A-QED Verification of Hardware Accelerators

Eshan Singh[*], Florian Lonsing[*], Saranyu Chattopadhyay[*], Maxwell Strange[*], Peng Wei[§], Xiaofan Zhang[‡], Yuan Zhou[↓], Deming Chen[‡], Jason Cong[§], Priyanka Raina[*], Zhiru Zhang[↓], Clark Barrett[*] and Subhasish Mitra[*]

[*]Stanford University [§]University of California, Los Angeles [‡]University of Illinois, Urbana Champaign [↓]Cornell University

***Abstract*—We present A-QED (Accelerator-Quick Error Detection), a new approach for pre-silicon formal verification of stand-alone hardware accelerators. A-QED relies on bounded model checking -- however, it does not require extensive design-specific properties or a full formal design specification. While A-QED is effective for both RTL and high-level synthesis (HLS) design flows, it integrates seamlessly with HLS flows. Our A-QED results on several hardware accelerator designs demonstrate its practicality and effectiveness: 1. A-QED detected all bugs detected by conventional verification flow. 2. A-QED detected bugs that escaped conventional verification flow. 3. A-QED improved verification productivity dramatically, by 30X, in one of our case studies (1 person-day using A-QED vs. 30 person-days using conventional verification flow). 4. A-QED produced short counterexamples for easy debug (37X shorter on average vs. conventional verification flow).**

***Keywords*—** *Bounded Model Checking, Formal verification, Pre-silicon verification, Accelerators, QED, Quick Error Detection*

I. Introduction

Pre-silicon verification is used to detect logic design flaws (*logic bugs*) before integrated circuits (*ICs*) are manufactured. Several industrial reports highlight significant challenges associated with existing pre-silicon verification methodologies (e.g., [Foster 15]). This paper is about pre-silicon verification of stand-alone hardware accelerators. Unlike general-purpose processors, *hardware accelerators* implement a (set of) specific function(s) (e.g., encryption, 3D Rendering, or Deep Neural Network inference [Cong 17, Zhou 18, Hao 19]) and are widely used for building energy-efficient (heterogeneous) System-on-Chips (*SoCs*) [Cong 12, Cota 15]. While many publications target processor verification (too many to enumerate), very few address accelerator verification. Hardware accelerator verification remains highly challenging because: 1. Unlike processors with a detailed specification (the *ISA* or the Instruction Set Architecture), hardware accelerators often lack precise descriptions of their functionality and interfaces; 2. SoCs integrate a wide variety of functions and there can be many design variants (employing a rich diversity of design techniques) even for the same hardware accelerator function (e.g., various energy and execution time targets). Each design variant must be verified thoroughly and quickly; and, 3. Accelerator verification lacks decades of rich experience unlike processor verification.

We present A-QED (*Accelerator-Quick Error Detection*), a new formal technique for pre-silicon verification of stand-alone hardware accelerators. A-QED is inspired by Symbolic QED [Lin 15, Singh 18] (which targets designs containing processor cores) and other self-consistency techniques for processors (e.g., [Jones 96]). A-QED relies on Bounded Model Checking (*BMC*) [Clarke 01] to symbolically analyze input sequences of increasing lengths for self-consistency, i.e., whether an operation with the same inputs always results in the same outputs. A-QED targets *stand-alone* verification of hardware accelerators (i.e., A-QED does not require the accelerator to be integrated inside a larger SoC). In addition to design reuse, a stand-alone technique has several benefits, including better scalability and better bug visibility (i.e., bugs that may be difficult or impossible to reach in a specific SoC may be triggered by a short trace during stand-alone analysis).

A-QED is readily applicable for an important and commonly-used class of accelerators known as Loosely-Coupled Accelerators (*LCAs*) [Cong 12, 17, Zhou 18, Hao 19]. Unlike tightly-coupled accelerators (e.g., those directly integrated within a processor pipeline), LCAs are generally placed on the SoC's on-chip interconnection network, outside of the processor core(s). This separation provides several advantages: 1. LCAs can substantially improve energy and execution time by offloading complete tasks [Cong 12]; 2. LCAs can directly access memory with high bandwidth [Cota 15]; and, 3. LCAs can be reused more easily across different SoCs (making *independent* verification of stand-alone LCA designs crucial). Our A-QED approach in this paper targets stand-alone LCAs that perform *non-interfering operations*, i.e., operations which always generate the same result for a given input (not to be confused with combinational circuits). Many LCAs belong to this category (formal model in Sec. III).

An orthogonal trend in hardware accelerator design is the use of High-Level Synthesis (*HLS*) for design productivity. In HLS, the design is described in a high-level language (e.g., C/C++ or domain-specific language), and translated to RTL (e.g., Verilog). While A-QED can be used for both HLS and RTL designs, A-QED leverages HLS automation to considerably reduce A-QED setup time.

We applied A-QED to multiple accelerator designs: a memory-controller unit design for a *CGRA* (coarse-grained reconfigurable architecture) as well as HLS designs. The memory-controller unit design study allowed an apples-to-apples comparison of A-QED vs. conventional verification flow. Our study shows:

1. A-QED detected all bugs detected by the conventional verification flow. A-QED detected more bugs that escaped the conventional flow.
2. A-QED enabled a 30-fold improvement in verification productivity (1 person-day using A-QED vs. 30 person-days using the conventional flow), stemming from multiple aspects of A-QED: (a) A-QED does not require extensive design-specific properties or assertions or a full functional specification (that are often generated manually and are error-prone); (b) progress in BMC tools; (c) (optional) A-QED-HLS integration.
3. A-QED generated short counterexamples for easy debug: 37-fold shorter on average (6 cycles on average using A-QED vs. 224 using the conventional verification flow).

In addition, we introduce the formal basis for A-QED that enables a crisp understanding of its effectiveness as well as a thorough characterization of logic bugs detected by A-QED.

The rest of this paper is organized as follows. Sec. II provides an overview of the accelerator model targeted by A-QED in this paper. Sec. III presents a formal model of such accelerators. Sec. IV details how A-QED leverages HLS. Results are presented in Sec. V, followed by related work in Sec. VI. Sec. VII concludes this paper.

II. Accelerator Model Targeted in this Paper

Various accelerator models exist, based on various design characteristics: e.g., programmable vs. fixed-function architectures, asynchronous vs. synchronous communication with the host (e.g., processor core(s)), LCA vs. tightly-coupled [Patel 08, Cascaval 10, Cota 15]. While our general A-QED approach may be adapted to any accelerator, we focus on a specific model in this paper (formally defined in Sec. III). First, we informally explain the characteristics of accelerators that fit our model:

a) The accelerator is an LCA. (Sec. I, Fig. 1). Since an LCA is connected to the SoC interconnect, it can directly access system components such as memories.

b) A handshake protocol is used to communicate between the LCA and the host (e.g., processor core). This protocol must define when the inputs to/outputs from the accelerator are valid, and also when the accelerator and the host are each ready to receive inputs.

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

c) The LCA execution is *non-interfering*; i.e., the result produced by the accelerator for a given input is independent of any other inputs received (earlier or later). LCAs should not be confused with combinational circuits – they are complex sequential circuits.

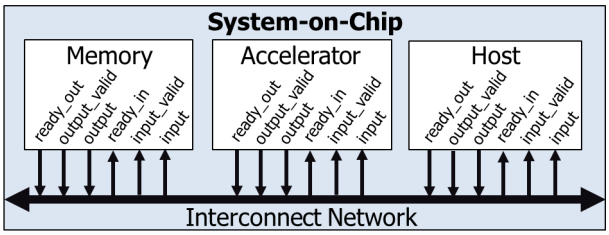

Fig. 1. Accelerator model in this paper.

A. Motivating example

We present a bug scenario to motivate A-QED. Fig. 2 shows an LCA design where four buffers forward inputs to execution units ($f(x)$), each of which takes several cycles to compute a result. Due to a bug, the *clock_enable* signal is disconnected from Buffer 4, which causes the design to produce incorrect outputs.

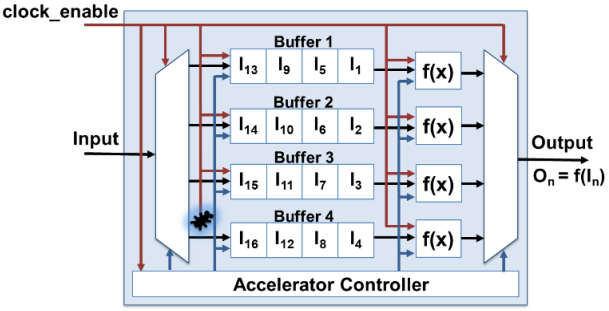

Fig. 2. Bug example: *clock_enable* disconnected from Buffer 4.

The design generates outputs O_1, O_2, …, for inputs I_1, I_2, …, where $O_i = f(I_i)$. The Accelerator Controller alternates between the four buffers (which act as input queues), shifting an input from each one in turn to the respective execution unit, if it is ready to execute it. For the 16 values loaded in the buffers (Fig. 2), the expected outputs are O_1, O_2, O_3, O_4, …, O_{16} for inputs I_1, I_2, I_3, I_4, …, I_{16}. When *clock_enable* = 0, the entire design pauses execution, maintaining its current values, and waits for *clock_enable* to go to 1. Suppose it is Buffer 4's turn to send inputs to its execution unit when *clock_enable* turns to 0. Since *clock_enable* is not connected to Buffer 4, Buffer 4 will shift I_4 to its execution unit despite *clock_enable* = 0. However, since the execution unit is disabled, it will not capture I_4. In the next cycle, when *clock_enable* returns to 1, the Accelerator Controller will erroneously send I_8 to the execution unit corresponding to Buffer 4. The generated outputs are O_1, O_2, O_3, and the incorrect $\boldsymbol{O_8}$. The bug can only be detected if the design is disabled on a cycle when it is Buffer 4's turn to shift out, and when Buffer 4 is not empty and not currently waiting for the execution unit to finish.

III. Formal Model

In this section, we formalize the class of accelerators described informally above. We define a transition system capable of modeling a general set of accelerators. Such transition systems implicitly include a clock signal to synchronize transitions between system states [Keller 76]. We formalize the properties that we expect to hold for the accelerators we target in this paper. We also show that these properties imply *total correctness*, i.e., informally, for a given input, an accelerator eventually produces a correct output.

In Fig. 3, we provide an overview of our accelerator model. Let $B = \{\bot, \top\}$ be the set of Boolean values. An accelerator *Acc* receives three inputs, an action, data, and the host ready signal. It generates two outputs, the result of an action and data, and the input ready signal. We define an accelerator formally as follows.

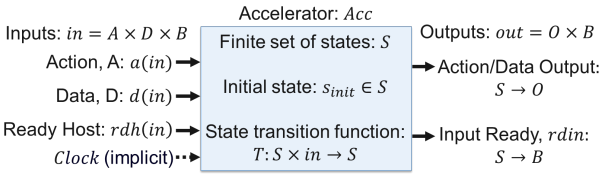

Fig. 3. Accelerator transition system model (Def. 1).

Definition 1: An accelerator is a finite state transition system $Acc := (S, s_{init}, rdin, A, a_\bot, D, O, o_\bot, T, F)$, where

- S is a finite set of states of the design;
- $s_{init} \in S$ is the initial state;
- $rdin: S \to B$ is the input-ready predicate which indicates whether the accelerator is in a state that is ready to accept an input;
- A is a finite set of actions supported by the accelerator, each specifying an operation to perform on the corresponding data input to produce an output;
- $a_\bot \in A$ is a distinguished element of A used to indicate that no operation is being selected or that the provided input is not valid;
- D is a finite set of data inputs;
- O is a finite set of outputs;
- $o_\bot \in O$ is a distinguished element of O used to indicate that no output is being produced or that the output produced is not valid;
- $T: S \times A \times D \times B \to S$ is the state transition function;
- $F: S \to O$ is the output function for action and data inputs.

An accelerator *Acc* starts in the initial state s_{init}. The execution of *Acc* is determined by a sequence of *inputs* from the set $I = A \times D \times B$. Each input $in \in I$ includes an action (i.e., a function or operation), the input data, and a Boolean value which is the *host ready* signal, representing whether the host is ready to accept any output being produced in the current state. We write $a(in)$, $d(in)$, and $rdh(in)$ for the first (action), second (data), and third (host ready) elements of an input in, respectively. Sometimes, it is useful to focus on just the action and data, which we denote by $ad(in) = (a(in), d(in))$. Given a state s and an input in, the next state is given by $s' = T(s, a(in), d(in), rdh(in))$, which we also write as $s' = T(s, in)$. At each state s, *Acc* produces an output $O = F(s)$ and the input ready predicate, $rdin$.

We use $\boldsymbol{v}$ to denote a sequence with elements denoted v_i and length $|\boldsymbol{v}|$, so $\boldsymbol{v} = \langle v_1, \ldots, v_{|\boldsymbol{v}|} \rangle$. We concatenate sequences (and with a slight abuse of notation, single elements with sequences) using '$\cdot$', e.g., $\boldsymbol{v} = v_1 \cdot \boldsymbol{v'}$, where $\boldsymbol{v'} = \langle v_2, \ldots, v_{|\boldsymbol{v}|} \rangle$.

Let $\boldsymbol{in}$ be a sequence of inputs with $|\boldsymbol{in}| = k$. From a state s_0, the sequence $\boldsymbol{in}$ induces a sequence $\boldsymbol{s}$ of states of the same length such that $s_i = T(s_{i-1}, in_i)$ for $1 \le i \le k$. We abbreviate this as $\boldsymbol{s} = T(s_0, \boldsymbol{in})$. We lift functions on sequence elements to functions over sequences in the natural way: e.g., $F(\boldsymbol{s}) = F(s_1) \cdot \ \cdots \ \cdot F(s_k)$.

Because it is relevant to the case studies we present later, we use a ready-valid protocol: only inputs in with $a(in) \neq a_\bot$, sent in a state s where $rdin(s)$ holds, are considered inputs provided to the accelerator; furthermore, only outputs different from $o_\bot$ provided when $rdh(in)$ holds are considered part of the output sequence produced. Formally, suppose that $\boldsymbol{s} = T(s_0, \boldsymbol{in})$, with $|\boldsymbol{in}| = k$. The sequence of *captured inputs*, $C_{in}(s_0, \boldsymbol{in})$ is the subsequence of $\boldsymbol{in}$ obtained by keeping only elements in_i where $a(in_i) \neq a_\bot$ and $rdin(s_i)$ holds. Similarly, the sequence of *captured outputs*, $C_{out}(s_0, \boldsymbol{in})$ is the subsequence of $F(s_0 \cdot \boldsymbol{s})$ such that $F(s_i) \neq o_\bot$ and $rdh(in_i)$ holds (where $0 \le i < k$). Note that $d(in)$ is ignored if $a(in) = a_\bot$, and the value of $rdh(in)$ is independent. We next formalize two general properties for the class of accelerators we target: functional consistency and responsiveness.

A. Functional Consistency (FC)

When applying A-QED in practice, we do not check the outputs produced by an accelerator against a formal specification (which

may or may not be available). Instead, we check whether its output function has the property of *functional consistency* (*FC*) in a mathematical sense: if we provide the accelerator with a sequence of valid inputs, then we expect it to produce a sequence of valid outputs such that the output values are the same whenever the corresponding input values are the same. This property captures the notion of non-interfering execution (Sec. II). Note that, unlike equivalence checking, we check the design against itself at different times, removing the need for a specification or a golden model.

Definition 2: An accelerator Acc is *functionally consistent* if for all input sequences $\boldsymbol{in}, \boldsymbol{in}'$, if

- $\boldsymbol{in^v} = C_{in}(s_{init}, \boldsymbol{in})$ with $|\boldsymbol{in^v}| = k$ and
- $\boldsymbol{o^v} = C_{out}(s_{init}, \boldsymbol{in} \cdot \boldsymbol{in}')$ with $|\boldsymbol{o^v}| \geq k$,

then $\forall\, 1 \leq i < k.\, ad(in_i^v) = ad(in_k^v) \rightarrow o_i^v = o_k^v$

Here $\boldsymbol{in}'$ provides the additonal inputs needed to generate at least k outputs. Note that, a functionally consistent accelerator should never produce an output *before* receiving the corresponding input. Indeed, for the class of accelerators we target, producing an output before receiving an input would be considered a bug (and such an accelerator would almost certainly also fail the functional consistency property).[1]

As detailed in Sec. IV, A-QED leverages BMC to check whether an accelerator is functionally consistent. To this end, it systematically (but implicitly using a symbolic representation) enumerates all possible input sequences of increasing length to find a counterexample to functional consistency. By Def. 2, a counterexample consists of a captured input sequence $\boldsymbol{in^v}$ of length k with corresponding captured output sequence $\boldsymbol{o^v}$, such that for some i, $ad(in_i^v) = ad(in_k^v)$ but $o_i^v \neq o_k^v$.

B. Accelerator Response Bound (RB)

Here, we formalize the notion of an *Accelerator Response Bound* (*RB*) which requires that if the host wants to provide an input or is waiting for an output, the accelerator cannot delay it forever. An example of a bug that fails this check would be one where the accelerator fails to return the result of an action due to an internal resource conflict (e.g., a deadlock) [Bayazit 05].

For a sequence $\boldsymbol{b}$ of Boolean values, let $\top(\boldsymbol{b})$ denote the subsequence consisting of exactly those elements of $\boldsymbol{b}$ equal to $\top$. We define the following responsiveness property.

Definition 3: Accelerator *Acc* is *responsive with bound n* if

(1) for every input sequence $\boldsymbol{in}$, if

- $|\top(rdin(T(s_{init}, \boldsymbol{in})))| = k$, then
- $\exists\, n. \forall\, \boldsymbol{in}'.\ |\boldsymbol{in}'| > n \rightarrow |\top(rdin(T(s_{init}, \boldsymbol{in} \cdot \boldsymbol{in}')))| > k$;

and

(2) for every input sequence $\boldsymbol{in}$, if

- $|C_{in}(s_{init}, \boldsymbol{in})| = k$ then
- $\exists\, n. \forall\, \boldsymbol{in}'. \left|\top\left(rdh(\boldsymbol{in}')\right)\right| > n \rightarrow |C_{out}(s_{init}, \boldsymbol{in} \cdot \boldsymbol{in}')| \geq k.$

The first part of the definition states that after some arbitrary input sequence, if the number of times input-ready has been true is k, then after some fixed finite number of additional inputs, the number of times input-ready has been true must be greater than k, i.e., it has been true at least one more time since the first k. This ensures the accelerator cannot starve the host by never allowing it to send an input. The second part states that after some arbitrary input sequence, if the number of captured inputs in that sequence is k, then after some fixed finite number of inputs $\boldsymbol{in}'$ where the host ready signal is true, all k of the outputs corresponding to the original input sequence have been produced. This prevents the accelerator from delaying indefinitely the production of outputs for any inputs in $\boldsymbol{in}$.

C. Total Correctness

Note that, the FC and RB properties we have defined are universal in that we expect them to hold for all accelerators in the class we are targeting. They do not rely on a formal specification of the accelerator. The only design parameter required is the accelerator response bound. This is intentional, as the main focus of A-QED is to target these universal properties, making it applicable to designs even without a formal specification. As a consequence, these properties do not cover all functional bugs (e.g. an input that consistently results in the wrong output). However, given a specification, the additional checks required are straightforward, as discussed in this section.

For completeness, we now formalize the notion of correctness with respect to a specification and illustrate to what extent this notion of correctness is covered by the properties introduced so far.

Definition 4: Given an accelerator *Acc*, let $Spec: A \times D \rightarrow O$ be a *specification function*, which for all action-data pairs (a, d), $a \neq a_\perp$, defines the expected output $Spec(a, d) \in O$ that the output function F of *Acc* is expected to produce.

Note that, the specification function *Spec* is independent of what state the accelerator is in. This is consistent with the non-interfering class of accelerators we are targeting, as discussed in Sec. II.

Definition 5: An accelerator *Acc* is *functionally correct* with respect to a specification *Spec* if for all input sequences $\boldsymbol{in}, \boldsymbol{in}'$, if

- $\boldsymbol{in^v} = C_{in}(s_{init}, \boldsymbol{in})$ with $|\boldsymbol{in^v}| = k$ and
- $\boldsymbol{o^v} = C_{out}(s_{init}, \boldsymbol{in} \cdot \boldsymbol{in}')$ with $|\boldsymbol{o^v}| = k$,

then $o_k^v = Spec(ad(in_k^v))$.

The definition is very similar to FC, except that the output for the last input must match its specification instead of being consistent with the output from any previous equivalent input.

Definition 6: An accelerator *Acc* is *totally correct* with respect to a specification *Spec* if it is functionally correct with respect to *Spec* and responsive with a given bound.

If an accelerator is functionally consistent and responsive with a given bound, this does not ensure that it is functionally correct with respect to a specification. To close the gap, we introduce the notion of *single-action correctness (SAC)*. Let $in_\perp = (a_\perp, d_\perp, \top)$, where $d_\perp$ is some arbitrary but fixed data input and let $(in_\perp)^k$ be a sequence of k repetitions of $in_\perp$.

Definition 7: An accelerator *Acc* is *single-action correct* with respect to a specification *Spec* if for every action-data pair (a, d), $a \neq a_\perp$, and input sequence $\boldsymbol{in}$, if k is the smallest value such that

- $\boldsymbol{in} = (a, d, \perp) \cdot (in_\perp)^k$ and
- $\boldsymbol{o^v} = C_{out}(s_{init}, \boldsymbol{in})$ with $|\boldsymbol{o^v}| = 1$,

then $o_1^v = Spec(a, d)$.

If a valid input is provided in the initial state, followed by a fixed sequence of invalid inputs until the output comes back, then the output for the input matches the specification. Note that we assume here (without loss of generality) that $rdin(s_{init})$ holds. We need one more definition before stating the main result of this section.

Definition 8: An accelerator *Acc* is *strongly connected* if for every $\boldsymbol{in}$, there exists $\boldsymbol{in}'$ such that if $\boldsymbol{s} = T(s_{init}, \boldsymbol{in} \cdot \boldsymbol{in}')$ with $|\boldsymbol{s}| = k$, then $s_k = s_{init}$.

Intuitively, an accelerator is strongly connected if it is possible to get from any reachable state back to the initial state.[2] We now state the main result of this section.

Proposition 1: *If a strongly connected accelerator Acc is functionally consistent, responsive with some finite bound, and single-action correct with respect to Spec, then it is totally correct with respect to Spec.*

Proof Sketch: Given $\boldsymbol{in}, \boldsymbol{in}'$, $\boldsymbol{in^v}$, and $\boldsymbol{o^v}$ as in Definition 5, we can, by single-input correctness, construct an input sequence starting with $ad(in_k^v)$ whose first output is $Spec(ad(in_k^v))$. Then, by strong connectedness, we can append to this input sequence to get back to s_{init}. Finally, appending $\boldsymbol{in} \cdot \boldsymbol{in}'$ results in a sequence where we can

---

[1] In practice, we strengthen the functional consistency property to also require that no output occurs before its corresponding input.

[2] If an accelerator is not strongly connected, the accelerator model can be extended with a reset signal that takes it back to s_{init}, and the notion of a valid input sequence can be extended to require that the reset signal can be applied at any point as long as an equal number of valid inputs and valid outputs have been seen. The remaining definitions and results then hold for the modified model.

conclude that $o_k^v = Spec(ad(in_k^v))$ by functional consistency with the first input/output pair.

Proposition 1 shows that all functional bugs are covered by checking functional consistency, responsiveness with a given bound, and single-action correctness. Though SAC is not our focus, given the *Spec* function, we can easily extend A-QED to perform SAC. Single-action bugs are typically not as challenging to find and are usually caught by standard verification techniques (e.g., [Reid 16]). A-QED is thus theoretically complete, though in practice it is limited by the scalability of the BMC tool.

IV. A-QED Setup

A-QED uses BMC to detect bugs. BMC takes as inputs a model (the design) and a set of properties, and symbolically analyzes input sequences to search for counterexamples to the properties. For BMC, we need to know how to apply legal (symbolic) inputs to the accelerator, how to analyze the accelerator outputs, and what to check to evaluate a property. We focus on FC and RB here. As noted in Sec. III, SAC is not our focus. This section details A-QED setup when leveraging HLS. While HLS makes A-QED easy to use, it is not required. A-QED can be applied for RTL designs as well, but that requires additional time and effort.

A. Setup with High-Level Synthesis

A high-level description of an accelerator (e.g., in C++) defines its operation within a function prototype. The variables of the prototype, provided as values or references (i.e., pointers), define the arguments passed to a call of the function. Hence, the result of each operation executed by the accelerator function depends only on these arguments (assuming no global variables). The arguments not only contain inputs but also (references to) variables where the result is stored. From the function definition, the inputs and outputs of the accelerator can be identified as: a) input variables, which become symbolic inputs for BMC; and b) results of the function call (values returned, plus updated variables passed as references) which are used for property checking. Constraints on the valid range of input values for the accelerator can be directly obtained from the valid range of each high-level input.

B. FC Checking

To apply A-QED to an accelerator, we generate the corresponding A-QED module. For the LCA model (Sec. II), the A-QED module interfaces with the accelerator during BMC – this enables stand-alone accelerator verification. For high-level designs, A-QED leverages HLS to not only synthesize the A-QED module (for BMC), but also to connect various signals between the accelerator and the A-QED module. **The A-QED module is used during pre-silicon verification only – it is not included in the final design.**

The A-QED module (Fig. 4) contains two functions. The first, *aqed_in,* monitors input sequences to the accelerator. It labels a certain input as I_{orig} or the "original." At some later point in the input sequence, BMC issues the same original I_{orig} to the accelerator again and A-QED labels it as "duplicate" or I_{dup}. The exact positions I_{orig} and I_{dup} in the input sequence are controlled by the BMC tool. The second function, *aqed_out*, analyzes the outputs produced by the accelerator (to check for FC).

An accelerator (especially an LCA) might receive multiple inputs in a batch (that may be processed by the accelerator in parallel). As long as the execution is noninterfering (Sec. II), such an accelerator is still a valid instance of the model in Sec. III. A-QED can then analyze single- (*batch size* = 1) or multiple-input (*batch size* > 1) batches. I_{orig} and I_{dup} can belong to the same or different (single- or multiple input-) batches.

In Fig. 4, *bmc_mem* is a global memory for accelerator inputs and outputs (the BMC places symbolic values in this memory). A detailed explanation of the pseudo-code in Fig. 4 is available in [RESULTS 20].

```
#PARAMETER MAX_BATCH_SIZE
#PARAMETER MAX_BATCH_COUNT
#PARAMETER IN_SIZE
#PARAMETER OUT_SIZE

\\ Data type definition
result {dup_done; fc_check; orig_labeled; orig_done};

\\ Pseudo-code assumes accelerator function with 2 inputs
\\ and 2 outputs, i.e., IN_SIZE = OUT_SIZE = 2

\\ Initialization of global state variables
orig_val[IN_SIZE] ← 0; orig_out[OUT_SIZE] ← 0; ORIG_BATCH
← FF; DUP_BATCH ← FF;
orig_labeled ← 0;  dup_labeled ← 0;
batch_ct ← 0; out_batch_ct ← 0; ORIG_IDX ← 0; DUP_IDX ← 0;
orig_done ← 0; mem_ptr ← 0;
dup_done ← 0; fc_check ← 0;

\\ Global Memory
bmc_mem[MAX_BATCH_SIZE*IN_SIZE*MAX_BATCH_COUNT]
\\ Pseudo-code assumes accelerator writes output result
\\ at the corresponding input location (in bmc_mem)

aqed_in (mem2acc, size, is_orig, is_dup, orig_idx, dup_idx) {
  label_orig ← (is_orig) && (orig_idx < size) && !orig_labeled;
  label_dup ← (is_dup) && (dup_idx < size) && !dup_labeled &&
((orig_labeled && (*(mem2acc + dup_idx*IN_SIZE) == orig_val[0]) &&
(*(mem2acc + 1 + dup_idx*IN_SIZE) == orig_val[1])) || (label_orig &&
(*(mem2acc + orig_idx*IN_SIZE) == *(mem2acc + dup_idx*IN_SIZE)
&& *(mem2acc + 1 + orig_idx*IN_SIZE) == *(mem2acc + 1 +
dup_idx*IN_SIZE))));
  if (label_orig) {
    orig_labeled ← 1; orig_val[0] ← *(mem2acc +
orig_idx*IN_SIZE); orig_val[1] ← *(mem2acc + 1 +
orig_idx*IN_SIZE);
    ORIG_BATCH ← batch_ct; ORIG_IDX ← orig_idx;}
  if (label_dup) {
    dup_labeled ←1; DUP_BATCH ← batch_ct; DUP_IDX
← dup_idx;}
  batch_ct ← batch_ct + 1; }

aqed_out (acc2mem) {
   orig_done ← orig_labeled && (out_batch_ct >= ORIG_BATCH);
    if (orig_done && (out_batch_ct == ORIG_BATCH) &&
!dup_done) {
     orig_out[0] ← *( acc2mem  + ORIG_IDX*OUT_SIZE);
     orig_out[1] ← *( acc2mem  + 1 + ORIG_IDX*OUT_SIZE);}
   if (orig_labeled && dup_labeled && (out_batch_ct ==
DUP_BATCH) &&  !dup_done) {
     dup_done ← 1;
     dup_0 ← *(acc2mem   + DUP_IDX*OUT_SIZE);
     dup_1 ← *(acc2mem  + 1 + DUP_IDX*OUT_SIZE);
     fc_check ← ((orig_out[0] == dup_0) && (orig_out[1] ==
dup_1)); }
   if (out_batch_ct > DUP_BATCH) {
     dup_done ← 1; }
   out_batch_ct ← out_batch_ct + 1;
return (dup_done, fc_check, orig_labeled, orig_done); }

aqed_top(batch_size, is_orig, orig_index, is_dup, dup_index) {
 result output;
 current_batch ←
&bmc_mem[MAX_BATCH_SIZE*IN_SIZE*mem_ptr];
 aqed_in(current_batch, batch_size, is_orig, is_dup, orig_index,
dup_index);
 acc(current_batch, batch_size);
 output ← aqed_out(current_batch);
 mem_ptr ← mem_ptr + 1;
return output; }
```

Fig. 4. Pseudo code for A-QED functions targeting FC. Actual implementations will vary depending on the accelerator and the HLS tool used, see [RESULTS 20].

To check for FC, the BMC tool searches for a counterexample to the following property:

$$dup\_done \rightarrow fc\_check$$

In Fig. 4, *dup_done* is true if outputs for both I_{orig} and I_{dup} have been generated, and *fc_check* is true if both these outputs match.

Some accelerator designs may require further A-QED module customization. For instance, an AES implementation (in Sec. V.B) uses a common key across an input batch. Details of such customization can be found in [RESULTS 20].

C. RB Checking

Checking for RB involves monitoring signals related to the ready-valid protocol, which can be synthesized together with the A-QED module using HLS: input-ready ($rdin$) and host-ready (rdh), and the related sequences of captured inputs ($C_{in}(s_0, \boldsymbol{in})$) and outputs ($C_{out}(s_0, \boldsymbol{in})$) for an input sequence $\boldsymbol{in}$ received in some state s_0.

A counterexample to RB is a counterexample to either part (1) or (2) of Def. 3 (Sec. III). Checking for part (1) is simple: check that signal $rdin$ does not stay low indefinitely. A counterexample to part (2) states that, after the accelerator has received k valid inputs starting from the initial state ($|C_{in}(s_{init}, \boldsymbol{in})| = k$), it fails to produce the expected k valid outputs regardless of how many times the host is ready (rdh is high) to accept outputs. To check for part (2), the BMC tool searches for counterexamples to the following property:

$$(cnt\_rdh \geq \tau) \wedge (cnt\_in \geq in\_min) \rightarrow rdy\_out$$

Parameters τ and in_min are design-specific constants and cnt_rdh and cnt_in are auxiliary signals to monitor the host-ready signal and captured inputs. The value of *cnt_rdh* is the number of cycles the host has been ready (rdh is high) to accept an output since it sent a certain input I. The value of *cnt_in* is the number of inputs captured by the accelerator since it captured input I. Parameter τ is the expected maximum number of cycles the accelerator takes to produce the output for a given input. It is a concrete implementation of parameter n in our formal model. In practice, some accelerators require more than one input to be provided before producing any outputs. This is handled by setting parameter *in_min* (this low-level detail was omitted from the formal model for simplicity, but it can easily be added). Finally, property *rdy_out* holds if the output for input I has been generated (and if further design-specific conditions hold, if any). We check whether *rdy_out* holds in the cycle where the preconditions hold, i.e., the host allowed the accelerator a sufficient number of cycles to produce the output for input I ($cnt\_rdh \geq \tau$) and it provided the accelerator with a sufficient number of captured inputs ($cnt\_in \geq in\_min$). If *rdy_out* does not hold given these preconditions, then the accelerator is unresponsive with bound τ.

V. Results

We demonstrate the effectiveness of A-QED for various designs (details in [RESULTS 20]). We did not artificially inject bugs. All A-QED results were generated using Cadence JasperGold version 2016.09p002 on an Intel Xeon E5-2640 v3 with 128GB of DRAM.

A. Memory-Controller Unit Case Study

We present a case study for a memory-controller unit (~17,000 flip-flops, ~97,600 gates) targeting CGRA-based accelerators. We had unique access to a tracked repository of various versions (SystemVerilog RTL) of this design (and bugs detected for each version using conventional verification). This allows an apples-to-apples comparison of A-QED vs. conventional verification flow.

The memory-controller unit supports several configurations (e.g., double buffer, line buffer, FIFO). The conventional simulation-based flow verified each configuration separately using well-crafted test patterns and full-fledged applications (e.g., point-wise multiplication, dilated convolution).

We applied A-QED for each configuration (except three, which involved interfering operations not supported by our accelerator model, as explained earlier). We created working C++ models for each configuration in consultation with the designers. For each such C++ model, we created A-QED module C++ functions (Fig. 4) and used HLS (Catapult) to generate the A-QED module RTL. For each configuration, we instantiated an RTL wrapper containing its A-QED module and the memory-controller (with its configuration bits hard-coded). For A-QED setup customization for this design, e.g., when a configuration does not provide a *ready* signal for the *ready-valid* protocol (Sec. III) or when *in_min* needs to be incorporated for RB checking (Sec. IV.C), please refer to [RESULTS 20].

The results of this study are presented in Table 1 and Fig. 5.

Observation 1: *A-QED significantly improved bug coverage vs. conventional verification flow.* With BMC as its backbone, A-QED finds the shortest sequence to trigger and detect bugs (within the BMC bound). This is in sharp contrast to conventional verification flows, where testbenches are highly dependent on the expertise of verification engineers. For the memory-controller unit, A-QED detected all logic bugs (Fig. 5) detected by the conventional verification flow for the studied configurations. A-QED uniquely detected additional (13%) bugs that were not detected by the conventional flow. These additional bugs represent difficult corner-case scenarios. For example, one such bug (triggered by a complex condition) caused a crash (after 70 cycles) during an application run (after the design was verified using conventional flow). In contrast, A-QED detected it in 1 second with a 6-cycle counterexample. A-QED detected one bug using RB and the remaining using FC.

Observation 2: *A-QED (with HLS support) dramatically improves verification productivity.* As Table 1 shows, the setup effort improves (i.e., reduces) 30-fold: 1 person-day using A-QED vs. 1 person-month using conventional verification flow.

Observation 3: *A-QED detects bugs quickly (⩽ 2 sec. runtime) with short counterexamples (*nearly 40-fold shorter on average vs. conventional verification flow*) enabling quick debug.*

Table 1: A-QED results for the memory-controller unit.

Verification Flow	Setup Effort* (person-days)	Runtime (seconds) [min, avg, max]	Trace (clock cycles) [min, avg, max]
A-QED	1	0.8, 1.2, 1.5	4, 6, 8
Conventional	30	378, 615, 734	51, 224, 321

*For conventional verification flow, setup includes the time to create the software functional model as well as testbenches.

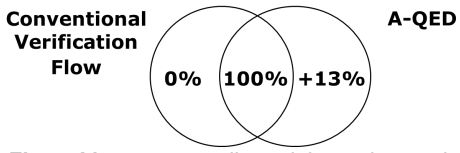

Fig. 5. Memory-controller unit bugs detected.

B. HLS designs

Table 2 presents A-QED results for some HLS LCA designs.

Table 2: A-QED results for HLS designs. (CEX = Counterexample)

Source	(Buggy) designs*	Bug	Runtime (min: sec)	CEX length (cycles)
AES encryption [Cong 17]	AES v1	FC	1:12	136
	AES v2		4:11	290
	AES v3		0:57	132
	AES v4		0:06	94
Custom design [Chi 19]	Dataflow	RB	0:28	98
Rosetta [Zhou 18]	Optical Flow		0:27	197
CHStone [Hara 09]	GSM	FC	4:05	65

* Abstracted versions in [RESULTS 20].

The Vivado HLS tool was used for these designs. Examples of bugs detected include various array indexing errors and incorrect FIFO sizing. Details on abstracted versions of the designs used in Table 2 (for BMC scalability) as well as A-QED module customization (e.g., to support a common key across a batch for AES) can be obtained from [RESULTS 20]).

Observation 4: *A-QED successfully detected bugs in various HLS accelerators.* FC bugs (across different designs) were detected using the same FC universal property (Sec. III).

VI. Related Work

There are numerous publications on pre-silicon verification: simulation-based approaches, formal approaches, and combinations thereof. Many of these publications focus on processor cores. The difficulties with verification of stand-alone hardware accelerators vs. processor verification are highlighted in Sec. I and are also well-explained in [Huang 18]. In contrast, A-QED enables verification of stand-alone hardware accelerators in an effective and practical way (especially in the context of HLS designs). As discussed in Sec. I, A-QED is inspired by Symbolic QED [Lin 15, Singh 18], which targets designs containing processor cores. However, there are important differences between A-QED vs. Symbolic QED: (a) A-QED targets stand-alone accelerators without any processor core; (b) A-QED leverages HLS to simplify the setup process; and (c) A-QED does not require a partitionable register file or memory for accelerators supporting the formal model in Sec. III.

A-QED is applicable to both RTL and HLS designs. For RTL, A-QED can leverage the Instruction-Level Abstraction (*ILA*) approach [Huang 18] to further improve verification productivity.

Unlike conventional BMC, A-QED does not require design-specific properties or a full specification (that are often created manually). While some tools try to automate formal property generation, it is often difficult to identify the "right" set. Attempts to generate design-specific properties from a specification can be problematic, since a complete specification may not be available (or the specification itself may be buggy [Singh 19]). As part of A-QED, checking for SAC may be necessary – however, it doesn't require a complete specification and can be largely automated (e.g., techniques in [Reid 16]). Finally, A-QED can be used in conjunction with many (commercial and academic) BMC engines.

A-QED is also distinct from simulation-based pre-silicon verification approaches including those that leverage HLS (e.g., [Campbell 19, Chi 19]: A-QED uses BMC, and therefore (a) is significantly more thorough, and (b) finds short counterexamples. As is well-known, BMC-based techniques can face scalability challenges (e.g., design size, BMC bound). This aspect of A-QED is discussed in Sec. VII.

VII. Conclusion

A-QED is a highly effective and practical approach for pre-silicon verification of stand-alone hardware accelerators. It leverages BMC but bypasses major BMC challenges (e.g., creation of design-specific properties). A-QED is especially attractive for HLS-based accelerator designs because it largely automates the verification setup process while avoiding extensive (manual) efforts in understanding RTL designs.

A-QED creates several promising research directions: 1) Extension of A-QED beyond the LCA model (including the handshake protocol); 2) Verifying designs that execute interfering operations (and not just non-interfering ones); 3) Improving the scalability of A-QED through techniques such as abstraction (e.g., [Andraus 04]), concolic execution (e.g., [Sen 05]), and symbolic starting states (e.g., [Fadiheh 18] for processor cores); 4) More detailed case studies to demonstrate the effectiveness of A-QED; and, 5) A-QED for post-silicon validation and debug. The use of A-QED-inspired techniques for accelerator hardware security (similar to Symbolic QED-inspired techniques for detecting security vulnerabilities in processors [Fadiheh 19]) is another interesting direction for future work.

Acknowledgement

This work was supported in part by the DARPA POSH program.

References

[Andraus 04] Andraus, A. A., and K. Sakallah, "Automatic abstraction and verification of verilog models," *Proc. DAC*, 2004.

[Bayazit 05] Bayazit, A. A., and S. Malik, "Complementary use of runtime validation and model checking," *Proc. ICCAD*, 2005.

[Campbell 19] Campbell, K., et al., "Hybrid Quick Error Detection: Validation and Debug of SoCs Through High-Level Synthesis," *IEEE Trans. CAD*, 2019.

[Cascaval 10] Cascaval, C., *et al.*, "A taxonomy of accelerator architectures and their programming models," *IBM Journal of Research and Development*, 2010.

[Chi 19] Chi, Y., *et al.*, "Rapid Cycle-Accurate Simulator for High-Level Synthesis," *Proc. Intl. Symp. FPGAs*, 2019.

[Clarke 01] Clarke, E., *et al.*, "Bounded Model Checking using Satisfiability Solving," *Formal Methods in System Design*, 2001.

[Cong 12] Cong, J., *et al.*, "Architecture support for accelerator-rich CMPs," *Proc. DAC*, 2012.

[Cong 17] Cong, J., *et al.*, "Bandwidth Optimization Through On-Chip Memory Restructuring for HLS," *Proc. DAC*, 2017.

[Cota 15] Cota, E. G., *et al*., "An Analysis of Accelerator Coupling in Heterogeneous Architectures," *Proc. DAC*, 2015.

[Fadiheh 18] Fadiheh, M. R., *et al.*, "Symbolic quick error detection using symbolic initial state for pre-silicon verification," *Proc. DATE*, 2018.

[Fadiheh 19] Fadiheh, M. R., *et al.*, "Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking," *Proc. DATE*, 2019.

[Foster 15] Foster, H. D., "Trends in Functional Verification: A 2014 Industry Study," *Proc. DAC*, 2015.

[Hara 09] Hara, Y., *et al.*, "Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis," *Journal of Information Processing*, 2009.

[Huang 18] Huang, B-Y., *et al.*, "Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification," *ACM Trans. Design Automation of Electronic Systems*, 2019.

[Jones 96] Jones, R., C-J. H. Seger and D. L. Dill, "Self-Consistency Checking," *Proc. FMCAD*, 1996.

[Keller 76] Keller, R.M., "Formal Verification of Parallel Programs," *Communs. of ACM*, 1976.

[Lin 15] Lin, D., *et al.*, "A Structured Approach to Post-Silicon Validation and Debug Using Symbolic Quick Error Detection," Proc. *IEEE Intl. Test Conf.*, 2015.

[Patel 08] Patel, S., and W. Hwu, "Accelerator Architectures," *IEEE Micro*, 2008.

[Reid 16] Reid, A., *et al.*, "End-to-end verification of processors with ISA-Formal," *Proc. Computer-Aided Verification*, 2016.

[RESULTS 20] https://github.com/upscale-project/aqed-dac2020-results

[Sen 05] Sen, K., *et al.*, "CUTE: a concolic unit testing engine for C," *ACM SIGSOFT Software Engineering Notes*, 2005.

[Singh 18] Singh, E., *et al.*, "Logic Bug Detection and Localization Using Symbolic Quick Error Detection," *IEEE Trans. CAD*, 2018.

[Singh 19] Singh, E., *et al.*, "Symbolic QED Pre-silicon Verification for Automotive Microcontroller Cores: Industrial Case Study," *Proc. DATE*, 2019.

[Zhou 18] Zhou, Y., *et al.*, "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs," *Proc. Intl. Symp. FPGAs*, 2018.