on RTL, without translating it to C++. Ahmed and Mishra [13] propose a technique, called QUEBS, for concolic RTL validation. Their key contribution is to first limit the set of branches that must be flipped, and then gradually relax the set as they are visited by testcases. This is a complementary way of managing the intractability of concolic analysis. Pinto and Hsiao [14] propose an approach, called CORT, for functional testcase generation for RTL. They first convert the RTL to instrumented C++, which is then compiled to binary. The binary is analyzed cycle-by-cycle. In each cycle, testcases are generated to explore different control-flow-paths. Our approach does not involve translation of RTL to C++, or cycle-by-cycle analysis, and is therefore complementary. All these techniques target RTL testing, while our focus is on improving coverage during equivalence checking between C++ and RTL.

There is considerable research on symbolic simulation [15], bounded model checking (BMC) [16] and full model checking for hardware verification and hardware equivalence checking [1], [3]. To our knowledge, SLEC-CF is the first use of concolic reasoning to address this challenge. Compared to classic equivalence checking, our aim is to maximize coverage while looking for falsifying stimuli, not full verification. Compared to symbolic simulation and BMC, concolic reasoning aims to provide better tractability by: (i) reducing the search statespace from all inputs to a smaller set of control signals; and (ii) allowing incremental analysis on a path-by-path basis. Concolic thus forms an early part of the layered strategy of using increasingly expensive verification techniques.

II. Background

Consider a finite set of Boolean variables V. Given any subset $X \subseteq V$ of variables, we write 2^X to mean the set of all assignments X. A sequential circuit is a tuple (Q, I, O, R) where: (i) $Q \subseteq V$ is the set of state variables; (ii) $I \subseteq V$ is the set of inputs; (iii) $O \subseteq V$ is the set of output variables; and $R : 2^Q \times 2^I \mapsto 2^Q \times 2^O$ is the transition "function" that maps a current state and input to the next state and output. Note that we consider deterministic circuits only. We assume a distinguished initial state $q_0 \in 2^Q$. An input sequence (IS) $\sigma = (i_1, i_2, \ldots, i_n)$ produces the output o, denoted $C(\sigma) = o$, iff there exists an intermediate sequence of states $q_1, \ldots, q_n$, and outputs $o_1, \ldots, o_n$ such that $o = o_n$ and:

$$\forall k \in [1, n] \centerdot R(q_{k-1}, i_k) = (q_k, o_k)$$

We are interested in checking (output) equivalence of circuits up to N steps, where N is a known positive integer. We say that the two circuits $C_1 = (Q_1, I, O_1, R_1)$ and $C_2 = (Q_2, I, O_2, R_2)$ with the same input variables are equivalent up to N steps, denoted $C_1 \equiv_N C_2$, iff for all input sequences σ of length no more than N, we have $C_1(\sigma) = C_2(\sigma)$. The counterexample (CEX) to equivalence is thus an input sequence σ such that $|\sigma| \leq N \wedge C_1(\sigma) \neq C_2(\sigma)$.

Consider the following example to demonstrate the kind of corner cases we aim to target using concolic verification. While the example is that of a single circuit, similar corner cases arise when two circuits are to be compared.

Example. The Verilog code in Figure 1 models the tautology: $\min(x1, x2) = -(\max(-x1, -x2))$. If it were correct, then the two outputs `o1` and `o2` would always be equal for all inputs `x1` and `x2`. However, it is buggy because of numeric overflow during negation (at lines 9 and 10). The program has two MUX selectors – `s1` and `s2`. When analyzing this program using `s1` and `s2` as control signals, SLEC-CF finds two CEXs – `(x1=0x0b, x2=0x20)` and `(x1=0x20, x2=0x08)`. Note that the two CEXs correspond to precisely the corner cases that indicate the bug, i.e., when one of the inputs is the "most negative" 6-bit number.

III. Concolic Verification

We use concolic verification to implement a new engine, called SLEC-CF, for checking $C_1 \equiv_N C_2$. We assume that C_1 and C_2 are defined at the Register-Transfer Level (RTL), e.g., using Verilog. We denote by *Sig* the set of selector signals of MUXes appearing in C_1 and C_2. SLEC-CF is parameterized by the search depth N, and a set of "control

```
module split(o1,o2,x1,x2);
   input wire signed [5:0] x1,x2;
   output wire [0:0] o1, o2;
   wire [0:0] s1,s2;
   assign s1 = (x1 < x2);
   wire signed [5:0] minVal, maxNeg;
   assign minVal = s1 ? x1 : x2;
   wire signed [5:0] neg1, neg2;
   assign neg1 = -x1;
   assign neg2 = -x2;
   assign s2 = (neg1 > neg2);
   assign maxNeg = s2 ? neg1 : neg2;
   assign o1 = (minVal == -maxNeg);
   assign o2 = 1;
endmodule
```

Fig. 1. Motivating Example

```
Algorithm SLEC-CF ($S$, $N$):
  1) Start with a random input sequence $\sigma$ of length $N$.
  2) For each prefix $\tilde{\sigma}$ of $\sigma$:
     a) Compute $(o_1, o_2) = \text{SIMUL}(\tilde{\sigma})$.
     b) If $o_1 \neq o_2$, then report $\tilde{\sigma}$ as a CEX, and exit.
  3) Invoke CEXGEN ($S$, $N$, $\sigma$).
  4) If CEXGEN returns "no more IS" then report "no
     CEX found" and exit.
  5) Otherwise set $\sigma$ to the input sequence returned by
     CEXGEN , and repeat from Step 2.
```

Fig. 2. Overall SLEC-CF algorithm.

signals" $S \subseteq Sig$. In essence, it explores the set of values of S systematically, looking for CEXs. It skips over values of S that are unrealizable, i.e., those which cannot be realized by any concrete execution of the circuits. SLEC-CF uses two main components – a simulator SIMUL and a IS generator CEXGEN – which behave as follows: (i) given an IS σ, SIMUL produces the output pair $(C_1(\sigma), C_2(\sigma))$; (ii) given an IS σ of length N, CEXGEN either produces a new IS of length N, or returns "no more IS". At a high level, SLEC-CF works as shown in Fig. 2. For Step 2, we used the simulator built in the SLEC tool. We focus on our main contribution, the IS generator.

Coverage. SLEC-CF also computes the level of CS value coverage, denoted by Cov, achieved during its exploration. Informally, Cov measures the fraction of CS values which have been encountered, or proven to be unrealizable. Thus $0 \leq Cov \leq 1$, where 1 denotes full coverage. SLEC-CF maintains a set, called $CSVal$, of encountered CS values, and unrealizable CS value prefixes. Initially, $CSVal = \emptyset$. Each invocation of CEXGEN updates $CSVal$ as a side-effect. At the end, Cov is obtained by dividing the number of CS values represented by $CSVal$ with the total number of possible CS values. CS values in different cycles are considered to be independent.

Example. Suppose we have $S = \{v_1, v_2\}$, and each v_i can have 4 possible values $\{0, 1, 2, 3\}$. Then the total number of CS values is 16. Suppose we have an input sequence σ over three clocks cycles which induces the following CS values: (i) cycle 1: $v_1 = 1, v_2 = 2$; (ii) cycle 2: $v_1 = 2, v_2 = 0$; and (iii) cycle 3: $v_1 = 1, v_2 = 2$. Thus, we have encountered two

```
Algorithm EXHAUSTIVE ($\sigma$):
(1) If this is the first call to EXHAUSTIVE , then initialize
    $St = \mathcal{S}(\sigma)$, and $T = (1, 1, \ldots, 1)$.
(2) If $St$ is empty return "no more IS". Otherwise, let
    $(v_i, d_i)$ be the top-most element of $St$.
(3) Recall that $T = (t_1, t_2, \ldots, t_N)$. If $t_i = m_i$, we
    have explored all possible values of $v_i$. In this case,
    pop $St$ and repeat from step 2. Otherwise, increment
    $t_i$, change the top element of $St$ to $(v_i, \oplus d_i)$, and invoke
    GETASSIGN($St$). Note that, in general, $St$ can specify the
    values of $S$ for fewer than $N$ clock cycles.
(4) If GETASSIGN($St$) returns an IS $\sigma'$, then: (a) set $St$ to
    $\mathcal{S}(\sigma')$; (b) set the values of $t_{i+1}$ through $t_N$ to 1; and (c)
    return $\sigma'$ as the new IS.
(5) Otherwise, GETASSIGN($St$) returns $\perp$ and we know
    that $St$ is unrealizable. Repeat from Step 2.
```

Fig. 3. Pseudo-code for EXHAUSTIVE IS generation.

distinct CS values. In addition, suppose we prove that any CS value in cycle 1 with $v_2 \in \{1, 3\}$ is unrealizable. Then, we have proven 8 CS values to be unrealizable, which are distinct from those induced by σ. Thus, in all, $CSVal$ represents 10 CS values. Therefore, $Cov = \frac{10}{16} = 0.625$.

Selecting Control Signals. MUX selectors represent decision points in the circuit's semantics, and the key idea behind concolic verification is to construct testcases that explore different corner cases in a program's execution. The set of all MUX selectors, Sig, is therefore a natural choice for S. This is what we use. Our benchmarks are derived from High-Level Synthesis (HLS) testcases. In HLS, hardware is compiled from a program written in a high-level language. In our case, the circuit C_2 is a RTL generated from a C++ program P via the CATAPULT [17] synthesizer, while C_1 is generated from P by the SLEC tool. SLEC converts branch conditions (if-then-else, switch-case, conditional expressions, etc.) appearing in P into corresponding MUX selectors in C_1. Therefore, Sig represents source level branch conditions as well.

IV. Input Sequence Generation

Since each control signal is a MUX selector, it has a finite bit-width. We assume that the set of possible values of each signal is ordered numerically, with wrap-around, and $\oplus v$ denotes the "next value" after v in this ordering. We use the ordering to explore all values of the signal systematically. For example, if the signal is 2-bit, and we start our exploration with "10", then the sequence of values explored is "10, 11, 00, 01", and $\oplus(11) = 00$. Note that, unless otherwise mentioned, all IS's mentions in this section are of length N. We propose an IS generation scheme – EXHAUSTIVE – which explores the set of all values of S lexicographically using a DFS strategy, starting with the value of S induced by the random IS selected in Step 1 of SLEC-CF (cf. Fig. 2). We now present EXHAUSTIVE.

A. Exhaustive IS Generation

EXHAUSTIVE maintains a stack St of values of S for up to N clock cycles. For simplicity, let $S =$

$\{v\}$ where v is Boolean. Then St is always of form $\langle(v_1,d_1),(v_2,d_2),\ldots,(v_k,d_k)\rangle$ where $k \leq N$, v_i is the instance of v at clock cycle i, and $d_i \in \{\bot,\top\}$ denotes the value of v_i. Given an IS σ, we write $\mathcal{S}(\sigma)$ to denote the value of St induced by σ. Thus, $\mathcal{S}(\sigma) = \langle(v_1,d_1),(v_2,d_2),\ldots,(v_N,d_N)\rangle$, where d_i is the value of v_i obtained by simulating C_1 and C_2 with σ.

We write $St_1 \preceq St_2$ to mean that the stack St_1 is a prefix of the stack St_2. We say that a stack St is "realizable" if there exists an IS σ such that $St \preceq \mathcal{S}(\sigma)$. Otherwise, we say that St is "unrealizable". We assume a procedure GETASSIGN() that takes a stack St as input, and returns: (i) an IS σ such that $St \preceq \mathcal{S}(\sigma)$ if St is realizable; or (ii) $\bot$ otherwise. We present the actual implementation of GETASSIGN in Section IV-B.

We also maintain a vector of numbers $T = (t_1, t_1, \ldots, t_N)$ where t_i denotes the "number of values of v_i" that have been explored already. In essence, our search explores all values of T in lexicographic order, starting with $(1,1,\ldots,1)$ and ending with $(m_1,m_2,\ldots,m_N)$ where m_i is the total number of possible values of v_i. Note that, for this article, since v is assumed to be Boolean, $m_i = 2$ for $1 \leq i \leq N$. The pseudo-code for EXHAUSTIVE given an IS σ is shown in Figure 3.

Valid Stimulus Generation. Our algorithm can handle a set of "constant signals", i.e., signals that are always assumed to be $\top$. Such signals are maintained separately, and the stack is extended by assigning $\top$ to each of them prior to the invocation of GETASSIGN(). Such constant signals are used to encode user-specified design constraints. Since their values are always pinned to $\top$, any IS (i.e., stimulus) returned by GETASSIGN() must be valid, i.e., satisfy the design constraints. Note that constant signals can encode arbitrary safety properties over multiple clock cycles, including System-Verilog assertions. In our implementation, the SLEC tool automatically performs the conversion from design-level assertions to circuit-level signals.

Random IS Generation. For comparison, we also implemented a IS generator, called RANDOM, which produces a random IS on every call. When the number of calls exceeds a user-defined limit, it returns "no more IS". In our experiments, we use RANDOM as a proxy for random simulation.

B. *Implementing* GETASSIGN

Figure 4 shows the pseudo-code for GETASSIGN. It relies on a procedure *IsEqual* that takes a circuit $C = (Q, I, O, R)$, and two output variables $\tilde{o}_1, \tilde{o}_2 \in O$, and returns: (i) $PROVEN$ if for all input sequences σ, $C(\sigma)(\tilde{o}_1) = C(\sigma)(\tilde{o}_2)$; or (ii) an IS σ such that $C(\sigma)(\tilde{o}_1) \neq C(\sigma)(\tilde{o}_2)$. For our experiments, we use the implementation of *IsEqual* available in the SLEC [18] tool. Recall that our overall goal is to falsify the equivalence of circuits $C_1 = (Q_1, I, O_1, R_1)$ and $C_2 = (Q_2, I, O_2, R_2)$. The "synchronous composition" of C_1 and C_2, denoted $C_1 \parallel C_2$, is the circuit $(Q_1 \cup Q_2, I, O_1 \cup O_2, R_{\parallel})$ such that:

$$R_{\parallel}(q,i) = (q',o) \iff \bigwedge_{j=1}^{2} R_j(q \cap Q_j, i) = (q' \cap Q_j, o \cap O_j)$$

Thus, one step of $C_1 \parallel C_2$ corresponds to one step of C_1 and one step of C_2. Formally, GETASSIGN: (i) constructs $C_1 \parallel C_2$;

```
Algorithm GETASSIGN ($St$):
  1) Let $St = \langle(v_1,d_1),(v_2,d_2),\ldots,(v_k,d_k)\rangle$; recall that
     $v$ is the only control signal; $v_i$ and $d_i$ are, respec-
     tively, its instance and value at clock cycle $i$.
  2) Let $\tilde{C} = C_1 \parallel C_2$
  3) Add two state variables, $ctr$ and $flag$, to $\tilde{C}$; their
     initial value is $1$
  4) Add two outputs $\tilde{o}_1$ and $\tilde{o}_2$ to $\tilde{C}$
  5) Update the transition relation of $\tilde{C}$ such that:
     – $ctr$ is incremented at each step: $ctr' := ctr + 1$
     – $flag$ keeps track of whether the sequence of
     values of $v$ observed so far matches $St$:
        $flag' := flag \wedge (v = d_{ctr})$
     – $\tilde{o}_2 := \bot$ and $\tilde{o}_1 := (ctr = N + 1) \wedge flag$.
  6) If $IsEqual(\tilde{C}, \tilde{o}_1, \tilde{o}_2) = PROVEN$ then return $\bot$
  7) Else return the IS produced by $IsEqual(\tilde{C}, \tilde{o}_1, \tilde{o}_2)$
```

Fig. 4. Pseudo-code for GETASSIGN.

(ii) augments it with state variables and outputs to encode St; and (iii) invokes *IsEqual* and interprets its result appropriately. St is encoded by: (i) adding a new variable ctr to count the number of clock cycles; (ii) adding a new variable $flag$ to track whether the sequence of values of v observed so far matches St; and (iii) ensuring that the output $\tilde{o}_1$ is true iff $flag = \top$ after N clock cycles. Thus, if $IsEqual(\tilde{C}, \tilde{o}_1, \tilde{o}_2)$ returns an IS σ, then $St \preceq \mathcal{S}(\sigma)$, and if $IsEqual(\tilde{C}, \tilde{o}_1, \tilde{o}_2)$ return $PROVEN$, then St is unrealizable.

V. Incremental Verification

The most expensive step in SLEC-CF is the call to GETASSIGN, whose complexity increases with N and $|S|$. To address this, we present an incremental version of SLEC-CF which starts with $N = 1$, and a small S, and iteratively enlarges them. We select a "control signal increment" value S_Δ, and create a sequence of increasingly larger CS sets $\tilde{S}_1 \subset \tilde{S}_2 \subset \cdots \subset \tilde{S}_M$, such that $M = \lceil \frac{|Sig|}{S_\Delta} \rceil$ and $\tilde{S}_M = Sig$, as follows:

- Sort *Sig* in the topological order induced by the circuits C_1 and C_2 from the inputs to the outputs.
- Partition *Sig* into M fragments $X_1, X_2, \ldots, X_M$, such that: $\forall 1 \leq i < M \centerdot |X_i| = S_\Delta$, and $|X_M| \leq S_\Delta$.
- Then, for $1 \leq i \leq M \centerdot \tilde{S}_i = \bigcup_{1 \leq j \leq i} X_j$.

We also select a maximum search depth N_{max}, and a search depth increment value N_Δ. Then, the pseudo-code for incremental SLEC-CF, denoted SLEC-CF-INC, is shown in Figure 5. Essentially, SLEC-CF-INC invokes SLEC-CF repeatedly, using two nested loops. The outer loop increases the set of control signals, while the inner loop increases the search depth.

The version of SLEC-CF-INC in Figure 5 makes both duplicate calls to GETASSIGN, as well as calls whose results can be derived from prior call results. For example, duplicate calls happen across multiple invocations of SLEC-CF with the same S but different N. We now present a caching mechanism to eliminate such redundant calls to GETASSIGN. The key idea is that the (non)realizability of a stack St can be implied by the (non)realizability of its extensions and restrictions.

```
Algorithm SLEC-CF-INC :
(1) For $j := 1; j \leq M; j := j + 1$:
(2)     For $N := 1; N \leq N_{max}; N := N + N_{\Delta}$:
(3)         Invoke SLEC-CF ($\tilde{S}_j$, $N$)
(4)         If SLEC-CF ($\tilde{S}_j$, $N$) reports a CEX $\tilde{\sigma}$
(5)             Report $\tilde{\sigma}$ as a CEX, and exit.
(6) Report "no CEX found" and exit.
```

Fig. 5. Overall SLEC-CF-INC algorithm.

```
Algorithm GETASSIGN-CACHED ($S$, $N$, $St$):
(1) If $\exists St' \in UnsatCache \centerdot St' \preceq St$ Return $\perp$
(2) If $\exists(\sigma', N') \in SatCache \centerdot St \preceq \mathcal{S}(\sigma', N')$ Return $\sigma'$
(3) $r :=$ GETASSIGN ($St$)
(4) If $r = \perp$ Add $St$ to $UnsatCache$
(5) Else Add $(r, N)$ to $SatCache$
(6) Return $r$
```

Fig. 6. Pseudo-code replacing call to GETASSIGN in SLEC-CF-INC.

Suppose we are in a call to SLEC-CF (S, N). Then any stack maintained by CEXGEN is a "partial" function that maps a control signal $v \in S$ and a clock cycle $i \in [1, N]$ to the value of v at cycle i. The mapping is partial because control signals become undefined as the stack shrinks. Let $Dom(X)$ denote the domain of a function X. We say that stack St_1 is a restriction of stack St_2, denoted $St_1 \preceq St_2$, iff $Dom(St_1) \subseteq Dom(St_2)$, and: $\forall(v, i) \in Dom(St_1) \centerdot St_1(v, i) = St_2(v, i)$. The extension relation is the inverse of restriction.

Proposition 1: Let St_1 and St_2 be two stacks such that $St_1 \preceq St_2$. Then: (i) if St_1 is unrealizable, then so is St_2; (ii) if St_2 is realizable, then so is St_1.

Stack Induced by an IS. Given an input sequence σ over at least N clock cycles, we write $\mathcal{S}(\sigma, N)$ to denote the stack whose domain is $Sig \times [1, N]$, such that $\forall(v, i) \in Sig \times [1, N] \centerdot \mathcal{S}(\sigma, N)(v, i)$ is the value assigned to v at clock cycle i by simulating C_1 and C_2 with σ. Note that if a call to GETASSIGN(St) by SLEC-CF (S,N) returns an IS σ, then $\mathcal{S}(\sigma, N)$ is realizable.

Caching Strategy. SLEC-CF-INC maintains two caches of results from GETASSIGN – $UnsatCache$ for calls that return $\perp$, and $SatCache$ for calls that return an IS σ. Specifically, the following two cache invariants are always maintained:

- **(UNSAT)** $UnsatCache$ contains an entry St iff a call to GETASSIGN (St) by SLEC-CF (S, N) returned $\perp$.
- **(SAT)** $SatCache$ contains an entry (σ, N) iff a call to GETASSIGN (St) by SLEC-CF (S, N) returned σ.

The call to GETASSIGN (St) in SLEC-CF is replaced by a cached version, GETASSIGN-CACHED (S, N, St), whose pseudo-code is shown in Fig. 6. Lines 1 and 2 look for a hit in $UnsatCache$ and $SatCache$, respectively. If neither cache results in a hit, line 3 calls GETASSIGN, and depending on its result, either $UnsatCache$ or $SatCache$ is updated at lines 4 and 5, respectively. Finally, the result is returned at line 6. Note that the cache lookup cost is non-trivial, especially as the cache grows in size. The check for $St_1 \preceq St_2$ requires, in the worst case, inspecting all the elements of St_1. In our current implementation, each cache entry is maintained separately. Thus, each cache lookup involves independent checks for each entry. Despite such a straightforward implementation, our experiments indicate measurable benefits from using the cache.

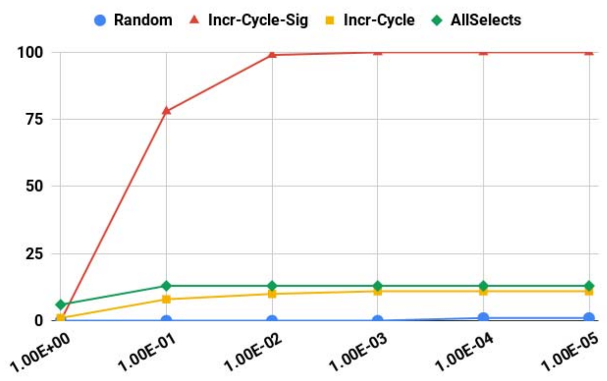

Fig. 7. Coverage achieved on 100 correct examples in 5h; X = coverage level; Y = # of problems where a technique achieved that coverage.

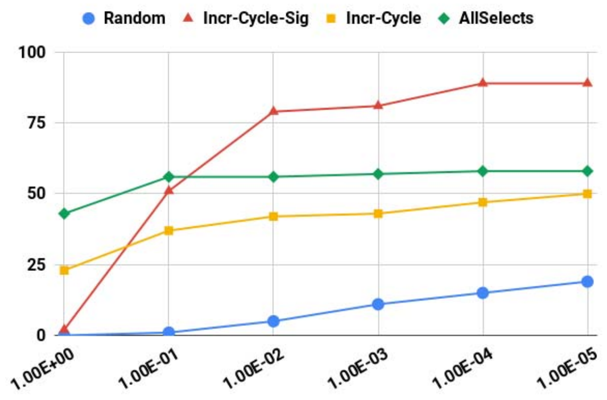

Fig. 8. Coverage achieved on 100 buggy examples in 2h; X = coverage level; Y = # of problems where a technique achieved that coverage.

VI. EXPERIMENTS

We used a benchmark of examples from SLEC's industrial HLS regression suite, containing about 10K correct examples, and 1300 buggy examples. We selected the 100 hardest examples from each category, where hardness is measured by the time required by SLEC to solve the problem using its standard non-concolic engine. We compared SLEC-CF with random simulation, and measured the effectiveness of incremental SLEC-CF. Experiments were done on a farm of machines, each with an Intel Xeon processor with 8 cores running at 2.5GHz, 64GB of RAM, and CentOS 6.6. Each experiment was limited to 8GB memory. For the correct and buggy examples, we used a time limit of 5h and 2h, respectively.

Coverage Results. First we compared the coverage achieved on the correct examples by four strategies: (i) Random = random simulation; (ii) AllSelects = SLEC-CF with $N = 10$, $S = Sig$; (iii) Incr-Cycle = SLEC-CF-INC with $N_{\Delta} = 1$,

$N_{max} = 10$, $S_\Delta = |Sig|$; and (iv) Incr-Cycle-Sig = SLEC-CF-INC with $N_\Delta = 1$, $N_{max} = 10$, $S_\Delta = 10$. Fig. 7 shows the results where the X axis denotes exponentially decreasing coverage values $\langle 1, 0.1, 0.01, \ldots, 10^{-5} \rangle$, and the Y axis shows the number of examples for which a specific strategy achieved at least a coverage value.

All three concolic strategies outperform Random. In particular, Random cannot achieve 10% coverage on even one example, while Incr-Cycle-Sig achieves it on 78 examples. Among the concolic strategies, AllSelects achieves full coverage on most (six) examples. However, its performance plateaus quickly after that. For example, while Incr-Cycle-Sig achieves 10% coverage on 78 examples, AllSelects can achieve it on only 13 examples. The reason is that, on easier examples with simple solver calls, AllSelects performs well, while Incr-Cycle-Sig wastes time as it restarts the search with each iteration. As the solver calls become harder, AllSelects is choked off, while Incr-Cycle-Sig continues to make some progress. Incr-Cycle's coverage is always dominated by either AllSelects or Incr-Cycle-Sig. In practice, we do not know how hard the solver calls will be. Therefore, a good strategy is to run AllSelects first, and then use Incr-Cycle-Sig if a satisfactory coverage is not achieved.

Fig. 8 shows the same results on the buggy examples. We see the same broad trends as in the case of the correct ones. Random's performance is the worst. While AllSelects achieves full coverage on most (43) examples, its performance again plateaus. By 10% coverage, Incr-Cycle-Sig has almost as many examples as AllSelects (51 vs. 56) and by 1% coverage it overtakes AllSelects (79 vs. 56 problems). This gives us further confidence about the robustness of our conclusions.

Falsification. Concolic search is also superior to Random in terms of finding falsifying stimuli. There are 42 buggy examples where AllSelects finds a CEX but Random does not, while there are only 3 cases where the opposite is true. When we compare Incr-Cycle-Sig with Random, the corresponding numbers are 24 and 13, meaning that Incr-Cycle-Sig is superior to Random as well. However, overall, AllSelects is better than Incr-Cycle-Sig at finding a CEX. There are 29 examples where AllSelects finds a CEX but Incr-Cycle-Sig does not. Of these, AllSelects also yields higher coverage in 15 cases. Thus, the relationship between coverage and the likelihood finding a CEX is not direct.

Caching. Disabling the cache leads to measurable performance penalties for the buggy examples. The number of examples where we achieved a coverage of at least 10% drops from 51 to 39. The geometric mean of coverage obtained drops from 0.09 to 0.04, and the median drops from 0.121 to 0.078. The total number of CEXs found also drops from 73 to 42. Note that we use geometric mean since the coverage values have a very big range. The number of solver calls jumps from about 287K to 970K. There were about 15M $SatCache$ hits, and 267M $UnsatCache$ hits. For the correct examples, the number of solver calls jumps from 127K to 333K. There were about 540K hits to the $SatCache$, and 27M hits to the $UnsatCache$. However, the overall performance was quite similar with-and-without the cache, indicating that the cache lookup costs are non-trivial, and the relatively smaller number of cache hits (compared to the buggy examples) mean that the cache benefits do not clearly outweigh its costs.

VII. Conclusion

We presented an approach, called SLEC-CF, for hardware equivalence checking using concolic search. SLEC-CF provides much higher coverage than random simulation, and always produces stimuli which respect user-specified design constraints. We also developed an incremental version of SLEC-CF that iteratively increases the set of control signals, as well as the search depth. Experiments over an industrial benchmark demonstrate the superiority of SLEC-CF, and the effectiveness of incrementality at providing good coverage for complex testcases. New schemes to determine a good set of control signals (e.g., by using source-level information from C++ designs) can further improve the effectiveness of SLEC-CF. In addition, caching benefits can be increased further with a more sophisticated cache implementation, e.g., one that uses sharing between the entries. Finally, additional work is needed to store efficiently the multiple CEXs generated by our approach.

References

[1] C. Pixley, "A theory and implementation of sequential hardware equivalence," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 11, no. 12, 1992.

[2] G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future," *IEEE Design Test of Computers*, vol. 26, no. 4, 2009.

[3] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur, and N. Sharma, "Non-cycle-accurate sequential equivalence checking," in *Proc. of DAC*, 2009.

[4] J. W. Duran and S. C. Ntafos, "An Evaluation of Random Testing," *IEEE TSE*, vol. SE-10, no. 4, 1984.

[5] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proc. of PLDI*, 2005.

[6] H. Kelly J., V. Dan S., C. John J., and R. Leanna K., "A Practical Tutorial on Modified Condition/Decision Coverage," NASA Langley Research Center, Technical report NASA/TM-2001-210876, 2001, https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20010057789.pdf.

[7] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," Microsoft Research, Technical report TR-2007-58, 2007.

[8] K. Sen and G. Agha, "CUTE and jcute: Concolic unit testing and explicit path model-checking tools," in *Proc. of CAV*, vol. 4144, 2006.

[9] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. of OSDI*, 2008.

[10] L. Liu and S. Vasudevan, "STAR: Generating input vectors for design validation by static analysis of RTL," in *Proc. of HLDVT*, 2009.

[11] ——, "Efficient validation input generation in RTL by hybridized source code analysis," in *Proc. of DATE*, 2011.

[12] Y. Zhang, W. Feng, and M. Huang, "Automatic generation of high-coverage tests for RTL designs using software techniques and tools," in *Proc. of ICIEA*, 2016.

[13] A. Ahmed and P. Mishra, "QUEBS: qualifying event based search in concolic testing for validation of RTL models," in *Proc. of ICCD*, 2017.

[14] S. Pinto and M. S. Hsiao, "RTL functional test generation using factored concolic execution," in *Proc. of ITC*, 2017.

[15] T. Matsumoto, T. Nishihara, Y. Kojima, and M. Fujita, "Equivalence checking of high-level designs based on symbolic simulation," in *Proc. of ICCAS*, 2009.

[16] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zue, *Bounded Model Checking*, ser. Advances in computers, 2003, vol. 58.

[17] Catapult C Synthesis, http://calypto.agranderdesign.com/catapult_c_synthesis.php.

[18] SLEC System-HLS, http://calypto.agranderdesign.com/slecsystemhls.php.