SQED and Symbolic Starting States

Author A, Author B, ...
Computer Science Department, Stanford University, Stanford, CA 94305, USA
E-mail: {authoremail}@stanford.edu

***Abstract*—TBA**

I. Notes

- 8 February 2021: SQED formal model and theoretical results (except extensions of QED tests with reset instructions) copied from FMCAD'20 paper.
- All proofs are included.
- The overview in Section III is the same as in the FMCAD paper. Maybe we can present the SSS approach in a similar and informal way first.

II. Introduction

III. Overview of SQED

We first informally introduce the basic concepts and terminology related to SQED. Fig. 1a shows an overview of the high-level workflow. Given a processor design $\mathcal{P}$, i.e., the DUV, SQED is based on symbolic execution of instruction sequences using BMC. We assume that an *instruction* $i = (op, l, (l', l''))$ consists of an opcode op, an output location l, and a pair (l', l'') of input locations.[1] *Locations* are an abstraction used to represent registers and memory locations.

The self-consistency check is based on executing two instructions that should always produce the same result. The two instructions are called an *original* and a *duplicate instruction*, respectively. The duplicate instruction has the *same opcode* as the original one, i.e., it implements the same functionality, but it operates on different input and output locations. The locations on which the duplicate instruction operates are determined by an *arbitrary but fixed bijective function* $L_D : \mathcal{L}_O \to \mathcal{L}_D$ between two subsets $\mathcal{L}_O$, the *original locations*, and $\mathcal{L}_D$, the *duplicate locations*, that form a partition of the set $\mathcal{L}$ of all locations in $\mathcal{P}$. An original instruction can only use locations in $\mathcal{L}_O$. An *instruction duplication function* Dup then maps any original instruction i_O to its duplicate i_D by copying the opcode and then applying L_D to its locations.

Example 1. *Let* $\mathcal{L} = \{0, \ldots, 31\}$ *be the identifiers of 32 registers of a processor* $\mathcal{P}$*, and consider the partition* $\mathcal{L}_O = \{0, 1, \ldots, 15\}$ *and* $\mathcal{L}_D = \{16, 17, \ldots, 31\}$*. Let* $i_O = (\mathsf{ADD}, l_{12}, (l_4, l_8))$ *be an original register-type ADD instruction operating on registers* $4, 8$*, and* 12*. Using* $L_D(k) = k + 16$*, we obtain* $Dup(i_O) = i_D = (\mathsf{ADD}, l_{28}, (l_{20}, l_{24}))$*.*

Consider a different partition $\mathcal{L}'_O = \{0, 2, 4, \ldots, 30\}$ *and* $\mathcal{L}'_D = \{1, 3, 5, \ldots, 31\}$ *and function* $L'_D(k) = k + 1$*. For this function,* $Dup(i_O) = (\mathsf{ADD}, l_{13}, (l_5, l_9))$*.*

Self-consistency checking is implemented using *QED tests*. A QED test is an instruction sequence $\boldsymbol{i} = \boldsymbol{i_O} :: \boldsymbol{i_D}$ consisting of a sequence $\boldsymbol{i_O}$ of n original instructions followed by a corresponding sequence $\boldsymbol{i_D} = Dup(\boldsymbol{i_O})$ of n duplicate instructions (where operator "::" denotes concatenation). A QED test $\boldsymbol{i}$ is symbolically executed from a *QED-consistent state*, that is, a state where the value stored in each original location l is the same as the value stored in its corresponding duplicate location $\mathcal{L}_D(l)$. The resulting final state after executing $\boldsymbol{i}$ should then also be QED-consistent. Fig. 1a illustrates the workflow. A QED test $\boldsymbol{i}$ *succeeds* if the final state that results from executing $\boldsymbol{i}$ is QED-consistent; otherwise it *fails*. Starting the execution in a QED-consistent state guarantees that original and duplicate instructions receive the same input values. Thus, if the final state is not QED-consistent, then this indicates that some pair of original and duplicate instructions behaved differently.

Example 2. *Consider Fig. 1b and the QED test* $\boldsymbol{i} = i_O :: i_D$ *consisting of one original instruction* i_O *and its duplicate* $Dup(i_O) = i_D$ *for some function* L_D*. Suppose that* $\boldsymbol{i}$ *is executed in a QED-consistent state* s_0 *(denoted by* $QEDcons(s_0)$ *and* $s_0(\mathcal{L}_O) = s_0(\mathcal{L}_D)$*) and both* i_O *and* i_D *execute correctly. Instruction* i_O *produces state* s_1*, where the values at duplicate locations remain unchanged, i.e.,* $s_0(\mathcal{L}_D) = s_1(\mathcal{L}_D)$*, because* i_O *operates on original locations only. When instruction* i_D *is executed in state* s_1*, it modifies only duplicate locations. The final state* s_2 *is QED-consistent (denoted by* $QEDcons(s_2)$ *and* $s_2(\mathcal{L}_O) = s_2(\mathcal{L}_D)$*), and thus QED test* $\boldsymbol{i}$ *succeeds.*

Example 3 (Bug Detection)**.** *Consider processor* $\mathcal{P}$ *and* $\mathcal{L}_O$ *and* $\mathcal{L}_D$ *from Example 1. Let* $i_{O,1} = (\mathsf{ADD}, l_{12}, (l_4, l_{15}))$ *and* $i_{O,2} = (\mathsf{MUL}, l_{15}, (l_{12}, l_{12}))$ *be original register-type addition and multiplication instructions. Using* $L_D(k) = k+16$*, we obtain* $Dup(i_{O,1}) = i_{D,1} = (\mathsf{ADD}, l_{28}, (l_{20}, l_{31}))$ *and* $Dup(i_{O,2}) = i_{D,2} = (\mathsf{MUL}, l_{31}, (l_{28}, l_{28}))$*. Assume that* $\mathcal{P}$ *has a bug that is triggered when two MUL instructions are executed in subsequent clock cycles, resulting in the corruption of the output location of the second MUL instruction.*[2] *Note that executing the QED test* $\boldsymbol{i} = i_{O,1}, i_{O,2} :: i_{D,1}, i_{D,2}$ *in a QED-consistent initial state produces a QED-consistent final state: the bug is not triggered by* $\boldsymbol{i}$ *because* $i_{D,1}$ *is executed between* $i_{O,2}$ *and* $i_{D,2}$*. A slightly longer test* $\boldsymbol{i} = i_{O,2}, i_{O,1}, i_{O,2} :: i_{D,2}, i_{D,1}, i_{D,2}$ *does trigger the bug, however, because the subsequence* $i_{O,2}, i_{D,2}$ *of two back-to-back MULs causes the first duplicate instruction* $i_{D,2}$ *in* $\boldsymbol{i}$ *to*

Acknowledgments: TBA

[1] This model is used for simplicity, but it could easily be extended to allow instructions with additional inputs or outputs.

[2] This scenario corresponds to a real bug in an out-of-order RISC-V design detected by SQED: https://github.com/ridecore/ridecore/issues/4.

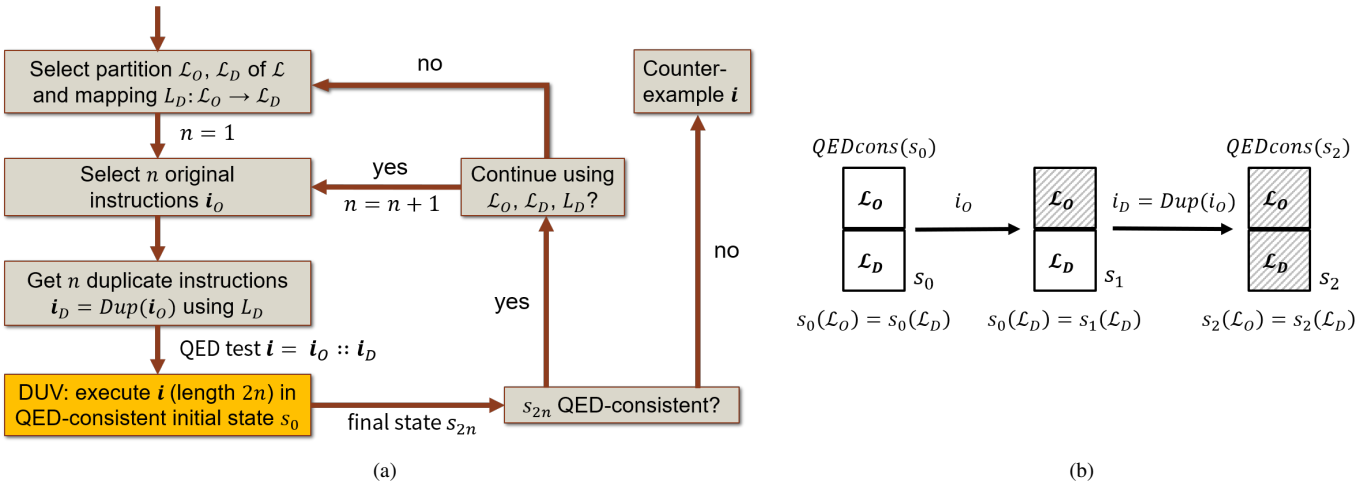

Fig. 1. SQED workflow from a theoretical perspective (a) and illustration of executing the QED test $\boldsymbol{i} = i_O :: i_D$ in Example 2 (b).

produce an incorrect result at l_{31}. This incorrect result then propagates through the next two instructions, resulting in a QED-inconsistent final state since the values at l_{15} and l_{31}, i.e., the output locations of $i_{O,2}$ and $i_{D,2}$, differ.

QED-consistency is the universal, implementation-independent property that is checked in SQED. In practice, the property must refer to some basic information about the design such as, e.g., symbolic register names, but this can be generated automatically from a high-level ISA description [1]. BMC is used to symbolically and exhaustively generate all possible QED tests up to a certain length $2n$ (the BMC bound). BMC ensures that SQED will find the shortest possible failing QED test first. The high-level workflow shown in Fig. 1a allows for flexibility in choosing the partition and mapping between original and duplicate locations. We rely on this flexibility for the results in this paper (Theorem 1). Current SQED implementations use a predefined partition and mapping, based on which BMC enumerates all possible QED tests. Extending implementations to have the BMC tool also choose a partition and mapping could be explored in future work.

We refer to related work [2]–[4] for case studies that demonstrate the effectiveness of BMC-based SQED on a variety of processor designs. The scalability of SQED in practice is determined by the scalability of the BMC tool being used. Thus, approaches for improving scalability of BMC can also be applied to SQED, e.g. abstraction, decomposition, and partial instantiation techniques [2].

IV. Instruction and Processor Model

We model a processor as a transition system containing an abstract set of locations. The set of locations includes registers and memory locations. A state of a processor consists of an *architectural* and a *non-architectural* part. In a state transition that results from executing an instruction, the architectural part of a state is modified explicitly by updating the value at the output location of the executed instruction. The architectural part of a state is also called the *software-visible* state of the processor. It comprises those parts of the state that can be updated by executing instructions of the user-level ISA of the processor, such as memory locations and general-purpose registers. The non-architectural part of a state comprises the remaining parts that are updated only implicitly by executing an instruction, such as pipeline or status registers.

Instructions are functions that take inputs from locations and write an output to a location. We assume that every instruction produces its result in one transition. In our model, we abstract away implementation details of complex processor designs (e.g., pipelined, out-of-order, multi-processor systems). This is for ease of presentation and reasoning. However, many of these complexities can be viewed as refinements of our abstraction, meaning that our formal results still hold on complex models (i.e., our results can be lowered to more detailed models such as those described in [2], [5]). Working out the details of such refinements is one important avenue for future work.

Definition 1 (Transition System)**.** *A processor is a* transition system *[6], [7] $\mathcal{P} = (\mathcal{V}, \mathcal{L}, S_{\overline{a}}, s_{\overline{a},I}, Op, I, T)$, where*

- *$\mathcal{V}$ is a set of* abstract data values*,*
- *$\mathcal{L}$ is a set of* memory locations *(from which we define the set S_a of* architectural states *as the set of total functions from locations to values, i.e. $S_a = \{s_a \mid s_a : \mathcal{L} \to \mathcal{V}\}$),*
- *$S_{\overline{a}}$ is a set of* non-architectural states *(from which we further define the set of all* states *as $S = S_a \times S_{\overline{a}}$),*
- *$s_{\overline{a},I} \in S_{\overline{a}}$ is a unique* initial non-architectural state *(from which we define the set of* initial states *as $S_I = S_a \times \{s_{\overline{a},I}\}$,*
- *Op is a set of* operation codes (opcodes)*,*
- *$I = Op \times \mathcal{L} \times \mathcal{L}^2$ is the set of* instructions*, and*
- *$T : S \times I \to S$ is the* transition function*, which is total.*

A state $s \in S$ with $s = (s_a, s_{\overline{a}})$ consists of an architectural part $s_a \in S_a$ and a non-architectural part $s_{\overline{a}} \in S_{\overline{a}}$. In the architectural part $s_a : \mathcal{L} \to \mathcal{V}$, $\mathcal{L}$ represents all possible registers and memory locations, i.e., in practical terms, $\mathcal{L}$ is the address space of $\mathcal{P}$. An initial state $s_I \in S_I$ with $s_I = (s_a, s_{\overline{a},I})$ is defined by a unique non-architectural part

$s_{\overline{a},I} \in S_{\overline{a}}$ and an arbitrary architectural part $s_a \in S_a$. We assume that $s_{\overline{a},I} \in S_{\overline{a}}$ is unique to make the exposition simpler. Our model could easily be extended to a set of initial non-architectural states. The number $|\mathcal{L}|$ of memory locations is arbitrary but fixed. We write $v = s(l)$ to denote the value $v = s_a(l)$ at location $l \in \mathcal{L}$ in state $s = (s_a, s_{\overline{a}})$. We also write $(v, v') = s(l, l')$ as shorthand for $v = s(l)$ and $v' = s(l')$.

To formally define instruction duplication, we need to reason about *original* and *duplicate* memory locations. To this end, we partition the set $\mathcal{L}$ of memory locations into two sets of equal size, the *original* and *duplicate locations* $\mathcal{L}_O$ and $\mathcal{L}_D$, respectively, i.e., $\mathcal{L}_O \cap \mathcal{L}_D = \emptyset$, $\mathcal{L}_O \cup \mathcal{L}_D = \mathcal{L}$, and $|\mathcal{L}_O| = |\mathcal{L}_D|$. Given $\mathcal{L}_O$ and $\mathcal{L}_D$, we define an **arbitrary but fixed** *bijective function* $L_D : \mathcal{L}_O \to \mathcal{L}_D$ that maps an original location $l_O \in \mathcal{L}_O$ to its corresponding duplicate location $l_D = L_D(l_O)$. The inverse of L_D is denoted by ${L_D}^{-1}$ and is uniquely defined. We write $(l_D, l'_D) = L_D(l_O, l'_O)$ as shorthand for $l_D = L_D(l_O)$ and $l'_D = L_D(l'_O)$. Function L_D implements a correspondence between original and duplicate locations, which we need to define QED-consistency (Definition 11 below).

An instruction $i \in I$ with $i = (op, l, (l', l''))$ is defined by an opcode $op \in Op$, an output location $l \in \mathcal{L}$, and a pair of input locations $(l', l'') \in \mathcal{L}^2$. Function $op : I \to Op$ maps an instruction to its opcode $op(i)$. Functions $L_{out} : I \to \mathcal{L}$ and $L_{in} : I \to \mathcal{L}^2$ map an instruction i to its output and input locations $L_{out}(i) = l$ and $L_{in}(i) = (l', l'')$, respectively. Given a state $s = (s_a, s_{\overline{a}})$, instruction i reads values in s from its input locations $L_{in}(i)$ and writes a value to its output location $L_{out}(i)$, resulting in a transition to a new state $s' = (s'_a, s'_{\overline{a}})$, written as $s' = T(s, i)$. The transition function T is total, i.e., for every instruction i and state s, there exists a successor state $s' = T(s, i)$. As mentioned above, we have kept the model simple in order to make the presentation more accessible, but our results can be lifted to many extensions, including, e.g., more complicated kinds of instructions or instructions with enabledness conditions cf. [8].

We write $\boldsymbol{i} \in I^n$ and $\boldsymbol{s} \in S^n$ to denote sequences $\boldsymbol{i} = \langle i_1, \ldots, i_n \rangle$ and $\boldsymbol{s} = \langle s_1, \ldots, s_n \rangle$ of n instructions and n states, respectively. We will use :: for sequence concatenation and extend the transition function T to sequences as follows.

Definition 2 (Path)**.** *Given sequences $\boldsymbol{i} = \langle i_1, \ldots, i_n \rangle$ and $\boldsymbol{s} = \langle s_1, \ldots, s_n \rangle$ of n instructions and states, $\boldsymbol{s}$ is a* path *from state $s_0 \in S$ to s_n via $\boldsymbol{i}$, written $\boldsymbol{s} = T(s_0, \boldsymbol{i})$, iff $\bigwedge_{k=0}^{n-1} s_{k+1} = T(s_k, i_{k+1})$.*

If $\boldsymbol{s} = T(s_0, \boldsymbol{i})$, then for convenience we also write $s_n = T(s_0, \boldsymbol{i})$ to denote the final state s_n.

Definition 3 (Reachable State)**.** *A state s is* reachable*, written $reach(s)$, iff $s = T(s_0, \boldsymbol{i})$ for some $s_0 \in S_I$ and instruction sequence $\boldsymbol{i}$.*

The set I of instructions contains as proper subsets the sets of *original* and *duplicate instructions*, I_O and I_D, respectively. Original (duplicate) instructions operate only on original (duplicate) locations, i.e., $\forall i_O \in I_O.\ L_{in}(i_O) \in \mathcal{L}_O^2 \wedge L_{out}(i_O) \in \mathcal{L}_O$ and $\forall i_D \in I_D.\ L_{in}(i_D) \in \mathcal{L}_D^2 \wedge L_{out}(i_D) \in \mathcal{L}_D$. Given these definitions, we formalize instruction duplication as follows.

Definition 4 (Instruction Duplication)**.** *Let $Dup : I_O \to I_D$ be an* instruction duplication function *that maps an original instruction $i_O = (op, l_O, (l'_O, l''_O))$ to a duplicate instruction $i_D = Dup(i_O) = (op, L_D(l_O), L_D(l'_O, l''_O))$ with respect to the bijective function L_D.*

An original instruction and its duplicate have the same opcode. We write $\boldsymbol{i_O} \in I_O^n$ and $\boldsymbol{i_D} \in I_D^n$ to denote sequences $\boldsymbol{i_O} = \langle i_{O,1}, \ldots, i_{O,n} \rangle$ and $\boldsymbol{i_D} = \langle i_{D,1}, \ldots, i_{D,n} \rangle$ of n original and duplicate instructions, respectively. We lift Dup in the natural way also to sequences of instructions as follows.

Definition 5 (Instruction Sequence Duplication)**.** *Let $\boldsymbol{i_O} = \langle i_{O,1}, \ldots, i_{O,n} \rangle$ be a sequence of original instructions. Then $Dup(\boldsymbol{i_O}) = \langle Dup(i_{O,1}), \ldots, Dup(i_{O,n}) \rangle$.*

V. Formalizing Correctness

Definition 6 (Abstract Specification)**.** $\forall\, s, s' \in S,\ i \in I.$

$$\begin{aligned} Spec(s, i, s') \leftrightarrow \forall l \in \mathcal{L}.\ & \\ & (l \neq L_{out}(i) \to s(l) = s'(l)) \wedge \qquad (1) \\ & (l = L_{out}(i) \to s'(l) = Spec_{op(i)}(s(L_{in}(i)))) \end{aligned}$$

As special cases of (1), original and duplicate instructions have the following properties:

$$\begin{aligned} \forall s, s' \in S,\ i_O \in I_O, l_O \in \mathcal{L}_O, i_D \in I_D, l_D \in \mathcal{L}_D.\ & \\ & (Spec(s, i_O, s') \to s(l_D) = s'(l_D)) \wedge \qquad (2) \\ & (Spec(s, i_D, s') \to s(l_O) = s'(l_O)) \qquad (3) \end{aligned}$$

The following *functional congruence* property of instructions also follows from (1):

$$\begin{aligned} \forall\, s_0, s_1, s', s'' \in S, i, i' \in I.\ & \\ \big[op(i) = op(i') \wedge Spec(s_0, i, s') \wedge Spec(s_1, i', s'') \wedge \qquad (4) \\ s_0(L_{in}(i)) = s_1(L_{in}(i'))\big] \to s'(L_{out}(i)) = s''(L_{out}(i')) \end{aligned}$$

Definition 7 (Correctness)**.** *A processor $\mathcal{P}$ is* correct *with respect to specification Spec iff $\forall\, i \in I, s \in S.\ reach(s) \to Spec(s, i, T(s, i))$.*

Definition 8 (Bug)**.** *A* bug with respect to specification *Spec in a processor $\mathcal{P}$ is defined by a pair $\mathcal{B} = \langle i_b, S_b \rangle$ consisting of an instruction $i_b \in I$ and a non-empty set $S_b \subseteq S$ of states such that $S_b = \{s \in S \mid reach(s) \wedge \neg Spec(s, i_b, T(s, i_b))\}$.*

Proposition 1. *A processor $\mathcal{P}$ has a bug with respect to specification Spec iff it is not correct with respect to Spec.*

Definition 9 (Single-Instruction Correctness)**.** *Processor $\mathcal{P}$ is* single-instruction correct *iff:*

$$\forall\, i \in I, s_0 \in S_I.\ Spec(s_0, i, T(s_0, i)).$$

Definition 10 (Single-Instruction Bug)**.** *Processor $\mathcal{P}$ has a* single-instruction bug *with respect to specification Spec iff $\exists\, i \in I, s_0 \in S_I.\ \neg Spec(s_0, i, T(s_0, i))$.*

VI. Self-Consistency as QED-Consistency

Definition 11 (QED-Consistency)**.** *A state s is* QED-consistent*, written $QEDcons(s)$, iff $\forall l_O \in \mathcal{L}_O.\ s(l_O) = s(L_D(l_O))$.*

Definition 12 (QED test)**.** *An instruction sequence $\boldsymbol{i}$ is a* QED test *if $\boldsymbol{i} = \boldsymbol{i_O} :: Dup(\boldsymbol{i_O})$ for some sequence $\boldsymbol{i_O}$ of original instructions.*

Corollary 1 (Functional Congruence: Duplicate Instructions)**.** *Given $i_O \in I_O$ and $i_D \in I_D$ with $i_D = Dup(i_O)$, the following holds for all states s_0, s_1, s', and s'':*

$$\big[Spec(s_0, i_O, s') \wedge Spec(s_1, i_D, s'') \wedge s_0(L_{in}(i_O)) = s_1(L_D(L_{in}(i_O)))\big] \rightarrow s'(L_{out}(i_O)) = s''(L_D(L_{out}(i_O)))$$

Lemma 1 (cf. Corollary 1)**.** *Given $i_O \in I_O$ and $i_D \in I_D$ with $i_D = Dup(i_O)$, the following holds for all states s_0, s_1, s', and s'':*

$$\big[Spec(s_0, i_O, s') \wedge Spec(s_1, i_D, s'') \wedge \forall l_O \in \mathcal{L}_O.\ s_0(l_O) = s_1(L_D(l_O))\big] \rightarrow \forall l_O \in \mathcal{L}_O.\ s'(l_O) = s''(L_D(l_O))$$

Proof. Assume that the antecedent of the implication holds, and let $l_O \in \mathcal{L}_O$ be an arbitrary original memory location. If $l_O = L_{out}(i_O)$, then $s'(L_{out}(i_O)) = s''(L_D(L_{out}(i_O)))$ by Corollary 1.

Suppose, on the other hand, that $l_O \neq L_{out}(i_O)$. Let $l_D = L_D(l_O)$ be the corresponding duplicate location. By the injectivity of L_D, we have $l_D \neq L_D(L_{out}(i_O))$, and thus $l_D \neq L_{out}(i_D)$. We can thus conclude from (1) that $s_0(l_O) = s'(l_O)$ and $s_1(l_D) = s''(l_D)$.

Finally, since $s_0(l_O) = s_1(l_D)$ by assumption, we derive $s'(l_O) = s''(l_D)$, that is, $s'(l_O) = s''(L_D(l_O))$. □

Lemma 2 (QED-Consistency and QED tests)**.** *Let $\boldsymbol{i} = \langle i_1, \ldots, i_{2n}\rangle$ be a QED test, let $\langle s_0, \ldots, s_{2n}\rangle$ be a sequence of $2n+1$ states, and let Spec be some abstract specification relation. Then,*

$$QEDcons(s_0) \wedge \Big(\bigwedge_{j:=0}^{2n-1} Spec(s_j, i_{j+1}, s_{j+1})\Big) \rightarrow QEDcons(s_{2n})$$

Proof. Assuming the antecedent, let $l_O \in \mathcal{L}_O$ be arbitrary but fixed with $l_D = L_D(l_O)$. By repeated application of (2), we derive $s_0(l_D) = s_1(l_D) = \ldots = s_n(l_D)$, and hence:

$$s_0(l_D) = s_n(l_D) \tag{5}$$

by transitivity. By repeated application of (3), we derive:

$$s_n(l_O) = s_{2n}(l_O) \tag{6}$$

Now, $QEDcons(s_0)$ implies $s_0(l_O) = s_0(L_D(l_O))$, from which it follows by (5) that $s_0(l_O) = s_n(L_D(l_O))$. By repeated application of Lemma 1, we can next derive $s_j(l_O) = s_{n+j}(L_D(l_O))$ for $1 \leq j \leq n$, and in particular, $s_n(l_O) = s_{2n}(L_D(l_O))$. Finally, by applying (6), we get $s_{2n}(l_O) = s_{2n}(L_D(l_O))$. Since l_O was chosen arbitrarily, $QEDcons(s_{2n})$ holds. □

VII. Soundness and Conditional Completeness

Definition 13 (Failing and Succeeding QED Tests)**.** *Let $\boldsymbol{i}$ be a QED test, $s_0 \in S_I$ an initial state such that $QEDcons(s_0)$ holds, and let $s = T(s_0, \boldsymbol{i})$. We say that:*

- *QED test $\boldsymbol{i}$* fails *if $\neg QEDcons(s)$.*
- *QED test $\boldsymbol{i}$* succeeds *if $QEDcons(s)$.*

Definition 14 (Processor QED-Consistency)**.** *A processor $\mathcal{P}$ is* QED-consistent *if all possible QED tests succeed.*

Definition 15 (Processor QED-Inconsistency)**.** *A processor $\mathcal{P}$ is* QED-inconsistent *if some QED test fails.*

Lemma 3. *Let $\mathcal{P}$ be a processor. If $\mathcal{P}$ is QED-inconsistent, then $\mathcal{P}$ is not correct with respect to* any *abstract specification relation.*

Proof. Let $\boldsymbol{i}$ be a failing QED test for $\mathcal{P}$ and assume that processor $\mathcal{P}$ is correct with respect to some abstract specification relation $Spec$. By Lemma 2, we conclude $QEDcons(s_{2n})$, which contradicts the assumption that $\boldsymbol{i}$ is a failing QED test. □

Definition 16 (Bug-Specific QED Test)**.** *Let $\mathcal{B} = \langle i_b, S_b\rangle$ be a bug in $\mathcal{P}$ with respect to Spec, where $i_b = (op_b, l^b_{out}, (l^b_{in1}, l^b_{in2}))$. The instruction sequence $\boldsymbol{i} = \langle i_1, \ldots, i_n, i_{n+1}, \ldots, i_{2n}\rangle$ is a* bug-specific QED test *for $\mathcal{B}$ if the following conditions hold:*

1) $i_{n+1} = i_b$.
2) *$\boldsymbol{i}$ is a QED test for some L_D, i.e. for $1 \leq k \leq n$, $i_{n+k} = Dup(i_k)$. In particular, $i_1 = (op_b, l_{out}, (l_{in1}, l_{in2}))$, with $(l_{in1}, l_{in2}, l_{out}) = L_D^{-1}((l^b_{in1}, l^b_{in2}, l^b_{out}))$.*
3) *There exists a path $\boldsymbol{s} \in S^{2n}$ from $s_0 \in S_I$ with $QEDcons(s_0)$, such that $\boldsymbol{s} = T(s_0, \boldsymbol{i}) = \langle s_1, \ldots, s_n, s_{n+1}, \ldots, s_{2n}\rangle$, where $s_n \in S_b$.*
4) *$Spec(s_0, i_1, s_1)$.*
5) *Additionally, we need three more conditions that depend on the bug types:*

Case A: *If i_b is a type-A bug with respect to s_n, i.e. $s_{n+1}(l^b_{out}) \neq Spec_{op_b}(s_n(l^b_{in1}), s_n(l^b_{in2}))$, then let $l_{orig} = l_{out}$ and $l_{dup} = l^b_{out}$.*

- *We then require:*
  - $s_{n+1}(l_{dup}) = s_{2n}(l_{dup})$,
  - $s_1(l_{orig}) = s_{2n}(l_{orig})$,
  - $s_0(L_{in}(i_b)) = s_n(L_{in}(i_b))$.

Case B: *If i_b is a type-B bug with respect to s_n, i.e. $s_n(l_{bad}) \neq s_{n+1}(l_{bad})$ for some $l_{bad} \neq l^b_{out}$, then let $l_{orig} = L_D^{-1}(l_{bad})$ with $l_{orig} \neq l_{out}$ and $l_{dup} = l_{bad}$.*

- *We then require:*
  - $s_{n+1}(l_{dup}) = s_{2n}(l_{dup})$,
  - $s_1(l_{orig}) = s_{2n}(l_{orig})$.
  - $s_1(l_{dup}) = s_n(l_{dup})$,

Lemma 4. *Let $\mathcal{P}$ be a processor with a bug $\mathcal{B} = \langle i_b, S_b\rangle$ with respect to specification Spec, for which there exists a bug-specific QED test $\boldsymbol{i}$. Then $\boldsymbol{i}$ fails.*

Proof. Let $\mathcal{B} = \langle i_b, S_b \rangle$ be a bug and $\boldsymbol{i}$ be a bug-specific QED test for $\mathcal{B}$. By Definition 16 we have $\boldsymbol{i} = \langle i_1, \ldots, i_n, i_{n+1}, \ldots, i_{2n} \rangle$ and $\boldsymbol{s} = T(s_0, \boldsymbol{i}) = \langle s_0, s_1, \ldots, s_n, s_{n+1}, \ldots, s_{2n} \rangle$, where $s_n \in S_b$ and $i_b = i_{n+1}$, and $QEDcons(s_0)$ holds. We show that $\neg QEDcons(s_{2n})$ holds by showing that $s_{2n}(l_{orig}) \neq s_{2n}(l_{dup})$. We distinguish the two cases A and B in Definition 16.

Case A. Since $QEDcons(s_0)$ and $Dup(i_1) = i_b$, we have

$$s_0(L_{in}(i_1)) = s_0(L_{in}(i_b)) \tag{7}$$

From the third requirement of Case A in Definition 16, we have $s_0(L_{in}(i_b)) = s_n(L_{in}(i_b))$, so it follows that,

$$s_0(L_{in}(i_1)) = s_n(L_{in}(i_b)) \tag{8}$$

By (8) and since $op(i_1) = op(i_b)$, also

$$Spec_{op(i_1)}(s_0(L_{in}(i_1))) = Spec_{op(i_b)}(s_n(L_{in}(i_b))) \tag{9}$$

Since $Spec(s_0, i_1, s_1)$ by Definition 16, we have

$$s_1(L_{out}(i_1)) = Spec_{op(i_1)}(s_0(L_{in}(i_1))) \tag{10}$$

Since we are in Case A, we have from Definition 16 that $l_{orig} = L_{out}(i_1)$, and from the second requirement of Case A, we have $s_1(l_{orig}) = s_{2n}(l_{orig})$, so it follows that,

$$s_{2n}(l_{orig}) = Spec_{op(i_1)}(s_0(L_{in}(i_1))) \tag{11}$$

Since i_b fails in state s_n, we have that,

$$s_{n+1}(L_{out}(i_b)) \neq Spec_{op(i_b)}(s_n(L_{in}(i_b))) \tag{12}$$

Again, from Case A in Definition 16, we have $l_{dup} = L_{out}(i_b)$, and from the first requirement of Case A, we have $s_{n+1}(l_{dup}) = s_{2n}(l_{dup})$, so it follows that,

$$s_{2n}(l_{dup}) \neq Spec_{op(i_b)}(s_n(L_{in}(i_b))) \tag{13}$$

Finally, (9) and (11) give us,

$$s_{2n}(l_{orig}) = Spec_{op(i_b)}(s_n(L_{in}(i_b))) \tag{14}$$

But then (13) and (14) imply $s_{2n}(l_{orig}) \neq s_{2n}(l_{dup})$, and hence $\neg QEDcons(s_{2n})$.

Case B. Since $QEDcons(s_0)$, we have

$$s_0(l_{orig}) = s_0(l_{dup}) \tag{15}$$

Since $Spec(s_0, i_1, s_1)$ by Definition 16, we have $s_0(l_{orig}) = s_1(l_{orig})$ and $s_0(l_{dup}) = s_1(l_{dup})$, and so it follows that,

$$s_1(l_{orig}) = s_1(l_{dup}) \tag{16}$$

Due to the requirements in Case B of Definition 16, we have

$$s_{n+1}(l_{dup}) = s_{2n}(l_{dup}) \tag{17}$$

$$s_1(l_{orig}) = s_{2n}(l_{orig}) \tag{18}$$

$$s_1(l_{dup}) = s_n(l_{dup}) \tag{19}$$

Now, because we are in Case B, we know that $s_n(l_{dup}) \neq s_{n+1}(l_{dup})$, so by (17),

$$s_{2n}(l_{dup}) \neq s_n(l_{dup}) \tag{20}$$

But (16), (18) and (19) give us:

$$s_n(l_{dup}) = s_{2n}(l_{orig}) \tag{21}$$

Thus, by (20) and (21),

$$s_{2n}(l_{dup}) \neq s_{2n}(l_{orig}) \tag{22}$$

and hence $\neg QEDcons(s_{2n})$. □

Theorem 1.

- *SQED is sound (Lemma 3).*
- *SQED is complete for bugs for which a bug-specific QED test exists (Lemma 4).*

Theorem 1 is relevant for practical applications of SQED. Referring to the high-level workflow shown in Fig. 1a, BMC symbolically explores all possible QED tests up to bound n for a particular fixed mapping L_D. If a failing QED test $\boldsymbol{i}$ is found, then by the soundness of SQED, $\boldsymbol{i}$ corresponds to a bug in the processor. By completeness, if there exists a bug for which a bug-specific QED test $\boldsymbol{i}$ exists, then with a sufficiently large bound n, BMC will find a sequence $\boldsymbol{i}$ that will fail.

VIII. Symbolic Starting States Proof Sketch

A. Statements

First, we restate the constraints on the symbolic starting state and axioms for a bug-free processor for completeness of this document.

Constraint C_1**:** There exists a time T_C at which all SIF instructions commit (i.e. no SIF instructions can write to the architectural state after T_C) while all Symbolic QED instructions commit after T_C.

Constraint C_2: At T_C, the architectural state is QED consistent. Further, features besides instructions (such as test modes) cannot write to architectural state after T_C (e.g., scan must be turned off for registers).

Constraint C_3: All Symbolic QED operand data of every Symbolic QED instruction I_1, must have one of the following properties:

1) if the operand data is available (i.e., I_1 has already read data for this operand) then this data must match the corresponding register/memory location (i.e., source operand location) data at T_C
2) if the operand data is not available at T_C, then I_1 is waiting for the result of an earlier Symbolic QED instruction for this operand data.

It will be proven that the above constraints form a sufficient condition to guarantee no false positives, given that any bug-free design satisfies the following two axioms after T_C .

Axiom 1:

Let I_1 and I_2 be two Symbolic QED instructions given abstractly as

$I_1 : d\ op\ s_1, s_2, ..., s_m$

$I_2 : d'\ op\ s'_1, s'_2, ..., s'_m$

Here *op* represents an operation performed on data stored in the operand locations (s_1, s'_1 etc.), and the instructions write the computed result to a destination location in the architectural state (e.g., d, d'). Let $data(s_1, I_1)$ be a notation to represent the operand s_1 data used by instruction I_1 for computing the result, (Note the actual data for the source operand may be either obtained from an architectural state, e.g., architectural register, or a micro-architectural state (e.g., by data forwarding). $data(s, I)$ abstracts away these extra details and only represents the data value used by instruction I for operand s.) and $val(T_{I_1}, d)$ represents the value in the architectural location d immediately after instruction I_1 commits at time T_{I_1} (i.e., the value written by instruction I_1 to location d).

Now, Axiom-1 states: If two Symbolic QED instructions I_1, I_2 have the same *op* and $\forall i \in \{1, ..., m\}$, $data(s_i, I_1) = data(s'_i, I_2)$, then $val(T_{I_1}, d) = val(T_{I_2}, d')$.

This Axiom simply states that when a Symbolic QED instruction executes twice on the same data, then it should result in the same outputs.

Axiom 2:

Let I_1 and I_2 be two Symbolic QED instructions given as below:

$I_2 : s_i\ op_2\ s'_1, s'_2, ..., s'_m$

$I_1 : d\ op_1\ s_1, s_2, ..., s_m$

where

$i \in \{1, ..., m\}$ and I_2 is the last Symbolic QED instruction with which I_1 has Read-after-Write (RAW) dependency for operand s_i. Then:

if I_1 , I_2 are as above, $data(s_i, I_1) = val(T_{I_2}, s_i)$.

Further, if there are no earlier Symbolic QED instructions writing to an operand s_i of I_1, where $i \in \{1, ..., m\}$, then $data(s_i, I_1) = val(T_C, s_i)$.

In words, this Axiom states that when any operand of a Symbolic QED instruction has RAW dependency with any earlier Symbolic QED instruction(s), it should obtain the correct source value (which is the result of the last instruction it is dependent on). Also, when an operand of a Symbolic QED instruction does not have dependencies with earlier Symbolic QED instructions, it obtains the correct source value from the architectural state at time T_C, to be used in its computation.

We can now precisely state the main Theorem of the paper.

Theorem 2.

- *Let Constraints C_1, C_2, and C_3 be satisfied by a symbolic starting state of a processor core.*
- *Let Axioms 1 and 2 hold after T_C for any bug-free design of the core (even if initialized at an unreachable starting state).*
- *If any QED property fails, then the failure must be caused by a bug in the design.*

Proof Outline:

See below for the full proof. We first define notation for a sequence of Symbolic QED instructions in a QED-compatible bug trace. Next, we isolate the first pair of Symbolic QED instructions which cause a QED property failure. We decompose the execution of these two instructions into a union of six mutually disjoint cases. For each case, we give a proof by contradiction (of one or more Axioms) that there must be a bug in the design, thus concluding the proof of the Theorem.

Next, a Corollary for partially-correct designs follows.

Corollary 2 (Partially-Correct Designs)**.** *Let Constraints C_1, C_2, and C_3 be satisfied by a symbolic initial state of a processor core. If Axiom-1 holds after T_C for any bug-free design of the core and a QED property fails, then the design cannot satisfy Axiom-2.*

Likewise, if Axiom-2 holds after T_C for any bug-free design of the core and a QED property fails, then the design cannot satisfy Axiom-1.

Proof Outline:

This result follows directly from the proof technique of Theorem-1. We again isolate the first pair of Symbolic QED instructions that causes a QED property failure and decompose the execution of these two instructions into two classes. In the first class, Axiom-1 must be violated and in the second class, Axiom-2 must be violated. The result follows.

B. Full Proofs

Theorem-1 Proof:

Let $I = \{O_1, D_1, O_2, D_2, ..., O_n, D_n\}$ be a sequence of Symbolic QED instructions chosen by the BMC tool that fails at least one QED property, i.e., these instructions are given in a QED-compatible bug trace. Here $\{O_i\}_{i=1}^{n}$ represents the subsequence of original instructions and $\{D_i\}_{i=1}^{n}$ represents the corresponding subsequence of duplicate instructions (i.e., using duplicate registers and memory space). Indexing of instructions in either of the two subsequences corresponds to the order in which they commit to architectural state. We can assume without loss of generality that original and duplicate instructions are pairwise-interleaved [3] as written in I above, and the first failed QED property is due to a mismatch between values in two architectural registers [4].

Now, let T_F [5] be the time when the first QED check fails. Then,

$\exists (R_i, R_i')$ such that $val(T_F, R_i) \neq val(T_F, R_i')$.

Here R_i, R_i' are registers which cause the QED check to fail, and $val(T, R)$ represents the value stored in register R at a given time T. Because Constraint C_2 is satisfied, we have $val(T_C, R_i) = val(T_C, R_i')$. Then, it must be true that

$\exists a \in \{1, ..., n\}$ such that $val(T_{O_a}, R_i) \neq val(T_{D_a}, R_i') \wedge \underset{b \in \{0,...,a-1\}}{\wedge} val(T_{O_b}, R_i) = val(T_{D_b}, R_i')$

where T_{O_a} is the time where O_a writes (commits) to its destination register R_i and T_{D_a} is the time where D_a writes to its destination register R_i'. Explicitly, (O_a, D_a) is the first pair of instructions within I that force the architectural state to be QED inconsistent. The equation (1) is true because there must exist some Symbolic QED instructions which write to the register pair R_i, R_i' such that the QED check fails for these registers, as Constraint C_1 ensures that all SIF instructions complete by T_C.

We can abstractly represent instructions O_a and D_a as below:

$O_a : R_i \ op \ s_1, s_2, ..., s_m$

$D_a : R_i' \ op \ s_1', s_2', ..., s_m'$

Here, *op* represents an operation performed on data stored in some source location [6] (s_1, s_1') etc., and each instruction writes a computed result to its destination register. Because (O_a, D_a) are a pair of Symbolic QED instructions, they implement the same operation. Also, at any time point when the architectural state is QED-consistent (e.g., T_C from C_2), we must have $val(T_C, R_i) = val(T_C, R_i')$, $val(T_C, s_1) = val(T_C, s_1')$ etc.

When O_a and D_a execute, only one of two conditions are satisfied:

(A) $\forall j \in \{1, ..., m\}, data(s_j, O_a) = data(s_j', D_a)$

(B) $\exists j \in \{1, ..., m\}, s.t. data(s_j, O_a) \neq data(s_j', D_a)$

Here, $data(s_j, O_a)$ represents the data which instruction O_a uses for its operand s_j during computation (as defined in Sec. 2.1.).

[3]Interleaving can be $[O_1, O_2, ..., O_n, D_1, D_2, ..., D_n]$, $[O_1, D_1, O_2, D_2, ..., O_n, D_n]$, etc. If the relative orderings of subsequences $\{O_i\}_{i=1}^{n}$ and $\{D_i\}_{i=1}^{n}$ is preserved, there is no impact on proof.

[4]It could alternatively be due to a mismatch in data at two memory locations, with no impact on proof.

[5]T_F is a later time than T_C, as T_F is when all the Symbolic QED instructions commit.

[6]Register or memory locations.

First, assume (A) holds. Then all operand data of O_a and D_a match:

$O_a : R_i \ op \ data_1, data_2, ..., s_m$

$D_a : R_i' \ op \ data_1, data_2, ..., data_m$

However, we also know that $val(T_{O_a}, R_i) \neq val(T_{D_a}, R_i')$. This implies that the same operation (*op*) performed on the same data results in two different values when executed at two different times, contradicting Axiom-1 of any bug-free design. Therefore, for case (A), Theorem-1 holds.

Next, assume (B) holds instead of (A). Depending on when the operands' data are available for the instruction to compute on, we have five mutually disjoint subcases for (B) (for each subcase, we show that Theorem-1 holds):

1) **(B.1)** At T_C, data for both operands s_j and s_j' is available.

   As Constraint C_3 holds, we know that $data(s_j, O_a) = val(T_C, s_j)$ and $data(s_j', D_a) = val(T_C, s_j')$. From Constraint C_2, the architectural state at T_C is QED-consistent. Therefore, $val(T_C, s_j) = val(T_C, s_j')$, which contradicts the assumption of (B) since $data(s_j, O_a) \neq data(s_j', D_a)$. Thus, **(B.1)** cannot arise when QED constraints are in place.
2) **(B.2)** At T_C, data for only one operand (pick s_j without loss of generality) is available.

   If **(B.2)** holds, only one of the following two cases must arise:

   a) **(B.2.1)** There are no earlier Symbolic QED instructions writing to s_j';
   b) **(B.2.2)** There is at least earlier Symbolic QED instruction with which D_a has RAW dependency for source operand s_j'.

   Assume case **(B.2.1)** holds. Then, from Constraints C_2 and C_3, at T_C, $data(s_j, O_a) = val(T_C, s_j)$ and $val(T_C, s_j) = val(T_C, s_j')$. If Axiom-2 holds, we have $data(s_j', D_a) = val(T_C, s_j')$, which implies by transitivity of equality that $data(s_j, O_a) = data(s_j, D_a)$. However, this contradicts the assumption of (B) that $data(s_j, O_a) \neq data(s_j', D_a)$. Thus for case **(B.2.1)**, if a QED property fails, it must be caused by a bug in the design.

   Now assume that **(B.2.2)** holds instead of **(B.2.1)**. Let D_x be the last instruction that D_a has RAW dependency on for source operand s_j' From Axiom-2, we have $data(s_j', D_a) = val(T_{D_x}, s_j')$. Let O_x be the corresponding original QED instructions for D_x. Note that O_a must have a RAW dependency with O_x. Therefore, from Axiom-2 we again have $data(s_j, O_a) = val(T_{O_x}, s_j)$. We also have from (2), $val(T_{O_x}, s_j) = val(T_{D_x}, s_j')$. Hence, by transitivity, $data(s_j, O_a) = data(s_j', D_a)$. However, this contradicts the assumptions of (B), i.e., $data(s_j, O_a) \neq data(s_j', D_a)$. Thus, Axiom-2 cannot hold and for case **(B.2.2)**, if a QED property fails, if is caused by a bug in the design.

3) **(B.3)** At T_C, data for both operands s_j and s'_j are not available.

   If case **(B.3)** holds, only one of two cases must arise:
   a) **(B.3.1)** There is no earlier Symbolic QED instruction pair that write to s_j, s'_j;
   b) **(B.3.2)** There is an earlier Symbolic QED instruction pair O_x, D_x that write to s_j, s'_j that is the last pair O_a, D_a have RAW dependencies on for these operands.

   Next, assume that **(B.3.1)** holds. From Axiom-2, we have $data(s_j, O_a) = val(T_C, s_j)$ and $data(s'_j, D_a) = val(T_C, s'_j)$. From Constraint C_2, we also know that $val(T_C, s_j) = val(T_C, s'_j)$. Thus by transitivity, we have $data(s_j, O_a) = data(s'_j, D_a)$, but this contradicts (B). Hence in case **(B.3.1)** we also conclude that if a QED property fails, it is caused by a bug in the design.
   Finally, assume that **(B.3.2)** holds. From Axiom-2, we know that $data(s_j, O_a) = val(T_{O_x}, s_j)$ and $data(s'_j, D_a) = val(T_{D_x}, s'_j$. From (3) we have $val(T_{O_x}, s_j) = val(T_{D_x}, s'_j)$, and then by transitivity, we have $data(s_j, O_a) = data(s'_j, D_a)$. This contradicts the assumption of (B). Thus for **(B.3.2)** we also conclude that if a QED property fails, it is caused by a bug inside the design.

We have shown that in each of the six possible mutually disjoint cases, a QED property failure can only be caused by a bug in the design. This proves Theorem-1.

Corollary-1 Proof:

Assume a QED property fails and again consider the setup in the proof of Theorem-1 for the QED-compatible bug trace. Again, let (O_a, D_a) be the first pair of instructions within I that force the architectural state to be QED inconsistent. Consider again the representation of these two instructions as O_a and D_a as below:

$O_a : R_i \ op \ s_1, s_2, ..., s_m$

$D_a : R'_i \ op \ s'_1, s'_2, ..., s'_m$

Here, *op* represents an operation performed on data stored in some source locations (s_1, s'_1) etc., and each instruction writes a computed result to its destination register. Because (O_a, D_a) are a pair of Symbolic QED instructions, they implement the same operation. Also, at any time point when the architectural state is QED-consistent (e.g., T_C from C_2), we must have $val(T_C, R_i) = val(T_C, R'_i)$, $val(T_C, s_1) = val(T_C, s'_1)$ etc.

When O_a and D_a execute, only one of two conditions are satisfied:

1) (A) $\forall j \in \{1, ..., m\}, data(s_j, O_a) = data(s'_j, D_a)$
2) (B) $\exists j \in \{1, ..., m\}, s.t. data(s_j, O_a) \neq data(s'_j, D_a)$

In the proof of Theorem-1, we showed that a QED failure in condition (A) cannot happen if Axiom-1 holds, and a QED failure in any of the five subcases of (B) cannot happen if Axiom-2 holds. Therefore, in designs where Axiom-1 must hold, only condition (B) can occur if a QED failure happens, and in designs where Axiom-2 must hold, only condition (A) occurs if a QED failure happens. This proves the result.

IX. Related Work

X. Conclusion and Future Work

References

[1] F. Lonsing, K. Ganesan, M. Mann, S. S. Nuthakki, E. Singh, M. Srouji, Y. Yang, S. Mitra, and C. W. Barrett, "Unlocking the Power of Formal Hardware Verification with CoSA and Symbolic QED: Invited Paper," in *Proc ICCAD*. ACM, 2019, pp. 1–8.

[2] D. Lin, E. Singh, C. Barrett, and S. Mitra, "A structured approach to post-silicon validation and debug using symbolic quick error detection," in *Proc. ITC*. IEEE, 2015, pp. 1–10.

[3] M. R. Fadiheh, J. Urdahl, S. S. Nuthakki, S. Mitra, C. Barrett, D. Stoffel, and W. Kunz, "Symbolic quick error detection using symbolic initial state for pre-silicon verification," in *Proc. DATE*. IEEE, 2018, pp. 55–60.

[4] E. Singh, K. Devarajegowda, S. Simon, R. Schnieder, K. Ganesan, M. R. Fadiheh, D. Stoffel, W. Kunz, C. W. Barrett, W. Ecker, and S. Mitra, "Symbolic QED Pre-Silicon Verification for Automotive Microcontroller Cores: Industrial Case Study," in *Proc. DATE*. IEEE, 2019, pp. 1000–1005.

[5] E. Singh, D. Lin, C. Barrett, and S. Mitra, "Logic bug detection and localization using symbolic quick error detection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2018.

[6] R. M. Keller, "A Fundamental Theorem of Asynchronous Parallel Computation," in *Parallel Processing, Proc. Sagamore Computer Conference*, ser. LNCS, vol. 24. Springer, 1974, pp. 102–112.

[7] R. M. Keller, "Formal Verification of Parallel Programs," *Commun. ACM*, vol. 19, no. 7, pp. 371–384, 1976.

[8] B. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification," *ACM Trans. Design Autom. Electr. Syst.*, vol. 24, no. 1, pp. 10:1–10:24, 2019.