G-QED: Generalized QED Pre-silicon Verification beyond Non-Interfering Hardware Accelerators

Saranyu Chattopadhyay*, Keerthikumara Devarajegowda‡, Bihan Zhao§, Florian Lonsing†,
Brandon A. D'Agostino*, Ioanna Vavelidou*, Vijay D. Bhatt§, Sebastian Prebeck§,
Wolfgang Ecker§, Caroline Trippel*, Clark Barrett†, Subhasish Mitra*
EE Dept.* and CS Dept.† Stanford University, Siemens EDA‡, Infineon Technologies§

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247903

Abstract—**Hardware accelerators (HAs) underpin high-performance and energy-efficient digital systems. Correctness of these systems thus depends on the correctness of constituent HAs. Self-consistency-based pre-silicon verification techniques, like A-QED (Accelerator Quick Error Detection), provide a quick and provably thorough HA verification framework that does not require extensive design-specific properties or a full functional specification. However, A-QED is limited to verifying HAs which are non-interfering – i.e., they produce the same result for a given input independent of its context within a sequence of inputs. We present a new technique called G-QED (Generalized QED) which goes beyond non-interfering HAs while retaining A-QED's benefits. Our extensive results as well as a detailed industrial case study show that: G-QED is highly thorough in detecting critical bugs in well-verified designs that otherwise escape traditional verification flows while simultaneously improving verification productivity 18-fold (from 370 person days to 21 person days). These results are backed by theoretical guarantees of soundness and completeness.**

Index Terms—**QED, Quick Error Detection, Accelerators, Processors, Functional consistency**

I. Introduction

Domain-specific hardware accelerators (*HAs*) are becoming increasingly crucial for high-throughput and energy-efficient digital systems. Today's digital systems, often referred to as System-on-Chips or *SoCs*, contain many HAs spanning various application domains. Each HA implements a set of functions referred to as *Actions* in this paper. HAs may be *tightly-coupled*, e.g., integrated within a processor's pipeline. More commonly though, HAs are *loosely-coupled*, interacting with other SoC components (other HAs, processor cores, memory) via on-chip networks. Given their pervasiveness [1], [2], loosely-coupled HAs (*LCAs*) are the focus of this paper although our presented techniques can be applied to tightly-coupled HAs as well.

Every HA must be verified for correctness both thoroughly *and* quickly to meet the time-to-market demands of the diverse applications they support [3]. As discussed in many prior publications [4]–[7], HA formal verification is challenged by: (1) the tremendous effort required to craft highly thorough design-specific properties and *full* functional specifications, and (2) the scalability of off-the-shelf formal tools. Beyond being time-consuming and error prone [4], [5], producing thorough properties and specifications is an uphill battle due to the rapidly evolving nature of HAs that support rapidly evolving applications.

A recent verification technique, Accelerator Quick Error Detection (*A-QED*, [6], [7]), overcomes the above challenges for a class of HAs that are *non-interfering* – i.e., HAs that produce the same output for a given action independent of its context within a sequence of actions. A-QED uses formal verification based on Bounded Model Checking (*BMC*, [8]). Unlike conventional BMC-based verification, A-QED does not require extensive design-specific properties or a *full* functional specification. Instead, A-QED uses *self-consistency* checks on a given HA. Specifically, A-QED checks for functional consistency (*FC*), the property that actions with identical inputs always produce the same outputs. In addition to FC, A-QED also performs single-action checking (*SAC*) and response bound checking (*RB*) explained later in this paper. A-QED is sound and complete for non-interfering HAs.

While non-interfering HAs readily capture a range of *fixed-function* designs, *interfering* HAs are becoming more and more prevalent. This is partly due to the rise of programmable HAs [9]. In fact, traditional processors may be viewed as an extreme case of interfering HAs where each instruction is an HA action. Interfering HAs contain *interfering* actions whose outputs *are* dependent on the outputs of other actions, inherently violating A-QED's FC checking. To complicate matters even further, an HA action might read the outputs produced by another action (or write its outputs to be consumed by another action) at clock cycles that depend on the execution of various other concurrent actions active in the HA. Thus, there is an urgent need for a new and *general* formal verification methodology for interfering (and non-interfering) HAs that preserves the benefits of A-QED (i.e., provably sound and complete verification without requiring extensive design-specific properties or full functional specifications) while reasoning about *interfering* actions (not possible using A-QED) – a highly difficult challenge.

In this paper, we overcome this challenge through *G-QED* or *Generalized QED* – a new technique for thoroughly verifying both interfering and non-interfering HAs (and processor cores by extension) – and make the following contributions:

- Given an HA with its set of actions, we present the G-QED technique to thoroughly verify that HA without requiring detailed knowledge about its design or the interfering nature of its various actions. G-QED supports both RTL and high-level synthesis design flows and integrates seamlessly with several industrial BMC tools.
- We provide a formal model of HAs, demonstrate its broad applicability, and prove soundness and completeness guarantees of G-QED for such HAs.
- We demonstrate the effectiveness and practicality of G-QED through extensive results on a wide variety of 44 moderate-sized HAs and processor cores (that fit in existing BMC tools).
- We present an industrial case study on live designs and demonstrate: (1) Significantly improved bug coverage of G-QED by uniquely detecting corner-case bugs that escaped industrial (simulation- and formal verification-based) verification flow (in addition to detecting all bugs detected by the industrial flow). (2) Dramatically

improved verification productivity of G-QED, from 370 person days using industrial flow to only 21 person days using G-QED – an 18-fold boost. (3) G-QED-enabled short design-design verification loops with quick turnaround for rapidly evolving designs.

While we focus on moderate-sized HAs that fit in existing BMC tools, there are further scaleup opportunities, e.g., new decomposition techniques enabled by self-consistency checking [7] – a topic of future work.

II. Motivating Example

The representative HA in Fig. 1 (adapted from a commercial design) is used to illustrate interfering HA verification challenges. The HA is connected to other SoC components via a handshake protocol similar to [6]. The HA only reads valid inputs (*in_valid* asserted) from the network when it is ready (*rdy_out* asserted). The network reads HA-generated outputs (*out_valid* asserted) when it is not blocked by other components (*rdy_in* asserted). The HA implements 3 actions $\{A_1, A_2, A_3\}$ as follows:

- $A_1(D)$: updates $Bypass$ register with D.
- $A_2(D)$: updates $Factor$ register with D. A_2 is stored in FIFO 2.
- $A_3(D, Bypass, Factor)$: generates an output $O = F(D)$ scaled by the $Factor$ register value. The scaling operation is skipped depending on the $Bypass$ register's value. A_3 is stored in FIFO 1. The output of this interfering action depends not only on D, but also on the values of the $Factor$ and $Bypass$ registers. The $Bypass$ and $Factor$ registers constitute the Relevant State Registers (*RSRs*) of A_3. RSRs are formally defined in Sec. IV

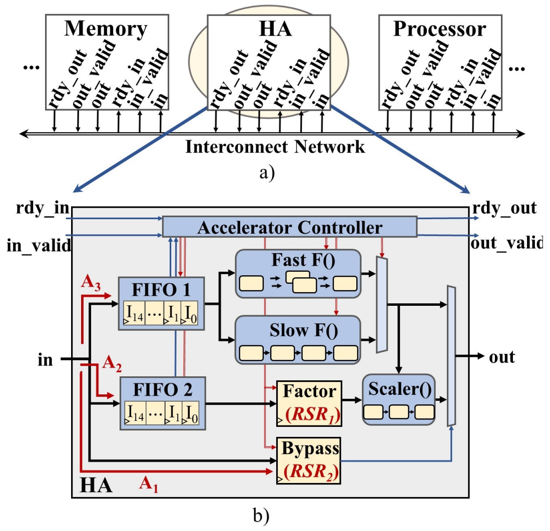

Fig. 1. a) HA *loosely-coupled* to other SoC components. b) HA example.

$F()$ and $Scaler()$ take multiple cycles to compute and pending inputs are stored in the FIFOs. If either FIFO is full, the HA switches to fast $F()$ implementation. If any of the $Scaler()$ inputs is 0, the unit is designed to skip computation for better power and performance. Thus, when the $Scaler()$ unit is bypassed, the HA updates $Factor$ register with 0.

Consider the following bug (adapted from an actual bug) - the *FIFO 1* full signal goes high only when the write pointer reaches 15 (starting from 0) but the FIFO can hold at most 15 entries. Hence, the 16^{th} A_3 overwrites its predecessor. This bug is only triggered if the *rdy_in* is low long enough. It can be detected by checking A_3 for FC. However, to perform FC, we need to constrain the RSRs to prevent false fails.

Challenge 1: The exact clock cycles when RSRs are read depend on the internal state and can be different for different RSRs. For example, consider action A_3. When the result of $F()$ is fed to the $Scaler()$, the value stored in Factor is also read. It is very important to precisely specify the clock cycle during which this value should be read. That is not an easy task because it depends on the latency of $F()$ (which in turn depends on the FIFO 1 state). Incorrect timing information can result in false fails.

Challenge 2: Consider the same bug in FIFO 2. If we constrain $Factor$ to a fixed value, A_3 will always read the same value from the $Factor$ register and pass FC check.

Challenge 3: Checking FC on A_2 is a non-trivial problem since an update can happen either from A_2 or because the HA updates it to 0 when the $Scaler()$ is bypassed. So it is not necessary that the i^{th} input will produce the i^{th} update to the $Factor$ register. Thus, we need to understand the design implementation to figure out when an input updates the register.

III. G-QED

The new FC approach is shown in Fig. 2. To check FC for the action pair $\{a, b\}$, the BMC runs three copies of the HA from the reset state. The same input sequence $I_0 \dots I_k$ is fed to the first two copies. For the first copy, the HA is allowed to finish executing all the inputs and then the RSRs for actions a and b are saved. For the second copy, input pair I_a, I_b corresponding to the action pair $\{a, b\}$ is fed to the HA and the output pair $\{O_a, O_b\}$ is recorded. For the third copy, an input sequence $I_{m+1} \dots I_n$, which is not necessarily the same as $I_0 \dots I_k$, is allowed to finish executing before sending the input pair I_a, I_b to the HA. The RSRs are saved before sending I_a, I_b and constrained to be the same as the RSR values saved in the first copy. Additionally, the output pair $\{O_a, O_b\}$ is checked with that from the second copy. The FC property is formulated as:

$$RSR_{\text{saved}}^{1^{st}\ copy} = RSR_{\text{saved}}^{3^{rd}\ copy} \rightarrow$$
$$\{O_a, O_b\}^{2^{nd}\ copy} = \{O_a, O_b\}^{3^{rd}\ copy}$$

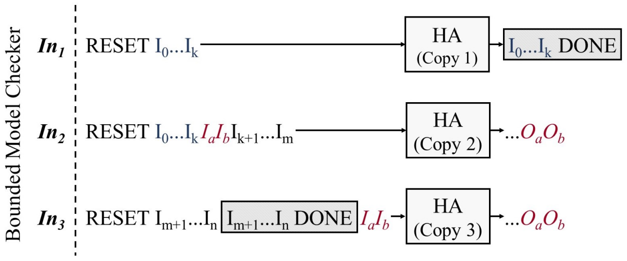

Fig. 2. FC check framework for the action pair {a, b}. The inputs $I_0 - I_n$ and the sequence lengths k, m and n are chosen symbolically. The inputs I_a and I_b are chosen symbolically with their respective actions a and b.

We assume that they RSRs used to calculate $\{O_a, O_b\}$ in the second copy have the same values as those saved in the first copy. This is elucidated in the following design constraint:

Design Constraint: If we send the input pair $\{I_a, I_b\}$ after all the inputs have finished executing in the first copy, then the output pair generated is the same as $\{O_a,\ O_b\}$ in the second copy.

This means the output generated or the RSR updated by an action in a sequence is independent of how long the design is idle in between the inputs. The HA example in Sec. II is idling when the *in_valid* is low. During this time, no more inputs are accepted by the HA but previous inputs keep executing. We have seen this constraint to hold for all the designs we looked at in Sec. V. We avoid the *Challenges 1* and *2* by constraining RSRs when no inputs are being executed.

In case of *challenge 3*, the $Factor$ value updated by A_2 can be propagated as an output of a future A_3 action. Thus, we address *challenge 3* by FC for the action pair $\{A_2, A_3\}$ instead of checking FC for A_2 action.

Pair wise checking of actions allows us to find bugs in the RSR updating logic since checking the RSRs directly is non-trivial for a general HA as discussed in *Challenge 3* in Sec. II. However, this is not the case for processors, so we consider all the RSRs as the output of every instruction. Thus, we don't have to check instructions in pairs for processors.

To catch the FIFO 2 bug, BMC will run:

- *In the first copy,* an input sequence such that 14 FIFO entries are filled. It will collect the RSRs after the input sequence has finished executing.
- *In the second copy* the same sequence followed by $I^{15}_{A_2}$, $I^1_{A_3}$ and $I^{16}_{A_2}$. Because of the bug, $I^{15}_{A_2}$ will be overwritten and output will be $F(I^1_{A_3}) \times I^{16}_{A_2}$.
- *In the third copy* an input sequence such that the RSR values after it finishes execution, the RSRs values match that of the first copy. Next, the BMC sends $I^{15}_{A_2}$, $I^1_{A_3}$ and the output will be $F(I^1_{A_3}) \times I^{15}_{A_2}$ not equal to the output in second copy.

The bug in FIFO 1 can be caught in a similar manner.

To ensure completeness, we run RB as well as SAC [6]. RB checks that the output for an input in a sequence is generated within some time bound. It works for interfering HAs. To check SAC for interfering actions, we need to consider the RSR values. To perform SAC on the action pair $\{A_1, A_2\}$, starting from reset state, we check that for all valid input values and associated RSRs the output produced by the HA is the same as expected by the A_1 and A_2 actions.

IV. Formal Models and Theoretical Results

In this section, we formalize G-QED. We adapt the HA formalism from [6] to model a general set of HAs. Such transition systems implicitly include a clock signal to synchronize transitions between system states [10]. Additionally, we formally define the notion of RSRs, the total correctness of the HA in the context of RSRs, and the FC property. We prove that FC is both sound and complete.

We first simplify the HA model in [6] by modeling all redundant inputs (either because the HA is not ready to read or the input is invalid) as an invalid action $a_\perp$ (the two cases of redundancy were handled separately in [6]). We similarly model redundant outputs (invalid or the host is not ready to read the output) as $o_\perp$. To model actions that only update RSRs and do not produce any output, we distinguish A_{no} from the full set of actions A. The actions that update the $Bypass$ and $Factor$ registers in Fig. 1 belong to this class of actions.

Definition 1 (Accelerator): An HA is a finite state transition system $Acc := (S, S_{init}, A, A_{no}, a_\perp, D, O, o_\perp, T, F)$, where:

- S is the finite set of states of the design. We assume that the system contains num independent state components so that $S = S_1 \times \cdots \times S_{num}$, where each S_i represents the set of values for the i^{th} component.
- $S_{init} \subseteq S$ is the set of initial states.
- A is a finite set of actions supported by the accelerator;
- $A_{no} \subseteq A$ is the subset of actions that do not generate a valid output but do update internal states of the accelerator;
- $a_\perp \in A$ 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;
- $o_\perp \in O$ is a distinguished element of O used to indicate that no output is being produced or that the output produced is not valid;
- $T : S \times A \times D \to S$ is the state transition function;
- $F : S \to O$ is the output function.

For the HA in Fig. 1, S_{init} contains just the reset state, A consists of the three actions that either update the $Bypass$ or $Factor$ registers or generate the output. An input has an action value of $a_\perp$ when *in_valid* or the *rdy_out* is low. An output takes the value $o_\perp$ when *rdy_in* or *out_valid* is low.

We assume that an accelerator Acc starts in some initial state $s_{init} \in S_{init}$. The execution of Acc is determined by the current input from the set $I = A \times D$. Each input $in \in I$ includes the input data and an action specifying the function to perform on the data. We write $a(in)$ and $d(in)$ for the first (action) and second (data) components of an input in, respectively. Given a state s and an input in, the next state is given by $s' = T(s, a(in), d(in))$, which we also write as $s' = T(s, in)$. At each state s, Acc produces an output $O = F(s)$. We use $\mathbf{v}$ to denote a sequence with elements denoted v_i and length $|\mathbf{v}|$, so $\mathbf{v} = \langle v_1, v_2, \ldots, v_{|\mathbf{v}|}\rangle$. We concatenate sequences (and with a slight abuse of notation, single elements with sequences) using '$\cdot$', e.g., $\mathbf{v} = v_1 \cdot \mathbf{v}'$, where $\mathbf{v}' = \langle v_2, \ldots, v_{|\mathbf{v}|}\rangle$.

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

We use $a_\perp$ to model inputs ignored by the HA and similarly $o_\perp$ to model outputs ignored by the host. We write $\mathbf{in}^k_\perp$ for a sequence of inputs $\mathbf{in}$ of length k such that $a(in_i) = a_\perp$ for $i \in [1, k]$ and similarly $\mathbf{o}^k_\perp$ for a sequence of k outputs, all of which are $o_\perp$. For an input sequence $\mathbf{in}$, the sequence of captured inputs, $C_{in}(\mathbf{in})$ is the subsequence of $\mathbf{in}$ obtained by keeping only elements in_i where $a(in_i) \neq a_\perp$ and $a(in_i) \notin A_{no}$. Similarly, if $\mathbf{s} = T(s_0, \mathbf{in})$, then the sequence of captured outputs, $C_{out}(s_0, \mathbf{in})$ is the subsequence of $F(s_0 \cdot \mathbf{s})$ obtained by keeping all elements $F(s_i)$ not equal to $o_\perp$.

To model interfering actions, we must allow the output produced by a specific input to depend on the current state as well as the input itself. To that end, we will introduce a notion of *relevant* state components. We first define projections of states. If $s \in S$ is a state, then we denote by s_i (with $i \in [1, num]$) the value of the i^{th} state component in the state s. We also call s_i the *projection of s with index i*. Let $p \subseteq [1, num]$ be a *projection set*. Then s_p is the sequence containing only the projections of s whose indices are in p ordered by index (i.e., $s_p = \langle s_{i_1}, \ldots, s_{i_{|p|}}\rangle$, where $p = \{i_1, \ldots, i_{|p|}\}$ and $i_j \leq i_{j+1}$ for $j \in [1, |p| - 1]$). We say that a state s is *stable* if for each $m > 0$, $s = T(s, \mathbf{in}^m_\perp)$ and $|C_{out}(s, \mathbf{in}^m_\perp)| = 0$, i.e., the state does not change and no outputs are produced when any sequence of invalid actions is applied.

Definition 2 (Relevant State Projection): Given an HA Acc and an ordered pair (a_i, a_j) of actions with $a_j \notin A_{no}$, the *relevant state projection* $rsp(a_i, a_j)$ is a minimal projection set p such that for all initial states s_{init}, input sequences $\mathbf{in_1}$, $\mathbf{in_2}$, integers k^1, l^1, k^2, l^2 and inputs in_1 and in_2 with $a(in_1) = a_i$ and $a(in_2) = a_j$, if

- $s^{1\_pre} = T(s_{init}, \mathbf{in_1} \cdot \mathbf{in}_{\perp}^{k^1})$, with s^{1_pre} stable,
- $s^{1\_post} = T(s^{1\_pre}, in_1 \cdot in_2 \cdot \mathbf{in}_{\perp}^{l^1})$, with s^{1_post} stable,
- $\mathbf{o^1} = C_{out}(s^{1\_pre}, in_1 \cdot in_2 \cdot \mathbf{in}_{\perp}^{l^1})$,
- $s^{2\_pre} = T(s_{init}, \mathbf{in_2} \cdot \mathbf{in}_{\perp}^{k^2})$, with s^{2_pre} stable,
- $s^{2\_post} = T(s^{2\_pre}, in_1 \cdot in_2 \cdot \mathbf{in}_{\perp}^{l^2})$, with s^{2_post} stable,
- $\mathbf{o^2} = C_{out}(s^{2\_pre}, in_1 \cdot in_2 \cdot \mathbf{in}_{\perp}^{l^2})$,

then, $s_p^{1\_pre} = s_p^{2\_pre} \rightarrow \mathbf{o^1} = \mathbf{o^2}$.

Note that p may not be unique—in that case, we assume that rsp deterministically picks a minimal projection set from among those satistfying the above definition. For the example in Fig. 1, $rsp(A_3, a_{\perp})$ is $p = \{i_1, i_2\}$ where i_1 and i_2 are the indices of the $Bypass$ and $Factor$ registers, respectively.

We can now define a notion of correctness.

Definition 3 (Total Correctness): An HA Acc is correct if for every ordered pair of actions (a_i, a_j) such that $a_i = a_{\perp}$ or $a_i \in A_{no}$, and $a_j \notin A_{no}$ and $a_j \neq a_{\perp}$, input sequences $\mathbf{in}$, $\mathbf{in}'$, inputs in_1 and in_2 with $a(in_1) = a_i$ and $a(in_2) = a_j$, and integers k and l, if

- $s^{pre} = T(s_{init}, \mathbf{in} \cdot \mathbf{in}_{\perp}^{k})$, with s^{pre} stable,
- $\mathbf{o} = C_{out}(s_{init}, \mathbf{in} \cdot in_1 \cdot in_2 \cdot \mathbf{in}')$ with $|\mathbf{o}| = |C_{in}(\mathbf{in} \cdot in_1 \cdot in_2)|$

then $o_{|o|} = Spec^{ij}(d(in_1), d(in_2), s^{pre}_{rsp(a_i,a_j)})$, where $Spec^{ij} : D \times D \times S_{i_1} \times \cdots \times S_{i_{|p|}} \rightarrow O$ (with $rsp(a_i, a_j) = p = \{i_1, \ldots, i_{|p|}\}$) defines the expected output for actions (a_i, a_j).

This definition of total correctness assumes an important design constraint—the output produced is independent of the presence of $a_{\perp}$ actions in the input sequences. This is the same constraint as Constraint in Sec. III.

Now, we will define functional consistency.

Definition 4 (Functional Consistency Property): Given an HA Acc and an ordered pair of actions (a_i, a_j) such that $a_i = a_{\perp}$ or $a_i \in A_{no}$, and $a_j \notin A_{no}$ and $a_j \neq a_{\perp}$, for any input sequences $\mathbf{in_1}$, $\mathbf{in_2}$, $\mathbf{in_1'}$, inputs in_1, in_2 with $a(in_1) = a_i$ and $a(in_2) = a_j$, and values k^1, k^2 and l, if

- $s^{1\_pre} = T(s_{init}, \mathbf{in_1} \cdot \mathbf{in}_{\perp}^{k^1})$, with s^{1_pre} stable,
- $\mathbf{o^1} = C_{out}(s_{init}, \mathbf{in_1} \cdot in_1 \cdot in_2 \cdot \mathbf{in_1'})$ with $|\mathbf{o^1}| = |C_{in}(\mathbf{in_1} \cdot in_1 \cdot in_2)|$,
- $s^{2\_pre} = T(s_{init}, \mathbf{in_2} \cdot \mathbf{in}_{\perp}^{k^2})$, with s^{2_pre} stable,
- $\mathbf{o^2} = C_{out}(s^{2\_pre}, in_1 \cdot in_2 \cdot \mathbf{in}_{\perp}^{l})$ with $|\mathbf{o^2}| = 1$,

then $s^{1\_pre}_{rsp(a_i,a_j)} = s^{2\_pre}_{rsp(a_i,a_j)} \rightarrow o^1_{|o^1|} = o^2_{|o^2|}$.

We will now show the soundness and completeness of FC.

Lemma 1 (Soundness of FC): If an HA Acc is totally correct, it is functionally consistent.

Proof 1: In def. 4, $o^1_{|o^1|} = Spec^{ij}(d(in_1), d(in_2), s^{1\_pre}_{rsp(a_i,a_j)})$ by definition 3. $o^2_{|o^2|} = Spec^{ij}(d(in_1), d(in_2), s^{2\_pre}_{rsp(a_i,a_j)})$ by definition 3 with $\mathbf{in}$ as $\mathbf{in_2} \cdot \mathbf{in}_{\perp}^{k^2}$ and $\mathbf{in}' = \mathbf{in}_{\perp}^{l}$ from definition 4. Then, $s^{1\_pre}_{rsp(a_i,a_j)} = s^{2\_pre}_{rsp(a_i,a_j)} \rightarrow Spec^{ij}(d(in_1), d(in_2), s^{1\_pre}_{rsp(a_i,a_j)}) = Spec^{ij}(d(in_1), d(in_2), s^{2\_pre}_{rsp(a_i,a_j)}) \rightarrow o^1_{|o^1|} = o^2_{|o^2|}$.

To prove completeness, we need Single Action Correctness.

Definition 5 (Single Action Correctness Property): Given an HA Acc and an ordered pair of actions (a_i, a_j) such that $a_i = a_{\perp}$ or $a_i \in A_{no}$, and $a_j \notin A_{no}$ and $a_j \neq a_{\perp}$, for any input sequence $\mathbf{in}$, inputs in_1 and in_2 with $a(in_1) = a_i$ and $a(in_2) = a_j$, and value k, if

- $s^{pre} = T(s_{init}, \mathbf{in} \cdot \mathbf{in}_{\perp}^{k})$, with s^{pre} stable,
- $\mathbf{o} = C_{out}(s^{pre}, in_1 \cdot in_2 \cdot \mathbf{in}_{\perp}^{l})$ with $|\mathbf{o}| = 1$,

then $o_1 = Spec^{ij}(d(in_1), d(in_2), s^{pre}_{rsp(a_i,a_j)})$.

Lemma 2 (Completeness of FC): If an HA Acc is single action correct but not totally correct, it fails FC.

Proof 2: Since Acc is not total correct, by definition 3, there exists $\mathit{in}_\mathbf{1}$ such that $o^1_{|o^1|} \neq Spec^{ij}(in_1, in_2, s^{1\_pre}_{rsp(a_i,a_j)})$ where $s^{pre} = T(s_{init}, \mathit{in}_\mathbf{1} \cdot in_{\perp}^{k})$. Now, we choose a sequence in_2 such that $s^{2\_pre}_{rsp(a_i,a_j)} = s^{1\_pre}_{rsp(a_i,a_j)}$. Since Acc is single action correct, in definition 4, $o^2_1 = Spec^{ij}(in_1, in_2, s^{2\_pre}_{rsp(a_i,a_j)}) = Spec^{ij}(in_1, in_2, s^{1\_pre}_{rsp(a_i,a_j)}) \rightarrow o^1_{|o^1|} \neq o^2_1$.

V. Results

A. G-QED for Various HAs and Processor Cores

We demonstrate the effectiveness and practicality of G-QED using 44 designs covering a wide variety of HAs and processor cores, including several industrial designs. While we focus on interfering HAs, we have also included non-interfering HAs to demonstrate the generality of G-QED. The HAs represent diverse application domains such as security, neural nets, and image processing. The processor cores belong to the RISC-V family [11] targeting embedded security and automotive domains. The design sizes are suitable for existing BMC tools.

Table I summarizes the results. All designs were previously verified using state-of-the-art simulation-based verification. The industrial designs were additionally verified using state-of-the-art formal verification. All designs were available in Verilog RTL. For G-QED runs on *Open Source Designs*, we used JasperGold® (V2016.09p002) on an Intel®Xeon® E5-2640 v3 with 128GB of DRAM. We used OneSpin 360 DV-Verify®on an Intel®Xeon®E5-2690 v3@2.6GHz with 50GB RAM to run G-QED on *Industrial Designs*.

Observation 1: **G-QED enables highly thorough verification of a wide variety of HAs and processor cores**. G-QED uniquely detected bugs that were missed by conventional verification flows (which includes both simulation and traditional formal verification), in addition to all bugs detected using those flows. The bugs uniquely detected by G-QED represent corner-case scenarios that are often highly difficult to detect. For example, for the AIE-A design, G-QED detected a difficult bug which got triggered by a complex sequence of 16 actions causing some inputs of another action to be dropped. G-QED could detect this bug within 5 hours with a 32-cycle counterexample. From our industrial experience, such bugs can be very tricky to detect using simulations.

Observation 2: **G-QED enables quick verification of HAs and processors**. G-QED detected bugs quickly with short counterexamples (in terms of clock cycles), which in turn enabled quick debug. Since G-QED uses BMC, it finds the shortest sequence to detect bugs. This is in sharp contrast to conventional flows where counterexample lengths and error traces are highly dependent on the precision of properties, assertions, and test cases. The effort required to set up G-QED is small resulting in large verification productivity benefits – an

TABLE I. G-QED results: HAs and Processors

Design			**Size**		**#Bugs**		**Runtime**	**Bug trace**
Name	**Type**	**#Versions**	**#FF**	**#Gates**	Bugs detected by G-QED and conventional flow	New bugs detected by G-QED only	**[max:avg:min]** minutes	**[max:avg:min]** clock cycles
Open Source Designs								
Stream cipher	Interfering HA	1	13k	86k	1	0	15:15:15	35:35:35
Deflate		1	10k	60k	Not available	1	5:5:5	11:11:11
MEM controller1		2	20k	0.1M	2	0	2:1:1	4:4:4
MEM controller2		13	20k	0.1M	12	1	5:2:1	9:7:4
FIFO	Non-interfering HA	5	20k	0.1M	5	3	2:1:1	7:5:4
NVDLA cbuf		1	0.1M	0.3M	1	0	21:21:21	25:25:25
NVDLA cacc		2	11k	56k	2	0	14:12:11	108:106:105
AES		4	2k	31k	4	0	4:1:<1	290:163:94
Industrial Designs								
AIE-A	Interfering HA	2	4k	71k	G-QED detected all bugs detected by industrial flow. Bug counts are not reported for reasons of confidentiality	4	240:180:60	32:26:20
AIE-D		2	4k	71k		5	300:150:90	32:30:20
Core1	Processor core	4	3.8k	54k		0	1:1:<1	7:7:6
Core2		2	2.5k	45k		0	1:1:1	5:5:5
Core3		1	1.8k	18k		0	<1:<1:<1	4:4:4
Core4		4	5k	65k		0	2:1:1	6:6:6
A total of 44 designs are used for G-QED results								

apples-to-apples comparison with industrial verification flows is presented in Sec. V-B.

Observation 3: **G-QED provides a unified verification solution for interfering HAs, non-interfering HAs, and processor cores**. G-QED thus advances state-of-the-art compared to other QED pre-silicon verification techniques such as A-QED [6] (for non-interfering HAs), and Symbolic QED [12], S^{2}QED [13], and C-S^{2}QED [14] (for processor cores).

Observation 4: **G-QED integrates seamlessly with various BMC engines.** We used G-QED together with JasperGold® (V2016.09p002) from Cadence and OneSpin 360 DV-Verify® from Siemens EDA to generate Table I.

B. Industrial Case Study on Live Designs

For an apples-to-apples comparison of G-QED versus industrial verification flows, we conducted a case study using Artificial Intelligence Engine (*AIE*) HAs. We had access to several (buggy) versions of two Verilog designs, AIE-A and AIE-D (Table I), where AIE-D is a derivative of AIE-A with additional features. Both designs represent interfering HAs that are also LCAs. At the start of our G-QED study, both HAs already passed pre-silicon verification sign-off and were in the final stages of design – hence, we refer to them as *live* designs.

1) Industrial Verification Flow: Our industrial verification flow included both Constrained Random Simulation (*CRS*) with UVM-based [15] test bench and Formal Verification (*FV*) with design-specific properties. CRS was carried out at the top level, targeting all design functionality. For design-specific property creation, a framework similar to [16]. was used to reduce manual effort. A standard industrial verification process similar to [17] was employed. The sign-off criteria for verification was 100% structural code coverage and 100% functional coverage from both CRS and FV combined.

2) G-QED Flow: For each AIE HA, we defined the set of actions and extracted the associated RSR registers and input-output interface protocol details with help from design engineers. We then used the methodology explained in Sec. III. No knowledge of the internal architecture (performance optimizations, parallelism, internal pipelining) or existing industrial verification flow was required. With the G-QED FC (Sec. III), we were able to detect and root-cause 9 new bugs (missed by the industrial flow) within 3 person weeks, in addition to detecting bugs detected by the industrial flow (Table II).

TABLE II. Summary of industrial case study

Design	**Industrial flow effort** (person days)	**G-QED effort** (person days)	**Bugs detected by G-QED only**
AIE-A	140 (CRS: 110, FV: 30)	13	4
AIE-D	230 (CRS: 160, FV: 70)	8	5

Observation 5: **G-QED significantly improves bug coverage versus industrial verification flows.** G-QED uniquely detected 9 corner-case and difficult-to-activate bugs in well-verified industrial designs. Of these 9 bugs, 4 were due to the specification inaccuracies and 5 were in the RTL implementations that were only triggered by a complex sequence of actions. All these bugs were detected in less than 5 hours.

Observation 6: **G-QED dramatically improves verification productivity.** The industrial verification flow required 370 person days for the two HAs. G-QED required only 21 person days (including setup, running the BMC tool, and root-cause analysis) – an 18-fold verification productivity boost.

Observation 7: **G-QED can be set up very quickly.** It took only 7 and 2 person days for AIE-A and AIE-D to set up G-QED, respectively. In contrast, industrial verification flows often require detailed information about a design and its implementation. For the AIE HAs, it took 250 person days to set up CRS and FV (test bench and property development).

Observation 8: **G-QED enables quick debug.** G-QED produced short counter-examples (32 cycles or fewer) for the AIE HAs (Table I) resulting in quick debug (less than a day). cause analysis. G-QED effort in Table II includes this debug effort. In contrast, it took several days for root cause analysis of some of the bugs detected by CRS.

Observation 9: **G-QED does not require low-level design details.** G-QED largely relies on self-consistency checks, minimizing the need to understand deep design details. This saves verification effort significantly.

Observation 10: **G-QED enables short design-design verification loops with quick turnaround for rapidly evolving designs.** Once the bugs detected by G-QED were fixed, we could reuse G-QED immediately to verify the fixed designs with little or no additional effort. In contrast, it required up to 120 person days for CRS and FV to debug, fix, update test cases and properties, and rerun verification.

VI. Related Work

Simulation-based methodologies dominate the pre-silicon verification practice in industry [15], [17], [18]. However, as demonstrated in Sec. V-B, simulation-based methods are inadequate for ensuring thorough and quick verification of HAs. While simulation scales for large designs, there can be major scaleup opportunities for G-QED using techniques such as [7] – a topic of future work.

While G-QED employs BMC for self-consistency checking, it differs from traditional formal verification (Sec. V-B). G-QED can leverage recent formal verification advances to further reduce associated manual efforts (e.g., automatic RSR extraction [19], ILA [5]).

Self-consistency-based checking has its roots in fault-tolerant computing, including the use of design diversity. For pre-silicon verification, publications such as [20], [21] are relevant. Completeness guarantees may be affected by these techniques (e.g., for a bug always triggered by a particular sequence of instructions, irrespective of the number of NOPs and stalls inserted in the sequence). Another closely approach [22] requires specific clock cycles when RSRs are read and written, which can be very challenging for HAs (Sec. II). In contrast, we proved G-QED to be sound and complete.

TABLE III. G-QED vs. Other pre-silicon QED techniques

Attribute	**G-QED** (This paper)	**A-QED** [6]	**A-QED^2** [7]	**Symbolic QED** [12], [23]	**S^{2}QED** [13] **C-S^{2}QED** [14]
Standalone HAs	Yes	Non-interfering HAs only		No	No
Processor cores	Yes	No	No	Yes	Yes
Designer input for RSRs	Yes	NA	NA	Yes	Yes
Clock cycle details for RSR updates	No	NA	NA	Yes	Yes
Theoretical guarantees	Yes	Yes	Yes	Yes ([12] implementation might miss some bugs)	Yes
Design size	Moderate	Moderate	Large	Moderate/ Large	Moderate
Symbolic starting states	No	No	No	No	Yes

Recently, there have been several publications on various flavors of QED pre-silicon verification techniques, namely, Symbolic QED [12], S^{2}QED [13], C-S^{2}QED [14], A-QED [6], and A-QED^2 [7]. Table III contrasts these techniques versus G-QED with respect to various attributes.

VII. Conclusion

G-QED presents a unified approach to highly thorough pre-silicon verification of interfering HAs, non-interfering HAs, and processor cores through self-consistency. With modest design assumptions, G-QED has been proven to be sound and complete. Results from a wide variety of designs, including an industrial case study, demonstrate its effectiveness and practicality. G-QED creates several promising research directions: (1) scale-up to very large designs through new decomposition and abstraction techniques (e.g., [7]). (2) G-QED with symbolic starting states to overcome BMC bounds; (3) G-QED for verifying third-party IP blocks with little/obfuscated internal design details; (4) functional safety verification through a combination of G-QED and formal fault injection techniques; and, (5) G-QED for deriving side-channel attacks in HAs (similar to timing side channels for processors [24]).

Acknowledgement

This work was supported in part by ACE (one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA), Stanford SystemX Alliance, and SRC-sponsored Global Research Collaboration.

References

[1] "A14 Bionic - Apple - WikiChip." [Online]. Available: https://en.wikichip.org/wiki/apple/ax/a14
[2] J. Cross, "A14 bionic FAQ: What you need to know about Apple's 5nm processor," 2020, https://www.macworld.com/article/234595/a14-bionic-faq-performancefeatures-cpu-gpu-neural-engine.html.
[3] Norrie et al., "The Design Process for Google's Training Chips: TPUv2 and TPUv3," *IEEE Micro*, 2021.
[4] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.
[5] Huang et al., "Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification," *ACM Trans. Des. Autom. Electron. Syst.*, 2019.
[6] Singh et al., "A-QED Verification of Hardware Accelerators," in *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020.* IEEE, 2020, pp. 1–6. [Online]. Available: https://doi.org/10.1109/DAC18072.2020.9218715
[7] Chattopadhyay et al., "Scaling up hardware accelerator verification using a-qed with functional decomposition," in *2021 Formal Methods in Computer Aided Design (FMCAD).* IEEE, 2021, pp. 42–52.
[8] Clarke et al., "Bounded Model Checking using Satisfiability Solving," *Formal methods in system design*, vol. 19, no. 1, pp. 7–34, 2001.
[9] Peccerillo et al., "A Survey on Hardware Accelerators: Taxonomy, Trends, Challenges, and Perspectives," *Journal of Systems Architecture*, vol. 129, p. 102561, 2022.
[10] R. M. Keller, "Formal Verification of Parallel Programs," *Communications of the ACM*, vol. 19, no. 7, pp. 371–384, 1976.
[11] A. Waterman and K. Asanovic. (2017, May) The RISC-V Instruction Set Manual, Volume I: User-Level ISA. https://riscv.org/specifications.
[12] Singh et al., "Logic Bug Detection and Localization Using Symbolic Quick Error Detection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.
[13] Fadiheh et al., "Symbolic Quick Error Detection using Symbolic Initial State for Pre-silicon Verification," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE, 2018, pp. 55–60.
[14] Devarajegowda et al., "Gap-free Processor Verification by S2QED and Property Generation," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020, pp. 526–531.
[15] "IEEE Standard for Universal Verification Methodology Language Reference Manual," *IEEE Std 1800.2-2017*, pp. 1–472, 2017.
[16] Devarajegowda et al., "Python Based Framework for HDSLs with an Underlying Formal Semantics," in *2017 IEEE/ACM ICCAD*, Nov 2017.
[17] Singh et al., "Symbolic QED Pre-silicon Verification for Automotive Microcontroller Cores: Industrial Case Study," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019.
[18] Chen et al., "Challenges and Trends in Modern SoC Design Verification," *IEEE Design & Test*, vol. 34, no. 5, pp. 7–22, 2017.
[19] Zeng et al., "Generating Architecture-level Abstractions from RTL Designs for Processors and Accelerators Part I: Determining Architectural State Variables," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD).* IEEE, 2021, pp. 1–9.
[20] R. B. Jones, C.-J. H. Seger, and D. L. Dill, "Self-consistency Checking," in *International Conference on Formal Methods in Computer-Aided Design.* Springer, 1996, pp. 159–171.
[21] J. R. Burch and D. L. Dill, "Automatic Verification of Pipelined Microprocessor Control," in *International Conference on Computer Aided Verification.* Springer, 1994, pp. 68–80.
[22] Reid et al., "End-to-end Verification of Processors with ISA-Formal," in *International Conference on Computer Aided Verification.* Springer, 2016, pp. 42–58.
[23] F. Lonsing, S. Mitra, and C. W. Barrett, "A Theoretical Framework for Symbolic Quick Error Detection," *2020 Formal Methods in Computer Aided Design (FMCAD)*, pp. 1–10, 2020.
[24] Fadiheh et al., "Processor Hardware Security Vulnerabilities and Their Detection by Unique Program Execution Checking," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE.