C-To-Verilog Translation Validation

Alan Leung Dimitar Bounov Sorin Lerner
University of California, San Diego
{aleung, dbounov, lerner}@cs.ucsd.edu

***Abstract*—To offset the high engineering cost of digital circuit design, hardware engineers are looking increasingly toward high-level languages such as C and C++ to implement their designs. To do this, they employ High-Level Synthesis (HLS) tools that translate their high-level specifications down to a hardware description language such as Verilog. Unfortunately, HLS tools themselves employ sophisticated optimization passes that may have bugs that silently introduce errors in realized hardware. The cost of such errors is high, as hardware is costly or impossible to repair if software patching is not an option. In this work, we present a translation validation approach for verifying the correctness of the HLS translation process. Given an initial C program and the generated Verilog code, our approach establishes their equivalence without relying on any intermediate results or representations produced by the HLS tool. We implemented our approach in a tool called VTV that is able to validate a body of programs compiled by the Xilinx Vivado HLS compiler.**

I. INTRODUCTION

To offset the high cost of digital circuit design, hardware engineers are increasingly employing general purpose programming languages like C or C++ to implement their designs. In a process known as *high-level synthesis* (HLS), the hardware designer implements desired functionality in a high-level language. An HLS tool then translates this high-level program into a hardware description language such as Verilog. HLS has been a topic of research for the past two decades [1], [2], [3], and the technology has matured to the point that commercial tools are becoming increasingly available. Examples include the Xilinx Vivado HLS compiler, the Synopsis Synphony C compiler, and the Bluespec compiler.

Unfortunately, HLS compilers are complex tools, akin to compilers, and it is well known that compilers are notoriously difficult to make reliable. For example, a study by Yang et al. [4] found hundreds of previously unknown bugs in mature, well-tested, and broadly used compilers. Errors in HLS compilation are very costly: once manufactured, hardware is expensive to repair, if even possible.

One well-known way of increasing the reliability of compilers is a technique called *translation validation* [5], [6]. Translation validation works as follows: on each run of the compiler, a separate tool tries to prove that the input program is semantically equivalent to the corresponding output program. Since HLS tools are in essence compilers, it is natural to apply translation validation techniques to them, or more broadly compiler verification techniques. Although there is previous work in this space [7], [8], [9], there is no previous approach which simultaneously: (1) performs automated validation of the *entire* C-to-Verilog translation from source to source, (2) works in a *blackbox* setting, where only the Verilog program and no other output is taken from the compiler, and (3) works in the setting of a *realistic* industrial strength HLS compiler. In this paper, we present an approach with all three benefits.

In our approach, we first convert both the C program and the Verilog program into a common intermediate representation (IR), then use bisimulation techniques to prove the two resulting IR programs equivalent. The main technical challenge lies in translating the naturally concurrent and unstructured Verilog code into a structured transition diagram amenable to bisimulation. Other technical challenges include the handling of low-level protocols for memory operations and termination.

To demonstrate the viability of our approach, we have implemented a tool called VTV that has verified translation of 17 benchmarks compiled by the Xilinx Vivado HLS compiler, with an average of 300 seconds per benchmark.

The rest of this paper is organized as follows: Section II introduces the Verilog language and motivates the equivalence checking problem. Section III presents a formal account of UIR diagrams, the intermediate representation we derive for both Verilog and C programs. Section IV covers the algorithms for translating Verilog and C programs to UIR diagrams. Sections V and VI cover the equivalence checking and inference algorithms. Section VII presents our experimental evaluation.

II. OVERVIEW

We present an overview of our approach through a simple example. Consider the following C function and its resulting Verilog module declaration, generated by the Vivado HLS Compiler. This example code computes the 32-bit CRC checksum of the input string `buf`.

```
// C implementation
ulong crc32(ulong crc, uchar buf[1024], ulong off, ulong len, ulong crc_table[256]){
    crc = crc ^ 0xffffffffL;
    unsigned int i;
    for (i=0; i<len; i++) crc = crc_table[(crc ^ buf[off + i]) & 0xff] ^ (crc >> 8);
    return crc ^ 0xffffffffL;
}
// Verilog module declaration
module crc32
    (clk, rst, start, done, idle, ready, crc, buf_address0, buf_ce0, buf_q0, off,
    len, crc_table_address0, crc_table_ce0, crc_table_q0, return);
```

Verilog contains types corresponding to the basic elements of hardware: 1) ports and wires, which just like their physical analogues do not hold state, but propagate values, and 2) registers, which hold their previously assigned values. Registers can store both data and control (for example the current state of the state machine), but there is no distinction between the two in the language itself.

Modules encapsulate logic and roughly correspond to discrete components in a hierarchical design: the communication between modules and their environment occurs through their declared ports. In the case of `crc32`, the ports include: 1) input arguments `crc`, `off`, `len`; 2)

```
// clocked always block
always @ (posedge clk) begin
  if ((2'b10 == CS)) begin i_reg <= i_reg_2; end
  else if (((2'b0 == CS) &~(start == 1'b0))) begin i_reg <= 3'b0; end
end

// unclocked always block
always @ (CS or exitcond) begin
  if (((2'b1 == CS) & (exitcond == 1'b0))) begin buf_ce0 = 1'b1; end
  else begin buf_ce0 = 1'b0; end
end

// continuous assignments
assign i_wire   = (i_reg + 3'b1);
assign exitcond = (i_reg == len ? 1'b1 : 1'b0);
```

Fig. 1: **Verilog Constructs.** `CS` holds the current FSM state.

an output value `return`; 3) miscellaneous control signals such as `clk`, `rst`, `start`, `done`; and 4) a memory interface consisting of inputs `buf_q0`, `crc_table_q0`, and outputs `buf_ce0`, `buf_address0`, `crc_table_ce0`, `crc_table_address0`.

Salient fragments of the Verilog code for `crc32` are shown in Figure 1. The actual logic of the module is implemented using two constructs: `always` blocks and continuous assignments. The top of Figure 1 shows an `always` block that updates the register `i_reg`, which corresponds to the loop induction variable `i` in the C program. The semantics of `always` blocks is such that the body of the block executes whenever a value in its *sensitivity list*, enclosed in `@()`, is updated. Stateful register updates occur in `always` blocks sensitized only on `posedge clk`, the positive edge of the clock signal. We call such blocks clocked `always` blocks. The second block in Figure 1 is an `always` block of a slightly different form. This block sets port `buf_ce0` to 1 in state 1 when `exitcond` is 0, which encodes the fact that we read from the `buf` array in the body of the loop. The sensitivity list contains precisely those identifiers read within the body, and the conditionals within the body are exhaustive. Blocks of this kind, which we called unclocked `always` blocks, implement combinational circuits with no internal state. The final construct to note, the *continuous assignment*, appears on the last two lines of Figure 1. Continuous assignments are an alternate syntax for combinational logic. They update their left-hand sides whenever the right-hand side changes.

Challenges. There are two key challenges to developing a translation validator between C and HLS-derived Verilog programs. The first challenge is to resolve the impedance mismatch between the low-level nature of the Verilog program, which conceptually conflates control and data state, and the high-level nature of C specifications which make obvious the distinction via structured control flow (loops and branches). The control flow of a typical Verilog program is spread throughout the program in various clocked `always` blocks: the simple act of updating the loop index in `crc32` is implemented across two clocked `always` blocks and a continuous assignment, and this does not even account for the actual computation of the body of the loop. Complicating matters further, the Verilog implementation makes use of various hardware-specific protocols that must be lifted to a higher representation to ease the burden of reasoning about program behavior. For instance, where the C program has an array access such as `buf[off + i]`, the Verilog program obeys a protocol in which an enable and address are provided in one cycle before the load value is read in the next cycle. As a result, reasoning about memory equality requires us to soundly detect loads and stores according to this protocol.

The second challenge is to infer the invariants necessary for establishing equivalence between the Verilog and C programs in the face of optimizations. For instance, the HLS compiler optimized the value of the loop exit condition `exitcond` from `i < len` to `i == len`. It did so by inferring the program invariant $\mathtt{i} \leq \mathtt{len}$, which together with the loop exit condition $\mathtt{i} \geq \mathtt{len}$ implies `i == len`. Our translation validator must therefore be able to infer such additional invariants to effectively prove program equivalence.

Our Approach. We apply two insights to address these challenges. The first insight is that the structure of HLS-derived Verilog is highly constrained and amenable to inexpensive syntactic reasoning. Particularly, the predicates occurring in conditionals are likely to be conditions under which finite state machine (FSM) transitions occur, so we extract them using a semantics preserving rewrite system. With the predicates extracted, we then employ an automated theorem prover to derive the static transition relation with respect to those predicates, essentially rederiving the underlying FSM implicitly defined by the Verilog specification. Proving equivalence between the extracted diagram and the original C control flow graph then reduces to the well-known problem of finding a bisimulation relation [10] that relates the states of both programs at key locations using logical formulae (invariants) over those states.

Our second insight, akin to [11], is to employ statistical invariant inference to guess the required invariants from a rich constraint language, and then to check that these invariants in fact hold. In particular, we build on the Daikon tool [12], which uses machine learning to guess likely invariants from execution traces. We adapt Daikon to the context of two programs by presenting pairs of traces, and achieve soundness with a checking algorithm that retains only provably valid invariants.

III. Unified Intermediate Representation

In this section we formally define our underlying program representation. We use a generalized form of control flow graph called a UIR diagram to model both C and Verilog programs. We define a UIR state $\sigma \in \Sigma$ to be a map *Vars* $\rightarrow$ *Vals* from variables to their values, where Σ is the set of all states.

Definition 1 (UIR Diagram): A UIR diagram π is a tuple $(\mathcal{L}, \mathcal{M}, \mathcal{E}, s, \phi, \mathcal{F}, [\![\cdot]\!])$, where

1) $\mathcal{L}$ is a finite set of labels,
2) $M : \mathcal{L} \rightarrow [\mathcal{O}]$ is a map from labels to sequences of operations,
3) $\mathcal{E} \subseteq (\mathcal{L} \times (\Sigma \rightarrow \mathcal{B}) \times \mathcal{L})$ is a finite set of tuples (l, p, l') called transitions where l, l' are labels and p is a state predicate,
4) $s \in \mathcal{L}$ is a designated start label,
5) $\mathcal{F} \subseteq \mathcal{L}$ is a (possibly empty) set of final labels,
6) $\phi : \Sigma \rightarrow \mathcal{B}$ is a state predicate, and
7) $[\![\cdot]\!] : \mathcal{O} \rightarrow (\Sigma \rightarrow \Sigma)$ is a map from operations to their corresponding state transformation functions.

The predicate ϕ defines the set of valid initial states for the program. In practice, ϕ is a manually-specified well-formedness condition on inputs to the program. Note that UIR diagrams are parametric in an underlying semantics of operations and expressions via $[\![\cdot]\!]$. This is vital for building equivalence proofs between C and Verilog programs, as we assume either semantic function $[\![\cdot]\!]_c$ or $[\![\cdot]\!]_v$ for UIR diagrams derived from C or Verilog, respectively.

Definition 2 (Semantic Step): Given UIR diagram $\pi = (\mathcal{L}, \mathcal{M}, \mathcal{E}, s, \phi, \mathcal{F}, [\![\cdot]\!])$, a configuration is a pair $\langle l, \sigma \rangle$ where $l \in \mathcal{L}$ and $\sigma \in \Sigma$. We define the semantic step relation on configurations as follows: $\langle l, \sigma \rangle \hookrightarrow \langle l', \sigma' \rangle \iff$

$$\exists p.\ (l, p, l') \in \mathcal{E} \wedge p(\sigma') \wedge \sigma' = \overline{[\![M(l)]\!]}(\sigma)$$

where $\overline{[\![\cdot]\!]}$ denotes $[\![\cdot]\!]$ lifted to sequences via function composition.

Definition 3 (Trace): A trace of a UIR diagram is a finite sequence of configurations $\tau = \langle l_0, \sigma_0 \rangle, ..., \langle l_n, \sigma_n \rangle$ such that $l_0 = s$, $\phi(\sigma_0) = true$, and $\langle l_k, \sigma_k \rangle \hookrightarrow \langle l_{k+1}, \sigma_{k+1} \rangle$ for all $0 \leq k < n$. We denote the set of all traces of π by η_π.

We define $\hookrightarrow^+$ to be the relation such that $\langle l, \sigma \rangle \hookrightarrow^+ \langle l', \sigma' \rangle$ iff $\langle l, \sigma \rangle \hookrightarrow \ldots \hookrightarrow \langle l', \sigma' \rangle$. Informally, $\hookrightarrow^+$ relates pairs of configurations at least one step apart.

Definition 4 (Relative Equivalence of Traces): Given predicates $\Phi, \Phi' : \Sigma \times \Sigma \to \mathcal{B}$ on pairs of states, we say that two traces τ_1, τ_2 with initial states σ_1, σ_2 and final states σ_1', σ_2', respectively, are relatively equivalent with respect to predicates Φ, Φ' iff $\Phi(\sigma_1, \sigma_2) \wedge \Phi'(\sigma_1', \sigma_2')$, which we denote $\tau_1 \equiv^{\Phi}_{\Phi'} \tau_2$.

Predicates Φ, Φ' define mappings from variables in one trace to their equivalents in the other trace via equality constraints of the form $u = v$, where $u \in dom(\Sigma_1)$ and $v \in dom(\Sigma_2)$. Together, the equality constraints of Φ define the set of variables whose values must be equal at the beginning of execution (inputs), and Φ' define the set of variables whose values must be equal at termination (return values and persistent memory). We call Φ a *relational precondition* and Φ' a *relational postcondition*. In practice, we use these relational pre- and postconditions to establish expected equalities between visible variables (such as inputs, outputs, and persistent arrays), at program entry and exit.

IV. TRANSLATION TO UIR

In this section we describe a minimal fragment of Verilog called µVerilog, then describe an algorithm for normalizing and converting µVerilog to UIR diagrams. The syntax of µVerilog programs is a small subset of Verilog syntax, consisting of only the following constructs: module declarations; input, output, wire, and register types; always blocks; nonblocking, blocking, and continuous assignments; and arithmetic and logical expressions. Figure 2 shows the skeleton of a normalized µVerilog program, where nba and ca are nonblocking and continuous assignment, respectively, and we have used the notation $\overline{x}$ as shorthand for 0 or more repetitions of x.

Normalization. We normalize µVerilog using a system of 7 rewrite rules, which we describe here. These rules reduce a µVerilog program to a simple syntactic form such that we can extract its control-flow predicates. MERGE-ALWAYS combines the bodies of two clocked `always` blocks. Repeated application eventually reduces all clocked `always` blocks into one. MERGE-IFS combines two `if-else` chains within a clocked `always` block into a single chain by exhaustive case analysis. FLATTEN-IFS combines two nested `if-else` constructs within a clocked `always` block into a single chain by exhaustive case analysis. PRUNE-F and PRUNE-T prune unreachable branches. NEST pushes assignments within a clocked `always` block into the body of all neighboring `if-else` chains. REDUCE-ASSN converts unclocked `always` blocks into continuous assignments.

```
module $\overline{decl}$
  always @(posedge clk)
    if($e_1$) $\overline{nba_1}$
    else if($e_2$) $\overline{nba_2}$
    else if($e_3$) $\overline{nba_3}$
    . . .
    else $\overline{nba_n}$
  $\overline{ca}$
```

```
do
  $e_1$                                  $\rightarrow$   $adjoin(conc(\overline{nba_1}), \overline{ca})$
  $\neg e_1 \wedge e_2$                  $\rightarrow$   $adjoin(conc(\overline{nba_2}), \overline{ca})$
  $\neg e_1 \wedge \neg e_2 \wedge e_3$  $\rightarrow$   $adjoin(conc(\overline{nba_3}), \overline{ca})$
  . . .
  $\neg e_1 \wedge \ldots \wedge \neg e_{n-1}$  $\rightarrow$   $adjoin(conc(\overline{nba_n}), \overline{ca})$
```

Fig. 2: Normalized µVerilog (left) and µGCL (right)

The form of a normalized µVerilog program is shown at left in Figure 2: a single clocked `always` block followed only by 0 or more continuous assignments. The program contains no unclocked `always` blocks due to applications of rewrite rule REDUCE-ASSN. Additionally, the body of the single `always` block is a single `if-else` chain with no nested conditionals.

After rewriting as described, the single `if-else` chain would contain only predicates from within clocked `always` blocks. To consider relevant predicates from other parts of the program, we collect predicates referencing the `CS` register and introduce them in empty clocked statements prior to rewriting.

Guarded Commands. We then convert the resulting normalized µVerilog program to a restricted, deterministic variant of Dijkstra's guarded command language [13] (µGCL). A µGCL program consists of a single `do` repetition with one or more mutually exclusively guarded commands, where each guarded command is a pair of a predicate and a sequence of assignments. µGCL shares the same syntax and semantics of expressions as µVerilog, but has only one operation: concurrent assignments of the form $v_1, v_2, ..., v_n = e_1, e_2, ..., e_n$, abbreviated $\overline{v} = \overline{e}$, that atomically update each $v \in \overline{v}$ with its respective element of $\overline{e}$. Exactly one guarded command is available for execution during any iteration. The restricted structure of normalized µVerilog allows translation to µGCL to proceed in a purely syntactic way. Continuous assignments are always active, so we model them in µGCL by both appending and prepending them to each guarded command in topological order (to preserve dependency order). Figure 2 shows the result of converting the normalized µVerilog at left to the µGCL at right, where *conc* converts a sequence of assignments to a single concurrent assignment, and *adjoin* performs the aforementioned operation on continuous assignments.

Given a µGCL program as a set of guarded commands G, we now wish to produce an equivalent UIR diagram. Intuitively, translation proceeds by determining the set of possible transitions between guarded commands, which we encode as a static reachability relation among the set of guarded commands.

Definition 5 (µGCL Static Command Reachability): A static command reachability relation R on guarded commands G is the set of all pairs $(g = p \to a, g' = p' \to a') \in G \times G$ such that $\exists\ \sigma.\ p(\sigma) \wedge p'([\![a]\!]_v(\sigma))$.

The static command reachability relation R overapproximates the set of possible transitions between guarded commands in G. Given R, we then construct the corresponding UIR diagram $\pi = (\mathcal{L}, \mathcal{M}, \mathcal{E}, s, \phi, \mathcal{F}, [\![\cdot]\!])$ as follows:

1) The body of each guarded command $p \rightarrow a$ receives a fresh label l such that $l \in \mathcal{L}$ and $\mathcal{M}(l) = a$.
2) For each $(p \rightarrow a, p' \rightarrow a') \in R$, let l, l' be the corresponding labels defined in Step 1. Add (l, p', l') to $\mathcal{E}$.
3) Compute the reset state σ_{reset} by performing a concrete execution of the µGCL in which `rst` is asserted to 1. Create a fresh label l_s such that $s = l_s$ and $\mathcal{M}(l_s) = init(\sigma_{reset})$ where $init(\sigma_{reset})$ simply initializes variables to their reset values. Let S be the set of all guarded commands $p \rightarrow a$ such that $p(\sigma_{reset})$ holds. If $|S| = 1$ then add tuple $(l_s, true, l)$ to $\mathcal{E}$ where l is the single element of S. If $|S| \neq 1$ then fail.
4) $\phi = true$ and $\mathcal{F} = \varnothing$.
5) $[\![\cdot]\!] = [\![\cdot]\!]_v$ where $[\![\cdot]\!]_v$ defines the state transformation semantics of µVerilog assignments and expressions.

Extraneous Control. At this point the diagram still contains extraneous control variables such as `rst` that represent hardware specific functionality with no analogue in the C specification. To extract only functionality enabled under normal operating conditions, we globally constrain the variables `rst` and `start` to constants 0 and 1, respectively, when computing the µGCL static command reachability relation.

Memory Operations. Another challenge is that memory operations still occur as low-level port assignments to enable and address wires, which we wish to lift to operations of the form $\texttt{load}(addr, mem)$ and $\texttt{store}(addr, value, mem)$.

Let $\pi = (\mathcal{L}, \mathcal{M}, \mathcal{E}, s, \phi, \mathcal{F}, [\![\cdot]\!])$ be the UIR diagram. We first determine the set of arrays referenced by detecting groups of variables of the form $arr\texttt{\_ce0}$, $arr\texttt{\_address0}$, and $arr\texttt{\_q0}$, where arr is the array name. Then, for every pair of labels $l, l' \in \mathcal{E}$ we define two logical formulas:

$$T_{l,l'} = \exists\, \sigma, \sigma'.\, P_{\rightarrow l}(\sigma) \wedge \langle l, \sigma\rangle \hookrightarrow \langle l', \sigma'\rangle \wedge (\sigma'(arr\texttt{\_ce0}) = 1)$$
$$F_{l,l'} = \exists\, \sigma, \sigma'.\, P_{\rightarrow l}(\sigma) \wedge \langle l, \sigma\rangle \hookrightarrow \langle l', \sigma'\rangle \wedge (\sigma'(arr\texttt{\_ce0}) = 0)$$

where $P_{\rightarrow l}$ is the disjunction of all predicates on incoming edges to l. Intuitively, $T_{l,l'}$ holds if there exists a trace with step from from label l to label l' along which the read enable signal $arr\texttt{\_ce0}$ has value 1, and $F_{l,l'}$ holds if there exists a trace with step from from label l to label l' along which the read enable signal $arr\texttt{\_ce0}$ has value 0. If $T_{l,l'}$ holds, then we insert a new label l_T with an operation assigning $arr\texttt{\_q0}$ to the loaded value. The incoming edge predicate to l_T correspondingly receives the additional conjunct $arr\texttt{\_ce0} = 1$ such that this transition is followed only in the case that the read occurs. Analogously, if $F_{l,l'}$ holds then we insert a new label l_F with an assignment operation that assigns port $arr\texttt{\_q0}$ with an unknown value (we model unknown values in SMT queries with unconstrained fresh variables). In other words, if $arr\texttt{\_ce0} = 0$ then no load occurs, and the value on the memory read port is undefined. Lifting `store` operations is conceptually the same, so we omit the details here.

Exit Labels. At this point, there is still no exit label in the diagram: whereas the C specification contains an explicit `return` statement, the FSM simply transitions back to the start state at termination. We simply detect such transitions and reroute them to a distinguished exit label instead.

C to UIR. Translation from C to UIR is straightforward, as a UIR diagram is just a generalized form of control flow graph. We use LLVM [14] to generate an unoptimized control flow graph for the C program, which we encode as a UIR diagram in which labels are basic blocks, transitions are control flow edges, and the start and exit labels are entry and exit nodes, respectively. Crucially, due to the differing semantics of LLVM and Verilog expressions, in this case $[\![\cdot]\!] = [\![\cdot]\!]_c$ which defines the state transformation semantics of LLVM instructions.

V. Bisimulation Relations

With UIR diagrams for both the C specification and Verilog implementation in hand, we can now describe our strategy for constructing equivalence proofs. Our equivalence proofs will take the form of *bisimulation relations*.

We say that two UIR diagrams are compatible iff they have the same input variables, return variables, and array variables. Given compatible diagrams π_1 and π_2, we define $\Psi(\pi_1, \pi_2)(\sigma_1, \sigma_2)$ to be true iff the values of input and array variables are pair-wise equal in σ_1 and σ_2; and $\Psi'(\pi_1, \pi_2)(\sigma_1, \sigma_2)$ to be true iff the values of array and return variables are pair-wise equal in σ_1 and σ_2. Ψ and Ψ' are the pre- and post-conditions of our verification.

Definition 6 (Equivalence of UIR Diagrams): Two compatible π_1 and π_2 are equivalent, written $\pi_1 \equiv \pi_2$, iff (where $\Phi = \Psi(\pi_1, \pi_2)$ and $\Phi' = \Psi'(\pi_1, \pi_2)$)

$$(\forall\, \tau_1 \in \eta_{\pi_1} \exists\, \tau_2 \in \eta_{\pi_2}.\ \tau_1 \equiv^{\Phi}_{\Phi'} \tau_2) \wedge$$
$$(\forall\, \tau_2 \in \eta_{\pi_2} \exists\, \tau_1 \in \eta_{\pi_1}.\ \tau_1 \equiv^{\Phi}_{\Phi'} \tau_2)$$

Definition 7 (Correlation Relation): A correlation relation $\mathcal{R}$ for UIR diagrams $\pi_1 = (\mathcal{L}_1, \mathcal{M}_1, \mathcal{E}_1, s_1, \phi_1, \mathcal{F}_1, [\![\cdot]\!]_1)$ and $\pi_2 = (\mathcal{L}_2, \mathcal{M}_2, \mathcal{E}_2, s_2, \phi_2, \mathcal{F}_2, [\![\cdot]\!]_2)$ is a set of tuples (l_1, l_2, ψ) called cutpoints, such that $l_1 \in \mathcal{L}_1$, $l_2 \in \mathcal{L}_2$, and $\psi : \Sigma_1 \times \Sigma_2 \rightarrow \mathcal{B}$ is a predicate relating the states of π_1 and π_2.

Definition 8 (Simulation Relation): Given compatible UIR diagrams $\pi_1 = (\mathcal{L}_1, \mathcal{M}_1, \mathcal{E}_1, s_1, \phi_1, \mathcal{F}_1, [\![\cdot]\!]_1)$ and $\pi_2 = (\mathcal{L}_2, \mathcal{M}_2, \mathcal{E}_2, s_2, \phi_2, \mathcal{F}_2, [\![\cdot]\!]_2)$, a simulation relation for π_1, π_2 is a correlation relation $\mathcal{R}$ for π_1, π_2 such that:

1) $(s_1, s_2, \phi_{pre}) \in \mathcal{R}$ where
   $\phi_{pre}(\sigma_1, \sigma_2) = \Psi(\pi_1, \pi_2)(\sigma_1, \sigma_2) \wedge \phi_1(\sigma_1) \wedge \phi_2(\sigma_2)$
2) $\forall\, (f_1, f_2) \in \mathcal{F}_1 \times \mathcal{F}_2.\ (f_1, f_2, \Psi'(\pi_1, \pi_2)) \in \mathcal{R}$
3) $\forall\, l_1, l_1', l_2, \sigma_1, \sigma_1', \sigma_2, \phi.$
   $(l_1, l_2, \phi) \in \mathcal{R} \wedge \langle l_1, \sigma_1\rangle \overset{\mathcal{R}}{\hookrightarrow}{}^{+}_{1} \langle l_1', \sigma_1'\rangle \wedge \phi(\sigma_1, \sigma_2)$
   $\implies \exists\, l_2', \sigma_2', \phi'.$
   $(l_1', l_2', \phi') \in \mathcal{R} \wedge \langle l_2, \sigma_2\rangle \overset{\mathcal{R}}{\hookrightarrow}{}^{+}_{2} \langle l_2', \sigma_2'\rangle \wedge \phi'(\sigma_1', \sigma_2')$

where $\overset{\mathcal{R}}{\hookrightarrow}{}^{+}_{1}, \overset{\mathcal{R}}{\hookrightarrow}{}^{+}_{2}$ are $\hookrightarrow^{+}_{1}, \hookrightarrow^{+}_{2}$ restricted to labels occurring in $\mathcal{R}$, respectively.

Definition 9 (Bisimulation Relation): A bisimulation relation for π_1, π_2 is a correlation relation $\mathcal{R}$ for π_1, π_2 such that $\mathcal{R}$ is a simulation relation for π_1, π_2 and $\mathcal{R}^{-1}$ is a simulation relation for π_2, π_1, where $\mathcal{R}^{-1} = \{(l_2, l_1, \phi) \mid (l_1, l_2, \phi) \in \mathcal{R}\}$.

Theorem 1: If there exists a bisimulation relation for π_1, π_2, then $\pi_1 \equiv \pi_2$.

A bisimulation relation is a witness that two UIR diagrams are equivalent: the requirements of Definition 8 can be checked mechanically using an automated theorem prover.

Benchmark	C	V	U	B
`arrayIncrement`	7	187	10	6
`sumArray`	7	155	9	9
`vectorAdd`	7	235	10	6
`scalarProduct`	9	199	10	10
`crc32`	9	211	10	8
`crossProduct`	10	290	18	37
`yuv2rgba`	10	452	21	48
`popCount`	10	143	8	13
`matrixAdd`	11	276	13	23

Benchmark	C	V	U	B
`min`	12	170	9	9
`mvMul`	13	287	36	38
`waveletTransform`	13	377	20	58
`mmMul`	17	377	28	80
`fletcher32`	19	336	18	46
`gsm_logarea`	20	233	12	16
`walshHadamard`	22	440	38	207
`sha1`	34	1157	296	476

Fig. 3: **Results. C**=lines of C, **V**=lines of Verilog, **U**=time to generate UIR (s), **B**=time to infer bisimulation (s)

VI. Inferring Bisimulation Relations

Our approach to bisimulation inference uses two mutually dependent techniques: (1) static exploration of the UIR diagrams in lock-step to infer the location of cutpoints, and (2) invocation of Daikon [12] to infer invariants at these cutpoints.

Daikon. Daikon is a tool that infers likely invariants from dynamic traces of execution: by running the program on a variety of inputs, Daikon collects traces of the values that occur at run-time, and then uses statistical techniques to infer likely invariants over variables in the program at certain program locations. We adapt this technique to pairs of programs by executing both programs on several test cases and collecting their observed run-time values (each test case produces one trace per program). We align and merge these traces, then run Daikon on the merged traces to produce likely invariants. We encapsulate this entire process in function *daikon*(l_1, l_2), which returns the set of invariants that Daikon infers for the aligned pair of program locations l_1 and l_2.

Algorithm. Our algorithm tries to find an *inferred relation*:

Definition 10 (Inferred Relation): An inferred relation is a simulation relation $\mathcal{R}$ with two additional properties:

1) For each $(l_1, l_2, \phi) \in \mathcal{R}$, ϕ is a conjunction of a subset of the invariants in *daikon*(l_1, l_2).
2) For each $(l_1, l_2, \phi) \in \mathcal{R}$, if there exists $(l_1', l_2', \phi') \in \mathcal{R}$ such that a static path exists from l_1 to l_1' and from l_2 to l_2', then that path is either finite or crosses another cutpoint.

Property 2 above makes automatic checking tractable: we reduce checking requirement 3 of Definition 8 to validity checks along only a finite set of pairs of loop-free paths.

Our algorithm starts with a relation $\mathcal{R}$ with cutpoints only for the entry and exits of π_1 and π_2, with the invariants described in requirements 1 and 2 of Definition 8. In practice, the entry point invariant states that both programs start with equal inputs and memory, and the exit point invariant asserts that upon termination they have the same output and memory. We then iteratively refine $\mathcal{R}$ using the following two steps, until either a fixed point is reached (in which case $\mathcal{R}$ is the inferred bisimulation relation), or an unfixable violation is found:

1) From an existing cutpoint $(l_1, l_2, \phi) \in \mathcal{R}$, traverse π_1 and π_2 in lock-step (while pruning out pairs of infeasible paths given precondition ϕ) until one of three conditions occurs:
   a) We hit an existing cutpoint $(l_1', l_2', \phi') \in \mathcal{R}$.
   b) We find a pair of paths that both loop on themselves, in which case we must *add* a new cutpoint (l_1', l_2', ϕ') to $\mathcal{R}$ to cut through both loops (i.e. maintain condition 2 in Definition 10). We set ϕ' to *daikon*(l_1', l_2').
   c) We find a pair of paths that don't have matched continuations (e.g. one path loops, while the other reaches exit). In this case we fail.
2) Regardless of which of (a) or (b) we came from, we have $(l_1, l_2, \phi) \in \mathcal{R}$ and $(l_1', l_2', \phi') \in \mathcal{R}$. We use the SMT solver Z3 [15] to check whether the source cutpoint's invariant ϕ is sufficient to establish the target cutpoint's invariant ϕ'. (i.e. whether requirement 3 of Definition 8 holds). If a violation is found then either:
   d) (l_1', l_2') are final labels, in which case we cannot find an inferred relation – the Daikon invariants are not strong enough to prove the post-condition or the programs are not equivalent.
   e) (l_1', l_2') are not final labels, in which case we weaken ϕ' by removing the Daikon invariants from ϕ' that Z3 cannot prove.

The above process must terminate because there are a finite number of possible candidate cutpoints to consider, and for each of them, we start from a finite set of invariants (provided by Daikon) and iteratively prune down.

VII. Evaluation

We implemented our approach in a tool called VTV, which consists of about 12.5K lines of Scala code. The underlying SMT solver is Z3.

We evaluated VTV on a variety of benchmarks from domains including linear algebra, cryptography, and image processing [16], [17], [18], each of which consists of a single C function. We compiled each function using the Xilinx Vivado HLS compiler into Verilog code, and then asked VTV to check the equivalence of the original C code and the resulting Verilog code. The only other inputs were the test inputs used by our tool to generate dynamic traces for Daikon, and well-formedness constraints on inputs when necessary. For each benchmark, very few (3-5) test inputs were needed for Daikon to infer sufficient invariants, and in almost all cases writing the inputs required no intuition about the program. Figure 3 shows our results. VTV was able to verify all benchmarks. Experiments were performed on a machine with an Intel Core i7-4510U CPU with 8GB RAM.

The verification time is dominated by two components, both shown in Figure 3: (1) the time to generate the UIR and (2) the time to infer the bisimulation relation, including the time to collect traces and run Daikon. Although in theory the UIR generation algorithm described in Section IV can produce a UIR diagram of size exponential in the number of Verilog conditionals, in practice this does not occur because most derived predicates are unsatisfiable (*e.g.,* $\texttt{CS} = 0 \ \wedge \ \texttt{CS} = 1$) and pruned via rewrite rules PRUNE-F and PRUNE-T.

Upon examining the UIR diagrams proven equivalent, we found several examples of non-trivial optimizations that had been performed, including: constant propagation and folding, basic block coalescing (combining basic blocks to make loop logic contain less jumps), loop-invariant code motion (hoisting invariant computations outside of a loop), branch hoisting (short circuting loop execution with an additional branch), branch folding (removing branches whose branch condition can be statically computed), and redundant load elimination.

Limitations. As with many other translation validation approaches [19], VTV does not currently handle aggressive optimizations that significantly alter control-flow structure (e.g. loop splitting and pipelining). In all benchmarks, no such optimizations occurred. However, to extend VTV to handle such optimizations in the future, we aim to leverage existing work that has attacked the problem from the context of software compilation [20], [19], [21]. Our approach currently works on a single function at a time. By default, Vivado compiles functions into separate modules. This lends itself to a clear strategy for extending our approach to multiple-function programs: treat each submodule interaction as a function call. The construction of bisimulation relations for the interprocedural context has been demonstrated by previous work, such as that of Pnueli and Zaks [22].

VIII. Related Work

Translation Validation. Although inspired by previous work on translation validation [5], [6], [7], [10], our approach does not employ classical techniques such as weakest preconditions to infer invariants, but instead uses statistical inference [12]. Thus, our approach is conceptually similar to work on nullspace invariant inference for program equivalence [11], which is also heuristic in nature but is limited to inferring equality constraints (which our approach is not).

HLS Verification. Bisimulation-based equivalence checking has been previously applied to the validation of scheduling in HLS tools [7], [23], [8], [24], [25]. These approaches assume the availability of a convenient intermediate representation such as CSP processes [23], CFGs [7], or FSMDs [8], [24], [25]. A key contribution of our work is a technique for equivalence-checking in the absence of such tool-provided intermediate representations. Kroening and Clarke [26] use bounded model-checking to find inconsistencies between a C program and Verilog implementation, but does not prove equivalence unless the bound can be shown exhaustive.

Replay-based equivalence checking [27], [28], [29] checks equivalence of C code and its translation by mimicking the optimizations performed by HLS, but via a certified reference flow. This approach can verify aggressive optimizations like loop pipelining [29], but relies on intermediate results to reconstruct optimizations, which our approach does not require.

Finally, DFG-based equivalence checking first converts programs to dataflow graphs [30] before checking their equivalence [31]. To handle loops, their technique uses explicit unrolling and thus only handles statically bounded loops. Our technique does not have this limitation due to UIR's ability to encode cyclical control flow.

IX. Conclusion

We have presented an approach for translation validation of HLS. Our approach establishes the equivalence of a C program and a Verilog program without requiring any intermediate results from the HLS tool. Moving forward, we intend to scale VTV to even larger programs by adding a modular analysis for handling multi-module implementations.

References

[1] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-level Synthesis: Introduction to Chip and System Design*, 1992.

[2] D. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems*, 1991.

[3] R. Gupta and F. Brewer, "High-level synthesis: A retrospective," in *High-Level Synthesis*, 2008.

[4] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *PLDI '11*.

[5] H. Samet, "Proving the correctness of heuristically optimized code," *Commun. ACM*, vol. 21, no. 7, Jul. 1978.

[6] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *TACAS '98*.

[7] S. Kundu, S. Lerner, and R. Gupta, "Validating high-level synthesis," in *CAV '08*.

[8] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar, "An equivalence-checking method for scheduling verification in high-level synthesis," in *TCAD*, vol. 27, no. 3, March 2008.

[9] T. Braibant and A. Chlipala, "Formal verification of hardware synthesis," in *CAV '13*.

[10] G. Necula, "Translation validation for an optimizing compiler," in *PLDI '00*.

[11] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken, "Data-driven equivalence checking," in *OOPSLA '13*.

[12] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1–3, Dec. 2007.

[13] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, Aug. 1975.

[14] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO '04*.

[15] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS '08*.

[16] NVIDIA, "CUDA toolkit 3.0," 2010.

[17] Rotem, Nadav, 2010, http://www.c-to-verilog.com.

[18] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *JIP*, vol. 17, 2009.

[19] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg, "VOC: A methodology for the translation validation of optimizing compilers," *Journal of Universal Computer Science*, vol. 9, 2003.

[20] ——, "VOC: A translation validator for optimizing compilers," in *COCV '02*.

[21] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu, "Translation and run-time validation of loop transformations," *Formal Methods in System Design*, vol. 27, no. 3, 2005.

[22] A. Pnueli and A. Zaks, "Translation validation of interprocedural optimizations," in *SVV '06*.

[23] S. Kundu, S. Lerner, and R. Gupta, "Automated refinement checking of concurrent systems," in *ICCAD '07*.

[24] C. Karfa, C. Mandal, and D. Sarkar, "Formal verification of code motion techniques using data-flow-driven equivalence checking," *TODAES*, vol. 17, no. 3, Jul. 2012.

[25] Y. Kim, S. Kopuri, and N. Mansouri, "Automated formal verification of scheduling process using finite state machines with datapath (FSMD)," in *ISQED '04*.

[26] D. Kroening, E. Clarke, and K. Yorav, "Behavioral consistency of C and Verilog programs using bounded model checking," in *DAC '03*.

[27] S. Ray, K. Hao, F. Xie, and J. Yang, "Formal Verification for High-Assurance Behavioral Synthesis," in *ATVA '09*.

[28] K. Hao, F. Xie, S. Ray, and J. Yang, "Optimizing equivalence checking for behavioral synthesis," in *DATE '10*.

[29] K. Hao, S. Ray, and F. Xie, "Equivalence checking for behaviorally synthesized pipelines," in *DAC '12*.

[30] A. Koelbl and C. Pixley, "Constructing efficient formal models from high-level descriptions using symbolic simulation," *Int. J. Parallel Program.*, vol. 33, no. 6, Dec. 2005.

[31] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley, "Solver technology for system-level to rtl equivalence checking," in *DATE '09*.